Lecture Notes in Computer Science 2508
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Dahlia Malkhi (Ed.)

Distributed Computing

16th International Conference, DISC 2002
Toulouse, France, October 28-30, 2002
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Dahlia Malkhi
The Hebrew University of Jerusalem
School of Computer Science and Engineering
Jerusalem 91904, Israel
E-mail: dalia@cs.huji.ac.il

Cataloging-in-Publication Data applied for

Bibliograhpic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at http://dnb.ddb.de

CR Subject Classification (1998): C.2.4, C.2.2, F.2.2, D.1.3, F.1.1, D.4.4-5

ISSN 0302-9743
ISBN 3-540-00073-9 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna e.K.
Printed on acid-free paper   SPIN: 10870889   06/3142   5 4 3 2 1 0

Preface

The International Symposium on DIStributed Computing (DISC) 2002 was held in Toulouse, France, on October 28–30, 2002. The realization of distributed systems on numerous fronts and the proliferation of the Internet continue to make the symposium highly important and relevant.

There were 76 regular submissions to DISC this year, which were read and evaluated by program committee members assisted by external reviewers. Twenty-four papers were selected by the program committee to be included in these proceedings. The quality of submissions was high, and the committee had to decline some papers worthy of publication.

The best student paper award was selected from among regular submissions that were not co-authored by any program committee member, and was given to Yongqiang Huang for his contribution "Assignment-Based Partitioning in a Condition Monitoring System", co-authored with Hector Garcia-Molina.

October 2002 Dahlia Malkhi

Organizing Committee

Mamoun Filali
Philippe Mauran
Gèrard Padiou (Chair)
Philippe Quèinnec
Anne-Marie Zerr (Secretary)

Steering Committee

Andrè Schiper, Chair (EPFL, Switzerland)
Michel Raynal, Vice-chair (Irisa, France)
Maurice Herlihy (Brown University, USA)
Dahlia Malkhi, Program Chair (Hebrew University of Jerusalem, Israel)
Jennifer Welch (Texas A&M University, USA)
Alex Shvartsman (University of Connecticut, USA)
Shay Kutten (Technion, Israel)

Program Committee

Gustavo Alonso (ETH Zurich, Switzerland)
Roberto Baldoni (University of Rome, Italy)
Dave Bakken (Washington State University, USA)
Paul Ezhilchelvan (University of Newcastle, UK)
Christof Fetzer (AT&T Labs-Research, USA)
Faith Fich (University of Toronto, Canada)
Roy Friedman (Technion, Israel)
Juan A. Garay (Bell Labs–Lucent Technologies, USA)
Yuh-Jzer Joung (National Taiwan University, Taiwan)
Nancy Lynch (MIT, USA)
Dahlia Malkhi, Chair (Hebrew University of Jerusalem, Israel)
Michael Merritt (AT&T Labs-Research, USA)
Dan Raz (Technion, Israel)
Robbert van Renesse (Cornell University, USA)
Gadi Taubenfeld (Open University and IDC, Israel)
Masafumi Yamashita (Kyushu University, Japan)

External Referees

Miriam Alluluf
Tal Anker
Roberto Beraldi
Geir Bjune
David Breitgand
Ranveer Chandra
Grisha Chockler
Nick Cook
Lenore J. Cowen
Tarana Damania
Roberto de Prisco
Ioanna Dionysiou
Oren Dobzinsky
Shlomi Dolev
Kevin Dorow
Andy Franz
Kjell Harald Gjermundrod
Rachid Guerraoui
Carl Hauser
Jean-Michel Helary
Maurice Herlihy
Keren Horowitz
Hirotsugu Kakugawa
Idit Keidar
Nils Klarlund
Wesley Lawrence
Victor Luchangco
Anna Lysyanskaya
Carlo Marchetti
Ofer Margo
Toshimitsu Masuzawa
A. David McKinnon
Radek Mista
Mark Moir
Graham Morgan
Ty Palmer
Elan Pavlov
David Peleg
Michel Raynal
Ohad Rodeh
Yaron Sella
Jayavel Shanmugasundaram
Vitaly Shmatikov
Ilya Shnaiderman
Santosh Shrivastava
Gurdip Singh
Emin Gun Sirer
Will Stephenson
Scott Stoller
Ichiro Suzuki
Anat Talmy
Yih-Kuen Tsay
Sara Tucci Piergiovanni
Werner Vogels
John Warne
Rebecca Wright
Zhen Xiao

Table of Contents

Early-Delivery Dynamic Atomic Broadcast
(Extended Abstract)*

Ziv Bar-Joseph[1], Idit Keidar[2], and Nancy Lynch[1]

[1] MIT Laboratory for Computer Science
zivbj@mit.edu, lynch@theory.lcs.mit.edu
[2] Technion Department of Electrical Engineering
idish@ee.technion.ac.il

Abstract. We consider a problem of atomic broadcast in a dynamic setting where processes may join, leave voluntarily, or fail (by stopping) during the course of computation. We provide a formal definition of the *Dynamic Atomic Broadcast* problem and present and analyze a new algorithm for its solution in a variant of a synchronous model, where processes have approximately synchronized clocks.
Our algorithm exhibits constant message delivery latency in the absence of failures, even during periods when participants join or leave. To the best of our knowledge, this is the first algorithm for totally ordered multicast in a dynamic setting to achieve constant latency bounds in the presence of joins and leaves. When failures occur, the latency bound is linear in the number of actual failures. Our algorithm uses a solution to a variation on the standard distributed consensus problem, in which participants do not know a priori who the other participants are. We define the new problem, which we call *Consensus with Uncertain Participants*, and give an early-deciding algorithm to solve it.

1 Introduction

We consider a problem of atomic broadcast in a *dynamic setting* where an unbounded number of participants may join, leave voluntarily, or fail (by stopping) during the course of computation. We formally define the *Dynamic Atomic Broadcast (DAB)* problem, which is an extension of the Atomic Broadcast problem [13] to a setting with infinitely many processes, any finite subset of which can participate at a given time. Just as Atomic Broadcast is a basic building block for state machine replication in a static setting, DAB can serve as a building block for state machine replication among a dynamic set of processes.

We present and analyze a new algorithm, which we call *Atom*, for solving the DAB problem in a variant of a synchronous crash failure model. Specifically, we assume that the processes solving DAB have access to approximately-synchronized local clocks and to a lower-level network that guarantees timely

* Full version appears as MIT Technical Report MIT-LCS-TR-840 [3]. This work supported by MURI AFOSR Award F49620-00-1-0327, AFOSR Contract F49620-00-1-0097, NTT Contract number MIT9904-12, and NSF Contract CCR-0121277.

message delivery among currently active processes. The challenge is to guarantee consistency among the sequences of messages delivered to different participants, while still achieving timely delivery, even in the presence of joins and leaves.

Atom exhibits *constant* message delivery latency in the absence of failures, *even during periods when participants join or leave*; this is in contrast to previous algorithms solving similar problems in the context of view-oriented group communication, e.g., [1,9]. When failures occur, Atom's latency bound is linear in the number of failures that actually occur; it does not depend on the number of potential failures, nor on the number of joins and leaves that occur.

A key difficulty for an algorithm solving DAB is that when a process fails, the network does not guarantee that the surviving processes all receive the same messages from the failed process. But the strong consistency requirements of DAB dictate that processes agree on which messages they deliver to their clients. The processes carry out a protocol to coordinate message delivery, which works roughly as follows: Each Atom process divides time into *slots*, using its local clock, and assigns each message sent by its client to a slot. Each process delivers messages to its client in order of slots, and within each slot, in order of sender identifiers. Each process determines the *membership* of each slot, and delivers messages only from senders that it considers to be members of the slot. To ensure consistency, the processes must agree on the membership of each slot.

Processes joining (or voluntarily leaving) the service coordinate their own join (or leave) by selecting a join-slot (or leave-slot) and informing the other processes of this choice, without delaying the normal delivery of messages. When a process fails, Atom uses a novel *distributed consensus service* to agree upon the slot in which it fails. The consensus service required by Atom differs from the standard stopping-failure consensus services studied in the distributed algorithms literature (see, e.g., [16]) in that the processes implementing the consensus service do not know a priori who the other participants are. Atom tracks process joins and leaves, and uses this information to approximate the active set of processes that should participate in consensus. However, different processes running Atom may have somewhat different perceptions of the active set, e.g., when a participant joins or leaves Atom at roughly the time consensus is initiated.

In order to address such uncertainties, we define a new consensus service, *consensus with uncertain participants (CUP)*. When a process i initiates CUP, it submits to CUP a finite set W_i estimating the current world, in addition to i's proposed initial consensus value v_i. The worlds suggested by different participants do not have to be identical, but some restrictions are imposed on their consistency. Consider, e.g., the case that process k joins Atom at roughly the time CUP is initiated. One initiator, i, may think that k has joined in time to participate and include k in W_i, while another, j, may exclude k from W_j. Process k cannot participate in the CUP algorithm in the usual way, because j would not take its value into account. On the other hand, if k does not participate at all, i could block, waiting forever for a message from k. We address such situations by allowing k to explicitly *abstain* from an instance of CUP, i.e., to participate without providing an input. A service that uses CUP must ensure the following *world consistency* assumption: that for every i, (1) W_i includes all the processes that ever initiate this instance of CUP (unless they fail or leave prior to i's initiation); and (2) if $j \in W_i$, (and neither i nor j fail or leave), then j participates in CUP either by initiating or by abstaining. Thus, W_i sets can differ only in the inclusion of processes that abstain, leave, or fail. Note that once an instance of CUP has been started, no new processes (that are not included in W_i) can join the running instance. Nevertheless, CUP provides a good abstraction for solving DAB, because Atom can invoke multiple instances of CUP with different sets of participants.

We give an early-deciding algorithm to solve CUP in a fail-stop model, that is, in an asynchronous crash failure model with perfect failure detectors. The failure detector is external to CUP; it is implemented by Atom. CUP uses a strategy similar to previous early-deciding consensus algorithms [10], but it also tolerates uncertainty about the set of participants, and moreover, it allows processes to leave voluntarily without incurring additional delays. The time required to reach consensus is linear in the number of failures that actually occur during an execution, and does not depend on the number of potential failures.

We also analyze the message-delivery latency of Atom under different failure assumptions. We show a constant latency bound for periods when no failures occur, even if joins and leaves occur. When failures occur, the latency is proportional to the number of actual failures. This is inevitable: atomic broadcast requires a number of rounds that is linear in the number of failures.

We envision a service using Atom, or a variation of it, deployed in a large LAN, where latency is predictable and message loss is bounded. In such settings, a network with the properties we assume can be implemented using forward error correction (see [2]), or retransmissions (see [20]). The algorithm can be extended for use in environments with looser time guarantees, e.g., networks with differentiated services; we outline ideas for such an extension in Section 7.4.

In summary, this paper makes the following main contributions: (1) the definitions of two new services for dynamic networks: DAB and CUP; (2) an early-delivery DAB algorithm, Atom, which exhibits constant latency in the absence failures; (3) a new early-deciding algorithm for CUP; and (4) the analysis of Atom's message-delivery latency under various failure assumptions.

The rest of this paper is organized as follows: Section 2 discusses related work. In Section 3, we specify the DAB service. In Section 4 we specify CUP and in Section 5, we present the CUP algorithm. Section 6 specifies the environment assumptions for Atom, and Section 7 presents the Atom algorithm. Section 8 concludes the paper. In the full paper [3], we present the algorithms in pseudocode and prove their correctness.

2 Related Work

Atomic broadcast in a dynamic universe, where processes join and leave, was first considered in the context of view-oriented group communication systems (GCSs) [6], pioneered by Isis [4]. Our service resembles those provided by GCSs;

although we do not export membership to the application, it is computed and would be easy to export.

GCSs, including those designed for synchronous systems and real-time applications (e.g., Cristian's [9], xAMp [18], and RTCAST [1]), generally run a group membership protocol every time a process joins or leaves, and therefore delay message delivery to all processes when joins or leaves occur. Cristian's service exhibits constant latency only in periods in which no joins or failures occur; latency during periods with multiple joins is not analyzed. xAMp is a GCS supporting a variety of communication primitives for real-time applications. The presentation of xAMp in [18] assumes that a membership service is given. The delays due to failures and joins are incurred in the membership part, which is not described or analyzed. The latency bound achieved by RTCAST is *linear* in the number of processes, even when no process fails, due to the use of a logical ring. Moreover, RTCAST makes stronger assumptions about its underlying network than we do – it uses an underlying reliable broadcast service that guarantees that correct processes deliver the same messages from faulty ones.

Light-weight group membership services [11] avoid running the full-scale membership for join and leaves by using atomic broadcast to disseminate join and leave messages in a consistent manner. Unlike our CUP service, the atomic broadcast services used by such systems do not tolerate uncertainty about the participants. Therefore, a race condition between a join and a concurrent failure can cause such light-weight group services (e.g., [11]) to violate consistency. Those light-weight group services that do preserve consist membership semantics (e.g., [19]), do incur extra delivery latencies whenever joins and leaves occur.

Other work on group membership in synchronous and real-time systems, e.g., [15,14] has focused on membership maintenance in a static, fairly small, group of processes, where processes are subject to failures but no new processes can join the system. Likewise, work analyzing time bounds of synchronous atomic broadcast, e.g., [12,8,7], considered a static universe, where processes could fail but not join. Thus, this work did not consider the DAB problem.

In a previous paper [2], we considered a simpler problem of dynamic totally ordered broadcast without all or nothing semantics. For this problem, the linear lower bound does not apply, and we exhibited an algorithm that solves the problem in constant time even in the presence of failures.

Recent work [17,5] considers different services, including (one shot) consensus, for infinitely many processes in asynchronous shared memory models. Chockler and Malkhi [5] present a consensus algorithm for infinitely many processes using a *static* set of active disks, a minority of which can fail. This differs from the model considered here, as in our model all system components may be ephemeral. Merritt and Taubenfeld [17] study consensus under different concurrency models, and show that if there is no bound on the number of participants, in an asynchronous shared memory model, solving consensus requires infinitely many bits. The algorithms they give tolerate only initial failures. To the best of our knowledge, atomic broadcast has not been considered in a similar context.

3 Dynamic Atomic Broadcast Service Specification

The universe is an infinite ordered set of endpoints, I; M is a message alphabet. Figure 1 presents DAB's signature. We assume that an application using DAB satisfies some basic well-formedness assumptions, (cf. [3]), e.g., that a process does not join / leave more than once, does not multicast messages before a join or after a leave, and does not send the same message more than once. We do not consider rejoining; instead, we consider the same client joining at new endpoints.

```
Input: joini, leavei, faili, i∈I        Output: join_OKi, leave_OKi, i∈I
       mcasti(m), m∈M, i∈I                      rcvi(m), m∈M, i∈I
```

Fig. 1. The signature of the DAB service.

We require that there be a total ordering $\mathcal{S}$ on all the messages received by any of the endpoints, such that for all $i \in I$, the following properties are satisfied.

- *Multicast order:* If $\texttt{mcast}_i\texttt{(m)}$ occurs before $\texttt{mcast}_i\texttt{(m')}$, then m precedes m' in $\mathcal{S}$.
- *Receive order:* If $\texttt{rcv}_i\texttt{(m)}$ occurs before $\texttt{rcv}_i\texttt{(m')}$ then m precedes m' in $\mathcal{S}$.
- *Multicast gap-freedom:* If $\texttt{mcast}_i\texttt{(m)}$, $\texttt{mcast}_i\texttt{(m')}$, and $\texttt{mcast}_i\texttt{(m'')}$ occur, in that order, and $\mathcal{S}$ contains m and m'', then $\mathcal{S}$ also contains m'.
- *Receive gap-freedom:* If $\mathcal{S}$ contains m, m', and m'', in that order, and $\texttt{rcv}_i\texttt{(m)}$ and $\texttt{rcv}_i\texttt{(m'')}$ occur, then $\texttt{rcv}_i\texttt{(m')}$ also occurs.
- *Multicast liveness:* If $\texttt{mcast}_i\texttt{(m)}$ occurs and no $\texttt{fail}_i$ occurs, then $m \in \mathcal{S}$.
- *Receive liveness:* If $m \in \mathcal{S}$, m is sent by i and i does not leave or fail, then $\texttt{rcv}_i\texttt{(m)}$ occurs, and for every m' that follows m in $\mathcal{S}$, $\texttt{rcv}_i\texttt{(m')}$ also occurs.

In addition to the above, DAB is required to satisfy basic integrity properties, e.g., that $\texttt{join\_OK}_i$ ($\texttt{leave\_OK}_i$) must be preceded by a $\texttt{join}_i$ ($\texttt{leave}_i$); that every $\texttt{join}_i$ ($\texttt{leave}_i$) is followed by a $\texttt{join\_OK}_i$ ($\texttt{leave\_OK}_i$); and that messages are not received more than once, and are not received unless they are multicast. The formal definitions of these appear in [3].

4 Consensus with Uncertain Participants – Specification

In order to solve DAB, we use the CUP service. CUP is an adaptation of the problem of fail-stop uniform consensus to a setting in which the set of participants is not known precisely ahead of time, and in which participants can leave the algorithm voluntarily after initiating it. Moreover, participants are not assumed to initiate at the same time. CUP assumes an underlying reliable network, and a perfect failure detector. The signature of the CUP service is presented in Figure 2; I is a universe as above; V is a totally ordered set of possible consensus *values*; and M_{CUP} is a message alphabet.

```
Input: init_i(v,W), v ∈ V, W ⊆ I, W finite, i ∈ I   // i initiates
       abstain_i, i ∈ I                              // i abstains
       net_rcv_i(m), m ∈ M_CUP, i ∈ I                // i receives message m
       leave_i, i ∈ I                                // i leaves
       leave_detect_i(j), j, i ∈ I                   // i detects that j has left
       fail_i, i ∈ I                                 // i fails
       fail_detect_i(j), j, i ∈ I                    // i detects that j failed

Output: decide_i(v), v ∈ V, i ∈ I                    // i decides on value v
        net_mcast_i(m), m ∈ M_CUP, i ∈ I             // i multicasts m
```

Fig. 2. The signature of CUP.

A process i may participate in CUP in two ways: it may *initiate* CUP using $\texttt{init}_i\texttt{(v,W)}$ and provide an initial value and an initial world, or it may *abstain* (using $\texttt{abstain}_i$). Informally speaking, a participant abstains when it does not need to participate in CUP, but because of uncertainty about CUP participants, some other participant may expect it to participate. CUP reports the consensus decision value to process i using the $\texttt{decide}_i\texttt{(v)}$ action. The environment provides a *leave detector* and a *failure detector*: $\texttt{leave\_detect}_i\texttt{(j)}$ notifies i that j has left the algorithm voluntarily, and $\texttt{fail\_detect}_i\texttt{(j)}$ notifies i that j has failed. In Section 4.1 we specify assumptions about CUP's environment; assuming these hold, CUP satisfies the following properties:

– *Uniform Agreement:* For any $i, j \in I$, if $\texttt{decide}_i\texttt{(v)}$ and $\texttt{decide}_j\texttt{(v')}$ both occur then $v = v'$.
– *Validity:* For any $i \in I$, if $\texttt{decide}_i\texttt{(v)}$ occurs then (1) for some j, $\texttt{init}_j\texttt{(v,*)}$ occurs; and (2) if $\texttt{init}_i\texttt{(v',*)}$ occurs then $v \leq v'$.
– *Termination:* If $\texttt{init}_i$ occurs, then a $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs.

The validity condition (2) is not a standard property for consensus but is needed for our use in Atom. Another difference from standard consensus is that participants that abstain need not be informed of the decision value.

In addition to these properties, CUP satisfies a well-formedness condition saying that only participants that have initiated can decide, and each participant decides at most once (cf. [3]).

4.1 CUP Environment Assumptions

CUP requires some simple well-formedness conditions saying that each participant begins participating (by initiating or abstaining) at most once, leaves at most once, and fails at most once. CUP also makes standard integrity assumptions about the underlying network, namely that every message that is received was previously sent, and no message is received at the same location more than once. Moreover, the order of message receipt between particular senders and receivers is FIFO. We now specify the more interesting environment assumptions.

The following assumptions are related to the worlds W suggested by participants in their init events. The first is a safety assumption saying that each W set submitted by an initiating participant i must include all participants that ever initiate CUP and that do not leave or fail prior to the $\texttt{init}_i$ event. This implies that every participant must be included in its own estimated world. The next is a liveness assumption saying that, if any process i expects another process j to participate, then j will actually do so, unless either i or j leaves or fails.

- *World consistency:* If $\texttt{init}_i\texttt{(*, W)}$ and $\texttt{init}_j\texttt{(*,*)}$ events occur, then either $j \in W$, or a $\texttt{leave}_j$ or $\texttt{fail}_j$ event occurs before the $\texttt{init}_i\texttt{(*, W)}$ event.
- *Init occurrence:* If an $\texttt{init}_i\texttt{(*,W)}$ event occurs and $j \in \texttt{W}$, then an $\texttt{init}_j$, $\texttt{abstain}_j$, $\texttt{leave}_i$, $\texttt{fail}_i$, $\texttt{leave}_j$, or $\texttt{fail}_j$ occurs.

The next assumptions are related to leaves, leave detection, and failure detection. The second property says that leaves are handled gracefully, in the sense that the occurrence of a $\texttt{leave\_detect}_i\texttt{(j)}$ implies that i has already received any messages sent by j prior to leaving. Thus, a $\texttt{leave\_detect}_i\texttt{(j)}$ is an indication that i has not lost any messages from j. Note that we do not have a failure assumption analogous to the lossless leave property; thus, failures are different from leaves in that we allow the possibility that some messages from failed processes may be lost.

- *Accurate leave detector:* For any $i, j \in I$, at most one $\texttt{leave\_detect}_i\texttt{(j)}$ event occurs, and if it occurs, then it is preceded by a $\texttt{leave}_j$.
- *Lossless leave:* Assume $\texttt{net\_mcast}_j\texttt{(m)}$ occurs and is followed by a $\texttt{leave}_j$. Then if a $\texttt{leave\_detect}_i\texttt{(j)}$ occurs, it is preceded by $\texttt{net\_rcv}_i\texttt{(m)}$.
- *Accurate failure detector:* For any $i, j \in I$, at most one $\texttt{fail\_detect}_i\texttt{(j)}$ event occurs, and if it occurs, then it is preceded by a $\texttt{fail}_j$.
- *Complete leave and failure detector:* If $\texttt{init}_i\texttt{(*,W)}$ occurs, $j \in W$, and $\texttt{leave}_j$ or $\texttt{fail}_j$ occurs, then $\texttt{fail\_detect}_i\texttt{(j)}$, $\texttt{leave\_detect}_i\texttt{(j)}$, $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs.

The next liveness assumption describes reliability of message delivery. It says that any message that is multicast by a non-failing participant that belongs to any of the W sets submitted to CUP, is received by all the non-leaving, non-failing members of all those W sets.

- *Reliable delivery:* Define $\texttt{U} = \cup_{k \in I}\{\ \texttt{W} \mid \texttt{init}_k\texttt{(*, W)}\ \ \texttt{occurs}\}$. If $i, j \in \texttt{U}$ and $\texttt{net\_mcast}_i\texttt{(m)}$ occurs after an $\texttt{init}_i$ or $\texttt{abstain}_i$ event, then a $\texttt{net\_rcv}_j\texttt{(m)}$, $\texttt{leave}_j$, $\texttt{fail}_i$, or $\texttt{fail}_j$ occurs.

5 The CUP Algorithm

The algorithm proceeds in asynchronous rounds, $1, 2, \ldots$. In each round, a process sends its current estimates of the value and the world to the other processes. Each process maintains two-dimensional arrays, value and world, with the value and world information it receives from all processes in all rounds. It records, in a variable `out[r]`, the processes that it knows will not participate in round `r` because they have left, abstained, or decided, and in a variable `failed[r]`, the processes that it learned have failed in this round.

When $\texttt{init}_i\texttt{(v,W)}$ occurs, process i triggers `net_mcast(i,1,v,W)` to send its initial value `v` and estimated world `W` to all processes, including itself. Note that two separate processes, A and B can initiate CUP with different (overlapping) subsets of processes in their `W` parameter. For example, it could be that A has `W = {A,B,C}` while B has `W = {A,B,D}`. However, in this case we are guaranteed from the *World consistency* and *Init occurrence* assumptions that C either fails prior to $\texttt{init}_B\texttt{(v,W)}$, or abstains, and the same holds for A and D. The world `W` is determined to be the set of processes that i thinks are still active, i.e., the processes in i's previous world that i does not know to be out or to have failed in round `r`. Process i may perform this multicast only if its round is `r-1`, it has received round `r-1` messages from all the processes in `W`, and it is not currently able to decide. The value `v` that is sent is the minimum value that i has recorded for round `r-1` from a process in `W`. When a $\texttt{net\_rcv}_i\texttt{(j,r,v,W)}$ occurs, process i puts `v` and `W` into the appropriate places in the `value` and `world` arrays.

Process i can decide at a round `r` when it has received messages from all processes in its `world[r,i]` except those that are out at round `r`, such that all of these messages contain the same value and contain worlds that are subsets of `world[r,i]`. The subset requirement ensures that processes in `world[r,i]` will not consider values from processes outside of `world[r,i]` in determining their values for future rounds. When process i decides, it multicasts an `OUT` message and stops participating in the algorithm.

When $\texttt{abstain}_i$ occurs, process i also sends an `OUT` message, so that other processes will know not to wait for further messages from it, and stops participating in the algorithm. When a $\texttt{net\_rcv}_i\texttt{(j,OUT)}$ occurs, process i records that j is out of the algorithm starting from the first round for which i has not yet received a regular message from j.

When $\texttt{leave}_i$ occurs, i just stops participating in the algorithm. When a $\texttt{leave\_detect}_i\texttt{(j)}$ event occurs, i records that j is out from the first round after the round of the last message received from j. The lossless leave assumption ensures that i has already received all the messages j sent. Process i knows that process j has failed if $\texttt{fail\_detect}_i\texttt{(j)}$ occurs.

In [3] we prove that when CUP's environment satisfies CUP's safety assumptions, CUP satisfies its safety guarantees, and when CUP's environment satisfies CUP's safety and liveness assumptions, CUP satisfies its liveness guarantees.

5.1 Analysis

The algorithm is early-deciding in the sense that the number of rounds it executes is proportional to the number of actual failures that occur, and does not depend on the number of participants or on the number of processes that leave. In [3], we prove the following theorem, which says that the algorithm always terminates after it can run two rounds without failures.

Theorem 1. *Suppose that* `r` $>$ `0`. *Suppose that there is a point t in the execution such that every process is in* `round` $\leq$ `r` *at point t, and no* `fail` *events happen from t onward. Then every process always has* `round` $\leq$ `r` $+2$.

The proof is based on the observation that once failures stop, the values and worlds that processes send in their round messages stop changing; the value converges to the minimum value that a live process has, and the world converges to the set of live processes. After a round in which all processes in `W` send the same value and the world `W`, all the live processes can decide.

We next analyze CUP's running time assuming the following bounds on message latency, failure and leave detection times, and the difference between different processes' initiation times. Note: time bounds are not assumed as a condition for correctness; they are only assumed for the sake of the analysis.

1. δ_1 is an upper bound on *message latency* and on *failure and leave detection time.* Moreover, if a message is lost due to failure, then the failure is detected at most δ_1 after the lost message was sent. More precisely,
   (a) if `net_rcv(m)` occurs, the time since the corresponding `net_mcast(m)` is at most δ_1.
   (b) Assume $\texttt{init}_i\texttt{(*,W)}$ occurs with $j \in W$ and $\texttt{fail}_j$ or $\texttt{leave}_j$ occurs at time t. Then $\texttt{fail\_detect}_i\texttt{(j)}$, $\texttt{leave\_detect}_i\texttt{(j)}$, $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs by time $t + \delta_1$.
   (c) Let $U = \cup_{k \in I}\{W|\ \texttt{init}_k(*, W) \text{ occurs}\}$. Assume $i, j \in U$ and $\texttt{net\_mcast}_j\texttt{(m)}$ occurs at time t but no $\texttt{net\_rcv}_i\texttt{(m)}$ occurs. Then either $\texttt{fail\_detect}_i\texttt{(j)}$, $\texttt{leave\_detect}_i\texttt{(j)}$, $\texttt{decide}_i$, $\texttt{leave}_i$, or $\texttt{fail}_i$ occurs by time $t + \delta_1$.
2. δ_2 bounds the difference between the initiation time of different processes. More precisely:
   Assume a process initiates at time t and does not fail by time $t+\delta_1$, and that $\texttt{init}_i\texttt{(*, W)}$ occurs. Then, every process $j \in W$ initiates, abstains, leaves, or fails by time $t + \delta_2$.

Given these bounds, in [3], we prove the following theorem:

Theorem 2. *Suppose that there is a point t in the execution such that no* `fail` *events happen from t onward. Suppose also that some process initiates CUP by time t. Then every process that decides, decides by time $t + 3\delta_1 + \delta_2$.*

6 Environment and Model Assumptions for Atom

We model time using a continuous global variable `now`, which holds the real time. This is a real variable, initially 0. Each endpoint i is equipped with a local clock, $\texttt{clock}_i$. We assume a bound of Γ on clock skew, where Γ is a positive real number. Specifically, for each endpoint i, we assume that in any state of the system that is reachable $|\ clock_i -\ now| \leq \Gamma/2$. That is, the difference between each local clock and the real time is at most $\Gamma/2$. It follows that the clock skew between any pair of processes is Γ, formally: in any reachable state, and for any two endpoints i and j, $|\ clock_i -\ clock_j| \leq \Gamma$. We assume that local processing time is 0 and that actions are scheduled immediately when they are enabled.

We assume that we are given a low-level reliable network service Net, with a message alphabet, M'. The Net signature is defined in Figure 3. The actions are the same as those of DAB, except that they are prefixed with `net_`.

Input: `net_join`$_\mathtt{i}$`, net_leave`$_\mathtt{i}$`, i∈I`	**Output:** `net_join_OK`$_\mathtt{i}$`, i∈I,`
`fail`$_\mathtt{i}$`, i∈I`	`net_leave_OK`$_\mathtt{i}$`, i∈I,`
`net_mcast`$_\mathtt{i}$`(m), m∈M′, i∈I,`	`net_rcv`$_\mathtt{i}$`(m), m∈M′, i∈I`

Fig. 3. The signature of the Net service.

Like DAB, Net assumes that its application satisfies the some basic integrity conditions. Assuming these, Net satisfies a number of safety and liveness properties. First, Net satisfies the basic integrity properties that DAB does. In addition, Net guarantees FIFO delivery of messages, and a simple liveness property:

– FIFO *delivery:* If `net_mcast`$_i$`(m)` occurs before `net_mcast`$_i$`(m')`, and `net_rcv`$_j$`(m')` occurs, then `net_rcv`$_j$`(m)` occurs before `net_rcv`$_j$`(m')`.
– *Eventual delivery:* Suppose `net_mcast`$_i$`(m)` occurs after `net_join_OK`$_j$, and no `fail`$_i$ occurs. Then either `net_leave`$_j$ or `fail`$_j$ or `net_rcv`$_j$`(m)` occurs.

Additionally, the network latency is bounded by a constant nonnegative real number Δ. The maximum message latency of Δ guaranteed by Net is intended to include any pre-send delay at the network module of the sending process, and is independent of the message size. Since an implementation of Net cannot predict the future, it must deliver messages within time Δ as long as no failures occur. In particular, if a message is sent more than Δ time before its sender fails, it must be delivered.

7 The Atom Algorithm

The Atom algorithm uses Net and CUP services as building blocks. It uses multiple instances of CUP. As before, `fail`$_i$ causes process i to stop. `fail`$_i$ is an input to all the components, i.e., Net and all instances of CUP (including dormant ones), and causes all of them to stop; `leave`$_i$ also goes directly to all the local instances of CUP, including dormant ones.

Atom defines the constant Θ, a positive real number that represents the duration of a time slot. We assume that $\Theta > \Delta$. We define the message alphabet M' of Net in term of the alphabet M of DAB:

– M_1, the set of finite sequences of elements of M. These are the bulk messages processes send.
– $M_2 = M_1 \cup \{JOIN, LEAVE\} \cup \{CUP\_INIT \times I\}$
– $M' = I \times M_2 \times \mathsf{N}$.

Each message contains either a bulk message (sequence of client messages) for a particular slot, a request to join or leave a particular slot, or a report that process has initiated consensus on behalf of a particular endpoint. Each message is tagged with the sender and the slot. The algorithm divides time and messages into slots, each of duration Θ. Each process multicasts all of its messages for a given slot in one bulk message. This is an abstraction that we make in order to simplify the presentation. In practice, the bulk message does not have to be sent as one message; a standard packet assembly/disassembly layer can be used.

Message delivery is also done in order of slots. Before delivering messages of a certain slot `s`, each process has to determine the *membership* of `s`, i.e., the set of processes from which to deliver slot `s` messages. To ensure consistency, all the processes that deliver messages for a certain slot have to agree upon its membership. Within each slot, messages are delivered in order of process indices, and for each process, messages from its bulk message are delivered in FIFO order.

7.1 Atom$_i$ Signature and Variables

The signature of Atom$_i$ includes all the interaction with the client and underlying network. In addition, Atom$_i$ has input and output actions for interacting with CUP. Since Atom uses multiple instances of CUP, at most one for each process j, actions of CUP automata are prefixed with `CUP(j)`, where `CUP(j)` is the instance of CUP used to agree in which slot process j fails. E.g., process i uses the action `CUP(j).init`$_i$ to initiate the CUP automaton associated with process j. `CUP.fail` and `CUP.leave` are *not* output actions of Atom, since they are routed directly from the environment to all instances of CUP.

Atom$_i$ also has two internal actions, `end_slot`$_i$, and `members`$_i$, which play a role in determining the membership of each slot. `end_slot(s)`$_i$ occurs at a time by which slot `s` messages from all processes should have reached i. At this point, processes from which messages are expected but do not arrive are *suspected* to have failed in this slot; we are guaranteed that these processes indeed have failed, but we are uncertain about the slot in which they fail. For each suspected process j, CUP(j) is run to have the surviving processes agree upon j's failure slot. This is needed because failed processes can be suspected at different slots by different surviving processes. After CUP reaches decisions about all the suspected processes that could have failed at slot `s`, `members(P,s)` can occur, with `P` being the agreed membership for slot `s`. When `members(P,s)`$_i$ occurs, the messages included in bulk messages that i received for slot `s` from processes in `P` are delivered (their delivery is triggered) in order of process indices.

A variable `join-slot` holds the slot at which a process starts participating in the algorithm; this will be the value of `current-slot` when `join_OK` will be issued, and the first slot for which a bulk message will be sent. If a process explicitly leaves the algorithm, its `leave-slot` holds the slot immediately following the last slot in which the process sends a bulk message. Both `join-slot` and `leave-slot` are initially ∞.

The flags `did-join-OK` and `did-leave` ensure that `join_OK` and `net_leave` actions are not performed more than once. The set `mcast-slots` tracks the slots for which the process already multicast a message (JOIN, LEAVE, or bulk). Likewise, `ended-slots` and `reported-slots` track the slots for which the `end_slot` or `members` actions, resp., were performed.

`out-buf[s]` stores the message (bulk, JOIN, or LEAVE) that is multicast for slot `s`; it initially holds an empty sequence, and in an active slot, all application messages are appended into it. A JOIN message is inserted for the slot before the `join-slot`, and a LEAVE message for the `leave-slot`. Either way, there is no overlap with a bulk message. Variables `joiners[s]` and `leavers[s]` keep track of the processes j for which `join-slot`$_j$ `=s` (resp. `leave-slot`$_j$ `=s`). `suspects[s]` is the set of processes suspected in slot `s` as determined when `end_slot(s)` occurs. `in-buf[j,s]` holds the sequence of messages received in a slot `s` bulk message from j. Its data type supports assignment, extraction of the head of the queue, and testing for emptiness. `alive[s]` is a derived variable containing the set of processes from which slot `s` messages were received.

There are three variables for tracking the status and values of the different instances of CUP. `CUP-status[j]` is initially `idle`; when CUP(j) is initiated, it becomes `running`; if a CUP_INIT message for j arrives, it becomes `req`; and when there is a decision for CUP(j), or if the process abstains from CUP(j), it becomes `done`. `CUP-req-val[j]` holds the lowest slot value associated with a CUP_INIT message for j ($\perp$ if no such message has arrived). Finally, `CUP-dec-val[j]` holds the decision reached by CUP(j), and $\perp$ if there is none.

7.2 Algorithm Flow

Upon an application `join`, Atom triggers `net_join`. Once the Net responds with a `net_join_OK`, Atom calculates the `join-slot` to be $2 + \lceil \Gamma/\Theta \rceil$ slots in the future. This will allow enough time for the JOIN message to reach the other processes. A JOIN message is then inserted into `out-buf[join-slot - 1]`. Once `current-slot` is `join-slot`, `join_OK` is issued to the application.

When the application issues a `leave`, the `leave-slot` is chosen to be the ensuing slot, and a LEAVE message is inserted into `out-buf[leave-slot]`. A `net_leave` is issued after the LEAVE message has been multicast, and the `net_leave_OK` triggers a `leave_OK` to the application.

Messages multicast by the application are appended to the bulk message for the current slot in `out-buf[current-slot]`. Once a slot `s` ends, the message pertaining to this slot is multicast using `net_mcast`. If `s = join-slot - 1`, a JOIN message is sent. If `s = leave-slot`, a LEAVE message is sent, and if `s` is between `join-slot` and `leave-slot - 1`, a bulk message is sent. A received bulk message is stored in the appropriate `in-buf`. When a (j, JOIN, s) (or (j, LEAVE, s)) message is received, j is added to `joiners[s]` (resp. `leavers[s]`). Additionally, when a LEAVE message is received, `CUP.leave_detect` is triggered for all running instances of CUP.

`end_slot`$_i$`(s)` occurs once i should have received all the slot `s` messages sent by non-failed processes. Since such messages are sent immediately when slot `s` ends, are delayed at most Δ time in Net, and the clock difference is at most Γ, i should have all the slot `s` messages $\Delta + \Gamma$ time after it began slot `s+1`. Process *i* expects to receive slot `s` messages from every process in `alive[s-1]` that does not leave in slot `s`. Any process from which a slot `s` message is expected but does not arrive becomes suspected at this point, and is included in `suspects[s]`.

For every suspected process, CUP is run in order to agree upon the slot at which the process failed. Note that CUP is only performed for failed processes since we implement a perfect failure detector. The slot `s` in which the process is suspected is used as the initial value for CUP. The estimated world for CUP is `alive[s]` $\cup$ `joiners[s+1]`. This way, if *k* joins in slot `s+1`, *k* is included in the estimated world. This is needed in order to satisfy the world consistency assumption of CUP, because *k* can detect the same failure at slot `s+1`, and therefore participate in CUP(j). When *i* initiates CUP(j), it also multicasts a (CUP_INIT, j) message. If a process *k* does not detect the failure and does not participate, the (CUP_INIT, j) message forces *k* to abstain. Since Atom implements the failure detector for CUP, the effect of `end_slot`$_i$`(s)` also triggers `CUP(k).fail_detect(j)` actions for every suspected process *j*, and for every currently running instance *k* of CUP.

Process *i* abstains from CUP(j) only if (1) a (CUP_INIT,*j*) message has previously arrived, setting `CUP-status[j]`$_i$ `= req`; and (2) `end_slot`$_i$ has already occurred for a slot value greater than `CUP-req-val[j]`$_i$. The latter condition ensures that *i* abstains only from instances of CUP that it will not initiate. There are two circumstances that can lead to a process *i* abstaining from CUP(j). First, if *i* is just joining, and the failure occurs before its `join-slot`, then *i* will not be affected by the decision because it does not deliver any messages for this slot. Second, if *j* has joined and immediately failed before *i* could see its JOIN message, then *j* did not send any bulk messages prior to its failure, and thus no process will deliver any messages from *j*.

The `members(P,s)` action triggers the delivery of all slot `s` messages from processes in `P`. It occurs once agreement is reached about the processes to be included in `P`. Recall that the slots at which a process *k* is suspected by two processes *i* and *j* can differ by at most one. Therefore, `members`$_i$`(P,s)` can occur after `end_slot(s+1)`, when the suspicions for slot `s+1` are determined, since all processes that *i* does not suspect at slot `s+1` could not have failed prior to ending slot `s`. Thus, after *i* gets decisions from all instances of CUP pertaining to processes suspected in slots up to `s+1` *i* can deliver all slot `s` messages. The set `P` includes every process *j* that is alive in slot `s` and for which there is either no CUP instance running, or the CUP decision value is greater than `s`.

In [3] we prove the following: (1) Atom satisfies CUP's safety assumptions independently of CUP; (2) assuming a service that satisfies the CUP safety guarantees, Atom satisfies CUP's liveness assumptions; and (3) assuming a service that satisfies the CUP safety and liveness guarantees, Atom satisfies the DAB service safety and liveness guarantees.

7.3 Latency Analysis

In failure free executions, Atom's message latency is bounded by $\Delta + 2\Theta + 2\Gamma$. We denote this bound by Δ_{Atom}. In executions with failures, the upper bound on message latency is linear in the number of failures. In [3], we prove the following:

Lemma 1. *Consider an execution in which no process fails. If the application at process j performs* `mcast`$_j$`(m)` *when* `current-slot`$_i$ `= s` *and if process i delivers m, then i delivers m immediately after* `end_slot`$_i$`(s+1)` *occurs.*

From this lemma, we derive the following theorem:

Theorem 3. *If the application at process j performs* `mcast`$_j$`(m)` *at time t, and if process i delivers m, then i delivers m by time $t + \Delta_{Atom} = t + \Delta + 2\Theta + 2\Gamma$.*

We then turn our attention to executions in which there is a long time period with no failures. We analyze the time it takes Atom to clear the backlog it has due to past failures, and reach a situation in which message latency is bounded by the same bound as in failure free executions, namely Δ_{Atom}, barring additional failures. The fact that once failures stop for a bounded time all messages are delivered within constant time implies that in periods with f failures, Atom's latency is at most linear in the number of failing processes.

In order to analyze how long it takes Atom to reach a stable point, we need to use our bounds on CUP's running time once failures stop. We first have to assign values to the constants that were used in the analysis of CUP in Section 5.1 (δ_1 and δ_2). Recall, δ_1 is an upper bound on message latency and on failure and leave detection time, and if a message is lost due to failure, then the failure is detected at most δ_1 after the lost message was sent; and δ_2 is an upper bound on the difference between different processes' initiation times. In [3], we prove the following bounds: $\delta_1 = \Delta + 3\Theta + 2\Gamma$; and $\delta_2 = \Gamma + \Theta$.

We then consider executions in which failures do occur but there are long time periods with no failures. We analyze the time it takes Atom to clear the backlog it has due to past failures, and again reach a situation in which message latency is bounded by Δ_{Atom}, barring additional failures.

Let $t_1 = \delta_2 + 4\delta_1$, where δ_2 and δ_1 are bounds as given above for the difference between process initiation times and failure detection time, resp. By the bounds above, we get that $t_1 = \Gamma + \Theta + 4(\Delta + 3\Theta + 2\Gamma) = 4\Delta + 9\Gamma + 13\Theta$.

Assume that from time t to time $t' = t + t_1$ there are no failures. We now show that if a message m is sent after time t', and there are no failures for a period of length Δ_{Atom} after m is sent, then m is delivered within Δ_{Atom} time of when it is sent. Since the delivery order preserves the FIFO order, this also implies that any message m' sent before time t' is delivered by time t' barring failures in the Δ_{Atom} time interval after m' is sent.

Theorem 4. *Assume no process fails between time t and $t' = t + t_1$. If* `mcast(m)`$_j$ *occurs at a time t'' such that $t + t_1 \leq t''$, and no failures occur from time t'' to time $t'' + \Delta_{Atom}$, and if i delivers m, then i delivers m by time $t'' + \Delta_{Atom}$.*

7.4 Future Direction: Extending Atom to Cope with Late Messages

In this paper, we assumed deterministic network latency bounds. Since the network latency, Δ is expected to be of a smaller order of magnitude than Θ, it would not significantly hurt time bounds if conservative assumptions are made in the choice of Δ. Future research may consider networks where latency bounds cannot be ensured. E.g., some networks may support differentiated services with probabilistic latency guarantees, and loss rates may exceed those assumed in the latency analysis of the underlying reliable network (see [2,20]).

Although Atom cannot guarantee atomic broadcast semantics while network latency exceeds its bound, it would be useful to modify Atom as to allow it to recover from such situations, and to once more provide correct semantics after network guarantees are re-established. In addition, it would be desirable to inform the application when a violation of Atom semantics occurs. There are some strategies that can be used to make Atom recover from periods in which network guarantees are violated. For example, a lost or late message can cause inaccurate failure suspicions. With Atom, if a process k is falsely suspected, it will receive a (CUP_INIT, k) message for itself. We can have the process use this as a trigger to "commit suicide", i.e., inform the application of the failure and have the application re-join as a new process (similar to the Isis [4] algorithm).

8 Conclusions

We have defined two new problems, *Dynamic Atomic Broadcast* and *Consensus with Uncertain Participants*. We have presented new algorithms for both problems. The latency of both of our algorithms depends linearly on the number of failures that occur during a particular execution, but does not depend on an upper bound on the potential number of failures, nor on the numbers of joins and leaves that happen during the execution.

Acknowledgments. We thank Alan Fekete, Rui Fan, Rachid Guerraoui, and the referees for useful comments that helped improve the presentation.

References

1. T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCAST: Lightweight multicast for real-time process groups. In *IEEE Real-Time Technology and Apps. Symp. (RTAS)*, Jun 1996.
2. Z. Bar-Joseph, I. Keidar, T. Anker, and N. Lynch. QoS preserving totally ordered multicast. In Franck Butelle, editor, *5th Int. Conf. On Prin. Of Dist. Sys. (OPODIS)*, pp. 143–162, Dec 2000. Special issue of *Studia Informatica Universalis.*
3. Z. Bar-Joseph, I. Keidar, and N. Lynch. Early-delivery dynamic atomic broadcast. Tech. Rep. MIT-LCS-TR-840, MIT Lab. for Comp. Sci., Apr 2002.
4. K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Comp. Soc. Press, 1994.

5. G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. In *21st ACM Symp. on Prin. of Dist. Comp. (PODC)*, July 2002. To appear.
6. G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Comp. Surveys*, 33(4):1–43, Dec 2001.
7. F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Journal of Real-Time Systems*, 2:195–212, 1990.
8. F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Inf. Comp.*, 118:158–179, Apr 1995.
9. F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Dist. Comp.*, 4(4):175–187, Apr 1991.
10. D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *Journal of the ACM*, 37(4):720–741, Oct 1990.
11. B. Glade, K. Birman, R. Cooper, and R. van Renesse. Lightweight process groups in the Isis system. *Dist. Sys. Eng.*, 1:29–36, 1993.
12. A. Gopal, R. Strong, S. Toueg, and F. Cristian. Early-delivery atomic broadcast. In *9th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 297–309, 1990.
13. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *chapter in: Distributed Systems*. ACM Press, 1993.
14. S. Katz, P. Lincoln, and J. Rushby. Low-overhead time-triggered group membership. In *11th Int. Wshop. on Dist. Algs. (WDAG)*, pp. 155–169, 1997.
15. H. Kopetz and G. Grunsteidl. TTP - a protocol for fault-tolerant real-time systems. *IEEE Computer*, pp. 14–23, January 1994.
16. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
17. M. Merritt and G. Taubenfeld. Computing with infinitely many processes under assumptions on concurrency and participation. In *14th Int. Symp. on DIStributed Comp. (DISC)*, Oct 2000.
18. L. Rodrigues and P. Verissimo. xAMp, A multi-primitive group communications service. In *IEEE Int. Symp. on Reliable Dist. Sys. (SRDS)*, pp. 112–121, Oct 1992.
19. L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, and K. Birman. A dynamic light-weight group service. In *15th IEEE Int. Symp. on Reliable Dist. Sys. (SRDS)*, pp. 23–25, Oct 1996.
20. P. Verissimo, J. Rufino, and L. Rodrigues. Enforcing real-time behaviour of lan-based protocols. In *10th IFAC Wshop. on Dist. Comp. Control Systems*, Sep 1991.

Secure Computation without Agreement (Extended Abstract)[*]

Shafi Goldwasser and Yehuda Lindell

Department of Computer Science and Applied Math,
Weizmann Institute of Science, Rehovot, Israel.
{shafi,lindell}@wisdom.weizmann.ac.il

Abstract. It has recently been shown that executions of authenticated Byzantine Agreement protocols in which more than a third of the parties are corrupted, cannot be composed concurrently, in parallel, or even sequentially (where the latter is true for deterministic protocols). This result puts into question any usage of authenticated Byzantine agreement in a setting where many executions take place. In particular, this is true for the whole body of work of secure multi-party protocols in the case that 1/3 or more of the parties are corrupted. Such protocols strongly rely on the extensive use of a broadcast channel, which is in turn realized using authenticated Byzantine Agreement. Essentially, this use of Byzantine Agreement cannot be eliminated since the standard definition of secure computation (for the case that less than 1/2 of the parties are corrupted) actually implies Byzantine Agreement. Moreover, it was accepted folklore that the use of a broadcast channel is essential for achieving secure multiparty computation, when 1/3 or more of the parties are corrupted.

In this paper we show that this folklore is false. We mildly relax the definition of secure computation allowing abort, and show how this definition can be reached. The difference between our definition and previous ones is as follows. Previously, if one honest party aborted then it was required that all other honest parties also abort. Thus, the parties *agree* on whether or not the protocol execution terminated successfully or not. In our new definition, it is possible that some parties abort while others receive output. Thus, there is no agreement regarding the success of the protocol execution. We stress that in all other aspects, our definition remains the same. In particular, if an output is received it is guaranteed to have been computed correctly. The novelty of the new definition is in *decoupling* the issue of agreement from the central security issues of privacy and correctness in secure computation. As a result the lower bounds of Byzantine Agreement no longer apply to secure computation. Indeed, we prove that secure multi-party computation can be achieved for any number of corrupted parties and without a broadcast channel (or trusted preprocessing phase as required for running authenticated Byzantine Agreement). An important corollary of our result is the ability to obtain multi-party protocols that compose.

[*] A full version of this paper can be found on the IACR Cryptology ePrint Archive, Report 2002/040, http://eprint.iacr.org

1 Introduction

In the setting of secure multi-party computation, a set of n parties with private inputs wish to jointly and securely compute a function of their inputs. This computation should be such that each party receives its correct output, and none of the parties learn anything beyond their prescribed output. This encompasses computations as simple as coin-tossing and agreement, and as complex as electronic voting, electronic auctions, electronic cash schemes, anonymous transactions, and private information retrieval schemes.

1.1 Ground Rules of the 80's

This problem was initiated and heavily studied in the mid to late 80's, during which time the following ground rules were set.

Security in multi-party computation. A number of different definitions were proposed for secure multi-party computation. These definitions aimed to ensure a number of important security properties. The most central of these are:

- *Privacy:* No party should learn anything more that its prescribed output.
- *Correctness:* The outputs received by the parties are guaranteed to be correct.
- *Independence of Inputs:* Corrupted parties' inputs are committed to independently of honest parties' inputs.
- *Guaranteed output delivery:* Corrupted parties should not be able to prevent honest parties from receiving their output. (This is not always possible and is therefore not always required.)
- *Fairness:* Corrupted parties should receive output only if honest parties do. (As with the previous item, this is not always achievable and is therefore not always fully required.)

The standard definition today [17,1,24,5] formalizes the above requirements in the following way. Consider an ideal world in which an external trusted party is willing to help the parties carry out their computation. An ideal computation takes place in the ideal world by having the parties simply send their inputs to the trusted party. This trusted party then computes the desired function and passes each party its prescribed output. Notice that all of the above security properties (and more) are ensured in this ideal computation. A real protocol that is run by the parties (in a world where no trusted party exists) is said to be secure, if no adversary controlling a coalition of corrupted parties can do more harm in a real execution that in the above ideal computation.

Broadcast: In the construction of protocols, the ability to "broadcast" messages (if needed) was assumed as a primitive, where broadcast takes on the meaning of the Byzantine Generals problem [22]. Namely, an honest party can deliver a message of its choice to all honest parties in a given round. Furthermore, all honest parties will receive the same message, even if the broadcasting party is corrupt. Let t be the number of corrupted parties controlled by the adversary. Then, from results obtained largely by the distributed computing community, it was known that:

1. For $t < n/3$, Byzantine agreement is possible by a deterministic protocol with round complexity $O(t)$ [25], and by a probabilistic protocol with expected round complexity $O(1)$ [10];
2. For $t \geq n/3$, broadcast is achievable using a protocol for *authenticated* Byzantine agreement, in which a public-key infrastructure for digital signatures is used [25,22]. (This public-key infrastructure is assumed to be setup in a trusted preprocessing phase.) We note that an information theoretic analogue also exists [26]. The round complexity of the above protocols is $O(t)$.

Assuming broadcast as a primitive in a point-to-point network was seen as non-problematic. This is because Byzantine Agreement is achievable for all values of t (with the added requirement of a trusted preprocessing phase in the case of $t \geq n/3$).

Fairness: As we have mentioned above, fairness is also considered as an important goal in secure computation. Since the basic notion of fairness is not achievable for all values of t, it takes on different meanings for different values of t. We will single out a few forms of fairness. On the one extreme, we have "complete fairness" that guarantees that if a corrupt party gets its output then all honest parties also get their output. On the other extreme, we have "no fairness" in which the adversary always gets its output and has the power to decide whether or not the honest parties also get output. An intermediate notion that we call "partial fairness" singles out a specified party such that if this specified party is honest then complete fairness is achieved. On the other hand, if the specified party is corrupt, then no fairness is achieved. Thus, fairness is partial.

1.2 Feasibility of Secure Computation

Wide-reaching results, demonstrating the feasibility of secure computation, were also presented in the late 80's. The most central of these are as follows:

1. For $t < n/3$, secure multi-party protocols with complete fairness (and guaranteed output delivery), can be achieved in a point-to-point network and without any setup assumptions. This can be achieved both in the information theoretic setting assuming private channels [4,8], and in the computational setting (assuming the existence of trapdoor permutations).[1]
2. For $t < n/2$, secure multi-party protocols with complete fairness (and guaranteed output delivery) can be achieved assuming the existence of a broadcast channel. This can be achieved in the infomation theoretic setting [27]

[1] The protocol of [16] uses oblivious transfer which can in turn be constructed from trapdoor permutations. Alternatively, one can transform the protocol of [4] to the computational model by encrypting all messages sent between players with public-key encryption. This transformation assumes the existence of public-key encryption only.

and in the computational setting [16] with the same assumptions as above. Alternatively, without assuming a broadcast channel, it can be achieved in a point to point network assuming a trusted pre-processing phase for setting up a public-key infrastructure (which is then used for running authenticated Byzantine Agreement).

3. For $t \geq n/2$, secure multi-party protocols with partial fairness can be achieved assuming a broadcast channel or a trusted pre-processing phase (as in case (2)), and in addition the existence of oblivious transfer [16,19,20]. Some works attempting to provide higher levels of fairness (e.g., ensuring that the corrupted parties progress at the same rate towards their output as the honest parties) also appeared [28,14,17,2].
   We note that in this case (of $t \geq n/2$) it is impossible to guarantee output delivery (even given a broadcast channel). Therefore, this property is not required (and some parties may not receive output at all).

We note that all of the above results consider a stand-alone execution of a multi-party protocol only.

1.3 Byzantine Agreement and Secure Computation

There is a close connection between Byzantine agreement and secure multi-party computation. First, Byzantine agreement (or broadcast) is used as a basic and central tool in the construction of secure protocols. In particular, all the feasibility results above assume a broadcast channel (and implement it using Byzantine agreement or authenticated Byzantine agreement). Second, Byzantine agreement is actually a special case of secure computation (this holds by the standard definition taken for the case that $t < n/2$ where output delivery is guaranteed). Therefore, all the lower bounds relating to Byzantine agreement immediately apply to secure multi-party computation. In particular, the Byzantine agreement problem cannot be solved for any $t \geq n/3$ [25]. Thus, it is also impossible to achieve general secure computation with guaranteed output delivery in a point-to-point network for $t \geq n/3$. On the other hand, for $t < n/2$ it *is* possible to obtain secure computation with guaranteed output delivery assuming a broadcast channel. This means that in order to achieve such secure computation for the range of $n/3 \leq t < n/2$, either a physical broadcast channel or a trusted pre-processing phase for running authenticated Byzantine agreement *must* be assumed.

More recently, it was shown that authenticated Byzantine agreement cannot be composed (concurrently or even in parallel), unless $t < n/3$ [23]. This has the following ramifications. On the one hand, in the range of $n/3 \leq t < n/2$, it is *impossible* to obtain general secure computation that composes without using a physical broadcast channel. This is because such a protocol in the point-to-point network model and with trusted pre-processing would imply authenticated Byzantine agreement that composes. On the other hand, as we have mentioned, in the range of $t \geq n/2$ the definitions of secure computation do not imply Byzantine agreement. Nevertheless, all protocols for secure computation in this range make extensive use of a broadcast primitive. The impossibility of composing authenticated Byzantine agreement puts this whole body of work into question when composition is required. Specifically without using a physical broadcast channel, none of these protocols compose (even in parallel). In summary, the current state of affairs is that there are no protocols for secure computation in a point-to-point network that compose in parallel or concurrently, for any $t \geq n/3$. Needless to say, the requirement of a physical broadcast channel is very undesirable (and often unrealistic).

1.4 Our Results

We present a mild relaxation of the standard definition of secure multi-party computation that decouples the issue of *agreement* from the issue of *secure multi-party computation.* In particular, our definition focuses on the central issues of privacy and correctness. Loosely speaking, our definition is different in the following way. As we have mentioned, for the case of $t \geq n/2$, it is impossible to guarantee output delivery and therefore some parties may conclude with a special abort symbol $\perp$, and not with their output. Previously [15], it was required that either *all* honest parties receive their outputs or *all* honest parties output $\perp$.[2] Thus the parties all *agree* on whether or not output was received. On the other hand, in our definition some honest parties may receive output while some receive $\perp$, and the requirement of agreement is removed. We stress that this is the only difference between our definition and the previous ones.

We show that it is possible to achieve secure computation according to the new definition for *any* $t < n$ and *without* a broadcast channel or setup assumption (assuming the same computational assumptions made, if any, by corresponding protocols that did use broadcast channels.) Thus, the lower bounds for Byzantine agreement indeed do not imply lower bounds for secure multi-party computation. We note that our results hold in both the information theoretic and computational models.

A hierarchy of definitions. In order to describe our results in more detail, we present a hierarchy of definitions for secure computation. All the definition fulfill the properties of privacy and correctness. The hierarchy that we present here relates to the issues of abort (or failure to receive output) and fairness.

1. *Secure computation without abort:* According to this definition, all parties are guaranteed to receive their output. (This is what we previously called "guaranteed output delivery".) This is the standard definition for the case of honest majority (i.e., $t < n/2$). Since all honest parties receive output, complete fairness is always obtained here.
2. *Secure computation with unanimous abort*: In this definition, it is ensured that either all honest parties receive their outputs or all honest parties abort. This definition can be considered with different levels of fairness:

[2] We note that in private communication, Goldreich stated that the requirement in [15] of having all parties abort or all parties receive output was only made in order to simplify the definition.

   a) *Complete fairness:* Recall that when complete fairness is achieved, the honest parties are guaranteed to receive output if the adversary does. Thus, here one of two cases can occur. Either all parties receive output or all parties abort. Thus, the adversary can conduct a denial of service attack, but nothing else. (This definition can only be achieved in the case of $t < n/2$.)
   b) *Partial fairness:* As in the case of complete fairness, the adversary may disrupt the computation and cause the honest parties to abort without receiving their prescribed output. However, unlike above, the adversary may receive the corrupted parties' outputs, even if the honest parties abort (and thus the abort is not always fair). In particular, the protocol *specifies* a single party such that the following holds. If this party is honest, then complete fairness is essentially achieved (i.e., either all parties abort or all parties receive correct output). On the other hand, if the specified party is corrupt, then fairness may be violated. That is, the adversary receives the corrupted parties' outputs first, and then decides whether or not the honest parties all receive their correct output or all receive abort (and thus the adversary may receive output while the honest parties do not).
      Although fairness is only guaranteed in the case that the specified party is not corrupted, there are applications where this feature may be of importance. For example, in a scenario where one of the parties may be "more trusted" than others (yet not too trusted), it may be of advantage to make this party the specified party. Another setting where this can be of advantage is one where all the participating parties are trusted. However, the problem that may arise is that of an external party "hacking" into the machine of one of the parties. In such a case, it may be possible to provide additional protection to the specified party.
   c) *No fairness:* This is the same as in the case of partial fairness except that the adversary always receives the corrupted parties' outputs first (i.e., there is no specified party).

   We stress that in all the above three definitions, if one honest party aborts then so do all honest parties, and thus all are aware of the fact that the protocol did not successfully terminate. This feature of having all parties succeed or fail together may be an important one in some applications.
3. *Secure computation with abort:* The only difference between this definition and the one immediately preceding it, is that some honest parties may receive output while others abort. That is, the requirement of unanimity with respect to abort is removed. This yields two different definitions, depending on whether partial fairness or no fairness is taken. (Complete fairness is not considered here because it only makes sense in a setting where all the parties, including the corrupted parties, either all receive output or all abort. Therefore, it is not relevant in the setting of secure computation with non-unanimous abort.)

Using the above terminology, the definition proposed by Goldreich [15] for the case of any $t < n$ is that of secure computation with unanimous abort and partial fairness. Our new definition is that of secure computation with abort, and as we have mentioned, its key feature is a decoupling of the issues of secure computation and agreement (or unanimity).

Achieving secure computation with abort. Using the terminology introduced above, our results show that secure computation with abort and partial fairness can be achieved for any $t < n$, and without a broadcast channel or a trusted pre-processing phase. We achieve this result in the following way. First, we define a weak variant of the Byzantine Generals problem, called *broadcast with abort,* in which not all parties are guaranteed to receive the broadcasted value. In particular, there exists a single value x such that every party either outputs x or aborts. Furthermore, when the broadcasting party is honest, the value x equals its input, similarly to the validity condition of Byzantine Generals. (Notice that in this variant, the parties do not necessarily agree on the output since some may output x while others abort.) We call this "broadcast with abort" because as with secure computation with abort, some parties may output x while other honest parties abort. We show how to achieve this type of broadcast with a simple deterministic protocol that runs in 2 rounds. Secure multi-party computation is then achieved by replacing the broadcast channel in known protocols with a broadcast with abort protocol. Despite the weak nature of agreement in this broadcast protocol, it is nevertheless enough for achieving secure multi-party computation with abort. Since our broadcast with abort protocol runs in only 2 rounds, we also obtain a very efficient transformation of protocols that work with a broadcast channel into protocols that require only a point-to-point network. In summary, we obtain the following theorem:

Theorem 1. (efficient transformation): *There exists an efficient protocol compiler that receives any protocol Π for the* broadcast model *and outputs a protocol Π' for the* point-to-point model *such that the following holds: If Π securely computes a functionality f with unanimous abort and with any level of fairness, then Π' securely computes f with abort and with no fairness. Furthermore, if Π tolerates up to t corruptions and runs for R rounds, then Π' tolerates up to t corruptions and runs for $O(R)$ rounds.*

Notice that in the transformation of Theorem 1, protocol Π' does not achieve complete fairness or partial fairness, even if Π did. Thus, fairness may be lost in the transformation. Nevertheless, meaningful secure computation is still obtained and at virtually no additional cost.

When obtaining some level of fairness is important, Theorem 1 does not provide a solution. We show that partial fairness *can* be obtained without a broadcast channel for the range of $t \geq n/2$ (recall that complete fairness cannot be obtained in this range, even with broadcast). That is, we prove the following theorem:

Theorem 2. (partial fairness): *For any probabilistic polynomial-time n-party functionality f, there exists a protocol in the point-to-point model for computing f that is secure with abort, partially fair and tolerates any $t < n$ corruptions.*

The theorem is proved by first showing that fairness can be boosted in the point-to-point model. That is, given a generic protocol for secure multi-party computation that achieves no fairness, one can construct a generic protocol for secure multi-party computation that achieves partial fairness. (Loosely speaking, a generic protocol is one that can be used to securely compute any efficient functionality.) Applying Theorem 1 to known protocols for the broadcast model, we obtain secure multi-party computation that achieves no fairness. Then, using the above "fairness boosting", we obtain Theorem 2. We note that the round complexity of the resulting protocol is of the same order of the "best" generic protocol that works in the broadcast model. In particular, based on the protocol of Beaver et al. [3], we obtain the first constant-round protocol in the point-to-point network for the range of $n/3 \leq t < n/2$.[3] That is:

Corollary 1. (constant round protocols without broadcast for $t < n/2$): *Assume that there exist public-key encryption schemes (or, alternatively, assume the existence of one-way functions and a model with private channels). Then, for every probabilistic polynomial-time functionality f, there exists a* constant round *protocol in the point-to-point network for computing f that is secure with abort, partially fair and tolerates $t < n/2$ corruptions.*

Composition of secure multi-party protocols. An important corollary of our result is the ability to obtain secure multi-party protocols for $t > n/3$ that compose in parallel or concurrently, without a broadcast channel. Until now, it was not known how to achieve such composition. This is because previously the broadcast channel in multi-party protocols was replaced by authenticated Byzantine agreement, and by [23] authenticated Byzantine Agreement does *not* compose even in parallel. (Authenticated Byzantine agreement was used because for $t > n/3$ standard Byzantine agreement cannot be applied.) Since we do not need to use authenticated Byzantine agreement to obtain secure computation, we succeed in bypassing this problem.

Discussion. We propose that the basic definition of secure computation should focus on the issues of privacy and correctness (and independence of inputs). In contrast, the property of agreement should be treated as an additional, and not central, feature. The benefit of taking such a position (irrespective of whether one is convinced conceptually) is that the feasibility of secure computation is completely decoupled from the feasibility of Byzantine agreement. Thus, the lower bounds relating to Byzantine agreement (and authenticated Byzantine agreement) do not imply anything regarding secure computation. Indeed, as we show, "broadcast with abort" is sufficient for secure computation. However, it lacks any flavor of agreement in the classical sense. This brings us to an important observation. Usually, proving a lower bound for a special case casts

[3] For the range of $t < n/3$, the broadcast channel in the protocol of [3] can be replaced by the expected constant-round Byzantine agreement protocol of Feldman and Micali [10]. However, there is no known authenticated Byzantine agreement protocol with analogous round complexity.

light on the difficulties in solving the general problem. However, in the case of secure computation this is not the case. Rather, the fact that the lower bounds of Byzantine agreement apply to secure computation is due to marginal issues relating to unanimity regarding the delivery of outputs, and not due to the main issues of security.

1.5 Related Work

Two recent independent results [12,13] study a problem similar to ours, although apparently for different motivation. They construct protocols for weak Byzantine Agreement for the cases of $t \geq n/3$ and then use this to obtain secure computation without the use of a broadcast channel. In short, they achieve secure computation with unanimous abort whereas we achieve secure computation with abort. However, our protocols are significantly more round efficient. See the full version of this paper for a careful and detailed comparison [18].

1.6 Organization

Due to lack of space in this extended abstract, we omit the formal description of the hierarchy of definitions of secure computation outlined in Section 1.4. We also omit the proofs of our constructions. We refer the reader to the full version of our paper for these and other details [18].

2 Broadcast with Abort

In this section, we present a weak variant of the Byzantine Generals problem, that we call *"broadcast with abort"*. The main idea is to weaken both the agreement and validity requirements so that some parties may output the broadcast value x while others output $\perp$. Formally,

Definition 1. (broadcast with abort): *Let $P_1, \ldots, P_n$, be n parties and let P_1 be the dealer with input x. In addition, let $\mathcal{A}$ be an adversary who controls up to t of the parties (which may include P_1). A protocol solves the* broadcast with abort *problem, tolerating t corruptions, if for any adversary $\mathcal{A}$ the following three properties hold:*

1. Agreement: *If an honest party outputs x', then all honest parties output either x' or $\perp$.*
2. Validity: *If P_1 is honest, then all honest parties output either x or $\perp$.*
3. Non-triviality: *If all parties are honest, then all parties output x.*

(The non-triviality requirement is needed to rule out a protocol in which all parties simply output $\perp$ and halt.) We now present a simple protocol that solves the broadcast with abort problem for *any* t. As we will see later, despite its simplicity, this protocol suffices for obtaining secure computation with abort.

Protocol 1 (broadcast with abort):

- **Input:** *P_1 has a value x to broadcast.*
- **The Protocol:**
  1. *P_1 sends x to all parties.*
  2. *Denote by x^i the value received by P_i from P_1 in the previous round. Then, every party P_i (for $i > 1$) sends its value x^i to all other parties.*
  3. *Denote the value received by P_i from P_j in the previous round by x^i_j (recall that x^i denotes the value P_i received from P_1 in the first round). Then, P_i outputs x^i if this is the only value that it saw (i.e., if $x^i = x^i_2 = \cdots = x^i_n$). Otherwise, it outputs $\perp$.*
     We note that if P_i did not receive any value in the first round, then it always outputs $\perp$.

We now prove that Protocol 1 is secure for any number of corrupted parties. That is,

Proposition 1. *Protocol 1 solves the broadcast with abort problem, and tolerates* any *$t < n$ corruptions.*

Proof. The fact that the non-triviality condition is fulfilled is immediate. We now prove the other two conditions:

1. *Agreement:* Let P_i be an honest party, such that P_i outputs a value x'. Then, it must be that P_i received x' from P_1 in the first round (i.e., $x^i = x'$). Therefore, P_i sent this value to all other parties in the second round. Now, a party P_j will output x^j if this is the only value that it saw during the execution. However, as we have just seen, P_j definitely saw x' in the second round. Thus, P_j will only output x^j if $x^j = x'$. On the other hand, if P_j does not output x^j, then it outputs $\perp$.
2. *Validity:* If P_1 is honest, then all parties receive x in the first round. Therefore, they will only output x or $\perp$.

This completes the proof.

2.1 Strengthening Broadcast with Abort

A natural question to ask is whether or not we can strengthen Definition 1 in one of the following two ways (and still obtain a protocol for $t \geq n/3$):

1. *Strengthen the agreement requirement:* If an honest party outputs a value x', then all honest parties output x'. (On the other hand, the validity requirement remains unchanged.)
2. *Strengthen the validity requirement:* If P_1 is honest, then all honest parties output x. (On the other hand, the agreement requirement remains unchanged.)

It is easy to see that the above strengthening of the agreement requirement results in the definition of weak Byzantine Generals. (The validity and non-triviality requirements combined together are equivalent to the validity requirement of weak Byzantine Generals.) Therefore, there exists no *deterministic* protocol for the case of $t \geq n/3$. Regarding the strengthening of the validity requirement, the resulting definition implies a problem known as "Crusader Agreement". This was shown to be unachievable for any $t \geq n/3$ by Dolev in [9]. We therefore conclude that the "broadcast with abort" requirements cannot be strengthened in either of the above two ways (for deterministic protocols), without incurring a $t < n/3$ lower bound.

3 Secure Computation with Abort and No Fairness

In this section, we show that any protocol for secure computation (with unanimous abort and any level of fairness) that uses a broadcast channel, can be "compiled" into a protocol for the point-to-point network that achieves secure computation with abort and no fairness. Furthermore, the fault tolerance of the compiled protocol is the same as the original one. Actually, we assume that the protocol is such that all parties terminate in the same round. We say that such a protocol has *simultaneous termination.* Without loss of generality, we also assume that all parties generate their output in the last round. The result of this section is formally stated in the following theorem:

Theorem 3. *There exists a* (*polynomial-time*) *protocol compiler that takes any protocol Π* (*with simultaneous termination*) *for the* broadcast model, *and outputs a protocol Π' for the* point-to-point model *such that the following holds: If Π is a protocol for information-theoretically* (*resp., computationally*) *secure computation with unanimous abort and any level of fairness, then Π' is a protocol for information-theoretically* (*resp., computationally*) *secure computation with abort and no fairness. Furthermore, Π' tolerates the same number of corruptions as Π.*

Combining Theorem 3 with known protocols (specifically, [27] and [16][4]), we obtain the following corollaries:

Corollary 2. (information-theoretic security – compilation of [27]): *For any probabilistic polynomial-time n-ary functionality f, there exists a protocol in the point-to-point model, for the information-theoretically secure computation of f with abort and no fairness, and tolerating any $t < n/2$ corruptions.*

Corollary 3. (computational security – compilation of [16]): *For any probabilistic polynomial-time n-ary functionality f, there exists a protocol in the point-to-point model, for the computationally secure computation of f with abort and no fairness, and tolerating* any *$t < n$ corruptions.*

[4] Both the [27] and [16] protocols have simultaneous termination

Due to lack of space in this extended abstract, the proof of Theorem 3 is omitted. Nevertheless, we present the motivation and construction of the protocol compiler.

The Protocol Compiler: Intuitively, we construct a protocol for the point-to-point model from a protocol for the broadcast model, by having the parties in the point-to-point network simulate the broadcast channel. When considering "pure" broadcast (i.e., Byzantine Generals), this is not possible for $t \geq n/3$. However, it suffices to simulate the broadcast channel using a protocol for "broadcast with abort". Recall that in such a protocol, either the correct value is delivered to all parties, or some parties output $\perp$. The idea is to halt the computation in the case that any honest party receives $\perp$ from a broadcast execution. The point at which the computation halts dictates which parties (if any) receive output. The key point is that if no honest party receives $\perp$, then the broadcast with abort protocol perfectly simulates a broadcast channel. Therefore, the result is that the original protocol (for the broadcast channel) is simulated perfectly until the point that it may prematurely halt.

Components of the compiler:

1. Broadcast with abort executions: Each broadcast of the original protocol (using the assumed broadcast channel) is replaced with an execution of the broadcast with abort protocol.
2. Blank rounds: Following each broadcast with abort execution, a blank round is added in which no protocol messages are sent. Rather, these blank rounds are used by parties to notify each other that they have received $\perp$. Specifically, if a party receives $\perp$ in a broadcast with abort execution, then it sends $\perp$ to all parties in the blank round that immediately follows. Likewise, if a party receives $\perp$ in a blank round, then it sends $\perp$ to all parties in the next blank round (it also does not participate in the next broadcast).

Thus each round of the original protocol is transformed into 3 rounds in the compiled protocol (2 rounds for broadcast with abort and an additional blank round). We now proceed to formally define the protocol compiler:

Construction 2 (protocol compiler): *Given a protocol Π, the compiler produces a protocol Π'. The specification of protocol Π' is as follows:*

- *The parties use* broadcast with abort *in order to emulate each broadcast message of protocol Π. Each round of Π is expanded into* 3 *rounds in Π':* broadcast with abort *is run in the first* 2 *rounds, and the third round is a* blank *round. Point-to-point messages of Π are sent unmodified in Π'. The parties emulate Π according to the following instructions:*
  1. Broadcasting messages: *Let P_i be a party who is supposed to broadcast a message m in the j^{th} round of Π. Then, in the j^{th} broadcast simulation of Π' (i.e., in rounds $3j$ and $3j+1$ of Π'), all parties run an execution of* broadcast with abort *in which P_i plays the dealer role and sends m.*

2. Sending point-to-point messages: *Any message that P_i is supposed to send to P_j over the point-to-point network in the j^{th} round of Π is sent by P_i to P_j over the point-to-point network in round $3j$ of Π'.*
3. Receiving messages: *For each message that party P_i is supposed to receive from a broadcast in Π, party P_i participates in an execution of* broadcast with abort *as a receiver. If its output from this execution is a message m, then it appends m to its view (to be used for determining its later steps according to Π).*
   If it receives $\perp$ from this execution, then it sends $\perp$ to all parties in the next round (i.e., in the blank round following the execution of broadcast with abort*), and halts immediately.*
4. Blank rounds: *If a party P_i receives $\perp$ in a blank round, then it sends $\perp$ to all parties in the next blank round and halts. In the 2 rounds preceding the next blank round, Party P_i does* not *send any point-to-point messages or messages belonging to a broadcast execution. (We note that if this blank round is the last round of the execution, then P_i simply halts.)*
5. Output: *If a party P_i received $\perp$ at any point in the execution (in an execution of* broadcast with abort *or in a blank round), then it outputs $\perp$. Otherwise, it outputs the value specified by Π.*

In order to prove the security of the compiled protocol, we show how to transform an ideal-model simulator for the original protocol into an ideal-model simulator for the compiled protocol. Although intuitively this seems straightforward, the actual proof is quite involved. See the full version for details.

4 Secure Computation with Abort and Partial Fairness

In this section we show that for any functionality f, there exists a protocol for the *computationally* secure computation of f with abort and partial fairness, and tolerating any $t < n$ corruptions. (This construction assumes the existence of trapdoor permutations.) Furthermore, for any functionality f, there exists a protocol for the *information-theoretically* secure computation of f with abort and partial fairness (and without any complexity assumptions), tolerating any $t < n/2$ corruptions.

We begin by motivating why the strategy used to obtain secure computation with abort and no fairness is not enough here. The problem lies in the fact that due to the use of a "broadcast with abort" protocol (and not a real broadcast channel), the adversary can disrupt communication between honest parties. That is, of course, unless this communication need not be sent over the broadcast channel. Now, in the definition of security with abort and partial fairness, once an honest P_1 receives its output, it must be able to give this output to all honest parties. That is, the adversary must not be allowed to disrupt the communication, following the time that an honest P_1 receives its output. This means that using a "broadcast with abort" protocol in the final stage where the remaining parties receive their outputs is problematic. We solve this problem here by having the parties compute a different functionality. This functionality is such that when P_1 gets its output, it can supply all the other parties with their output directly and without broadcast. On the other hand, P_1 itself should learn nothing of the other parties' outputs. As a first attempt, consider what happens if the parties compute the following instead of the original functionality f:

First attempt:

> ***Inputs:*** $\overline{x} = (x_1, \ldots, x_n)$
> ***Outputs:***
>
> - Party P_1 receives its own output $f_1(\overline{x})$. In addition, for every $i > 1$, it receives $c_i = f_i(\overline{x}) \oplus r_i$ for a uniformly distributed r_i.
> - For every $i > 1$, party P_i receives the string r_i.

That is, for each $i > 1$, party P_i receives a random pad r_i and P_1 receives an encryption of P_i's output $f_i(\overline{x})$ with that random pad. Now, assume that the parties use a protocol that is secure with abort and *no* fairness in order to compute this new functionality. Then, there are two possible outcomes to such a protocol execution: either all parties receive their prescribed output, or at least one honest party receives $\perp$. In the case that at least one honest party receives $\perp$, this party can notify P_1 who can then immediately halt. The result is that no parties, including the corrupted ones, receive output (if P_1 does not send the c_i values, then the parties only obtain r_i which contains no information on $f_i(\overline{x})$). In contrast, if all parties received their prescribed output, then party P_1 can send each party P_i its encryption c_i, allowing it to reconstruct its output $f_i(\overline{x})$. The key point is that the adversary is unable to prevent P_1 from sending these c_i values and no broadcast is needed in this last step. Of course, if P_1 is corrupted, then it will learn all the corrupted parties' outputs first. However, under the definition of partial fairness, this is allowed.

The flaw in the above strategy arises in the case that P_1 is corrupted. Specifically, a corrupted P_1 can send the honest parties modified values, causing them to conclude with incorrect outputs. This is in contradiction to what is required of all secure protocols. Therefore, we modify the functionality that is computed so that a corrupted P_1 is unable to cheat. In particular, the aim is to prevent the adversary from modifying $c_i = f_i(\overline{x}) \oplus r_i$ without P_i detecting this modification. If the adversary can be restrained in this way, then it can choose not to deliver an output; however, any output delivered is guaranteed to be correct. The above-described aim can be achieved using standard (information-theoretic) authentication techniques, based on pairwise independent hash functions. That is, let $\mathcal{H}$ be a family of pairwise independent hash functions $h : \{0,1\}^k \rightarrow \{0,1\}^k$. Then, the functionality that the parties compute is as follows:

Functionality F:

> ***Inputs:*** $\overline{x} = (x_1, \ldots, x_n)$
> ***Outputs:***

- Party P_1 receives its own output $f_1(\overline{x})$. In addition, for every $i > 1$, it receives $c_i = f_i(\overline{x}) \oplus r_i$ for a uniformly distributed r_i, and $a_i = h_i(c_i)$ for $h_i \in_R \mathcal{H}$.
- For every $i > 1$, party P_i receives the string r_i and the description of the hash function h_i.

Notice that as in the first attempt, P_1 learns nothing of the output of any honest P_i (since $f_i(\overline{x})$ is encrypted with a random pad). Furthermore, if P_1 attempts to modify c_i to c'_i in any way, then the probability that it will generate the correct authentication value $a'_i = h_i(c'_i)$ is at most 2^{-k} (by the pairwise independent properties of h_i). Thus, the only thing a corrupt P_1 can do is refuse to deliver the output. Using the above construction, we obtain the following theorem:

Theorem 4. *For any probabilistic polynomial-time n-ary functionality f, there exists a protocol in the point-to-point model for the computationally secure computation of f* with abort and partial fairness, *and tolerating any $t < n$ corruptions. Furthermore, there exists a protocol in the point-to-point model for the information-theoretically secure computation of f* with abort and partial fairness, *and tolerating any $t < n/2$ corruptions.*

Acknowledgements. We would like to thank Oded Goldreich for many helpful discussions.

References

1. D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.
2. D. Beaver and S. Goldwasser. Multiparty Computation with Fault Majority. In *CRYPTO'89*, Springer-Verlag (LNCS 435), 1989.
3. D. Beaver, S. Micali and P. Rogaway. The Round Complexity of Secure Protocols. In *22nd STOC*, pages 503–513, 1990.
4. M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *20th STOC*, pages 1–10, 1988.
5. R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, Vol. 13 (1), pages 143–202, 2000.
6. R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO*, 2001.
7. R. Canetti and H. Krawczyk. Universally Composable Notions of Key-Exchange and Secure Channels. In *EUROCRYPT*, 2002.
8. D. Chaum, C. Crepeau and I. Damgard. Multi-party Unconditionally Secure Protocols. In *20th STOC*, pages 11–19, 1988.
9. D. Dolev. The Byzantine Generals Strike Again. *Journal of Algorithms,* 3(1):14–30, 1982.
10. P. Feldman and S. Micali. An Optimal Algorithm for Synchronous Byzantine Agreement. *SIAM. J. Computing*, 26(2):873–933, 1997.

11. M. Fischer, N. Lynch, and M. Merritt. Easy Impossibility Proofs for Distributed Consensus Problems. *Distributed Computing*, 1(1):26–39, 1986.
12. M. Fitzi, N. Gisin, U. Maurer and O. Von Rotz. Unconditional Byzantine Agreement and Multi-Party Computation Secure Against Dishonest Minorities from Scratch. To appear in *Eurocrypt 2002.*
13. M. Fitzi, D. Gottesman, M. Hirt, T. Holenstein and A. Smith. Byzantine Agreement Secure Against Faulty Majorities From Scratch. To appear in *PODC,* 2002.
14. Z. Galil, S. Haber and M. Yung. Cryptographic Computation: Secure Fault Tolerant Protocols and the Public Key Model. In *CRYPTO 1987.*
15. O. Goldreich. *Secure Multi-Party Computation.* Manuscript. Preliminary version, 1998. Available from `http://www.wisdom.weizmann.ac.il/∼oded/pp.html`.
16. O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In *19th STOC,* pages 218–229, 1987. For details see [15].
17. S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90*, Springer-Verlag (LNCS 537), pages 77–93, 1990.
18. S. Goldwasser and Y. Lindell. Secure Computation Without Agreement (full version of this paper). IACR Cryptology ePrint Archive, Report 2002/040, `http://eprint.iacr.org`
19. J.Kilian, Founding Cryptograph on Oblivious Transfer. In *20th STOC*, pages 20–31, 1988.
20. J. Kilian, A general completeness theorem for two-party games. In Proc. 23rd Annual ACM Symposium on the Theory of Computing, pp. 553–560, New Orleans, Louisiana, 6-8 May 1991.
21. L. Lamport. The weak byzantine generals problem. In *JACM*, Vol. 30, pages 668–676, 1983.
22. L. Lamport, R. Shostack, and M. Pease. The Byzantine generals problem. *ACM Trans. Prog. Lang. and Systems*, 4(3):382–401, 1982.
23. Y. Lindell, A. Lysyanskaya and T. Rabin. On the Composition of Authenticated Byzantine Agreement. In *34th STOC*, 2002.
24. S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.
25. M. Pease, R. Shostak and L. Lamport. Reaching agreement in the presence of faults. In *JACM*, Vol. 27, pages 228–234, 1980.
26. B. Pfitzmann and M. Waidner. Information-Theoretic Pseudosignatures and Byzantine Agreement for $t >= n/3$. Technical Report RZ 2882 (#90830), IBM Research, 1996.
27. T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *21st STOC*, pages 73–85, 1989.
28. A. Yao. How to Generate and Exchange Secrets. In *27th FOCS*, pages 162–167, 1986.

The Lord of the Rings: Efficient Maintenance of Views at Data Warehouses*

D. Agrawal[1], A. El Abbadi[1], A. Mostéfaoui[2], M. Raynal[2], and M. Roy[2]

[1] Department of Computer Science, University of California, Santa Barbara, CA 93111, USA
[2] IRISA, Université de Rennes I
Campus de Beaulieu, 35042 Rennes, France

Abstract. Data warehouses have become extremely important to support online analytical processing (OLAP) queries in databases. Since the data view that is obtained at a data warehouse is derived from multiple data sources that are continuously updated, keeping a data warehouse up-to-date becomes a crucial problem. An approach referred to as the *incremental view maintenance* is widely used. Unfortunately, a precise and formal definition of view maintenance (which can actually be seen as a distributed computation problem) does not exist. This paper develops a formal model for maintaining views at data warehouses in a distributed asynchronous system. We start by formulating the view maintenance problem in terms of abstract update and data integration operations and state the notions of correctness associated with data warehouse views. We then present a basic protocol and establish its proof of correctness. Finally, we present an efficient version of the proposed protocol by incorporating several optimizations. So, this paper is mainly concerned with basic principles of distributed computing and their use to solve database related problems.

1 Introduction

Context of the study: As the capability for storing data increases, there is a greater need for developing analysis tools that provide aggregation and summarization capabilities. In the context of databases, a new class of query processing referred to as OLAP (*OnLine Analytical Processing*) has emerged for aggregating and summarizing vast amount of data. A typical example of an OLAP query is to identify total sales for a product type in a geographic region over a specific period of time. As this example illustrates, since OLAP queries need to scan the data for aggregation, executing them on traditional DBMSs will adversely impact the OLTP (*OnLine Transaction Processing*) workload, which typically consists of short update transactions. Furthermore, enterprise wide data is typically stored on multiple database repositories. Such geographical distribution leads to expensive distributed processing for OLAP queries which demand interactive response

* This work was supported in part by an NSF/INRIA grant INT−009 5527.

times. Unlike the OLTP workloads where up-to-date data is mandatory, OLAP queries typically involve historical data which need not be absolutely up-to-date. This property of OLAP queries has resulted in the emergence of the notion of *data warehouses.*

Data warehouses exploit the lack of *strict currency* requirement of OLAP queries by creating a redundant *view* of the data that is derived from the data repositories or multiple DBMSs comprising an enterprise. In the context of relational DBMSs, this view is specified in terms of an SPJ-expression (select-project-join) over the multiple relations contained in the data repositories. Once the view is created, a critical issue is how to keep the view both consistent as well as approximately current with respect to the changes that occur in the data sources from which the view is derived. One possibility is to re-compute the entire view periodically, e.g., daily, weekly, or monthly. However, such re-computation is unnecessary and wasteful [8]. Instead the view can be maintained incrementally. That is, a view is created initially. Subsequently, whenever changes occur at the data sources, a view maintenance algorithm is initiated to determine the incremental changes (both augmentation as well as removal) to the base view. Numerous protocols have been proposed to find efficient solution to the incremental maintenance problem (e.g., [2,6,9,14,18]). However, most of these approaches are database centric and fail to clearly formalize the problem resulting in subtle inefficiencies as well as ad-hoc restrictions on the system.

Content of the paper: The aim of this paper is to use basic principles of distributed computing to address and solve the view maintenance problem. In that sense, the paper lies at a "borderline" where distributed computing and databases cross. So, since view maintenance involves multiple data sources in an asynchronous environment, we formulate this problem as a distributed computation problem using notions from distributed algorithms and distributed computing. Our goal is to obtain a formal understanding of the system components of a data warehouse architecture much in the same way as concurrency control theory does that for database management systems.

To attain this goal, the paper starts by providing a precise definition of the notion of views in terms of abstract operations with well-defined properties, stating the safety and liveness requirements that provide an abstract and well-defined specification of the problem. Then, the paper introduces an algorithm along with its proof of correctness. Note that very few attempts have been made in the data warehouse literature to define and formally prove the correctness of view maintenance algorithms in asynchronous environments. Furthermore, most data warehousing system architectures require the FIFO property for communication channels between the warehouse and the data sources. The proposed algorithm imposes no such restriction. It is simple, powerful, proved correct and can be generalized to address multi-term view. Hence, the results presented in the paper are (1) a precise (and abstract) definition of the data warehouse view maintenance problem, and (2) a new simple and efficient view maintenance protocol with its proof.

Roadmap: The paper is made up of six sections. Section 2 introduces the underlying data/query model. Section 3 provides a specification of the data view management problem. Then, Section 4 presents and proves correct a data view management protocol. Section 5 presents the case of multi-term views. Finally, Section 6 provides a few concluding remarks.

2 The Model

The system model we consider is based on the data warehouse architecture [2, 17,18]. The system consists of two types of components: the data sources and the data warehouse. We assume that there are n distinct data sources, which can be modeled as data objects denoted $x_1, \ldots, x_n$. These data objects are updated independently of each other as a result of transactions. Note that in this paper we consider transactions that are restricted to a single data source. In [17], Zhuge *et al.* first develop this model and then generalize it to settings where transactions access multiple data sources on different sites. Our future work will expand our approach to include this generalized model. From a database perspective, x_i can be correlated with a relation in a DBMS. The data warehouse can be modeled as a function F whose current value depends on the current state of all data objects x_i.

Data Sources: A data object x_i can be updated by transactions issued by its clients. There are two types of update operations, denoted $\oplus$ and $\ominus$. These operations are such that:

– $\oplus$ and $\ominus$ are inverse of each other: $\oplus^{-1} \equiv \ominus$ and $\ominus^{-1} \equiv \oplus$ (this means that $(a \oplus b = c) \Leftrightarrow (a = c \ominus b)$. Let $\perp$ be the value such that $\forall a : \ (a \oplus \perp = a \ominus \perp = a)$.
– $\oplus$ is associative (this means that $((a \oplus b) \oplus c) \ = \ (a \oplus (b \oplus c)))$.

In database terminology, the $\oplus$ operation can be viewed as an insertion of a tuple in a relation whereas $\ominus$ can be viewed as a deletion of a tuple from the relation. A data object x_i is accessed by one operation at a time. Let $t = 1, 2, \ldots$ represents the time progression. The local history h_i of x_i is the sequence of values taken by x_i at different time instants. It is denoted $(x_i^{[t]})_{t=1,2,\ldots}$ Let us note that if x_i is not updated between $t1$ and $t2$, we have $x_i^{[t1]} = x_i^{[t1+1]} = \cdots = x_i^{[t2]}$1. It is assumed that $\forall i : \ \forall t : \ x_i^{[t]} \ /\!\!=\perp$. Note that x_i is sometimes used to denote the data variable, and sometimes to denote its value. This is done according to the context and when there is no ambiguity.

Data Warehouse: The data warehouse (DWH) provides its clients (when they query it) with the value of a function F defined on the data objects $x_1, \ldots, x_n$. It is assumed that DWH processes one query at a time. This means that the

1 This is sometimes called *stuttering*. The important point is that h_i includes all the updates of x_i.

execution of DWH can be modeled as a sequence of values taken by F, namely the sequence of values returned by the queries. Let us now define the type of the functions F that will be supported by DWH. Let $\otimes$ be an operation that is:

– associative $(a \otimes (b \otimes c) = (a \otimes b) \otimes c)$,
– commutative $(a \otimes b = b \otimes a)$, and
– distributive over $\oplus$ and $\ominus$ $(a \otimes (b \; op \; c) = (a \otimes b) \; op \; (a \otimes c))$ where op is either $\oplus$ or $\ominus$. Let us note that we consequently have $a \otimes \perp = \perp$[2].

In relational database terminology, the $\otimes$ can be viewed as a join operation among multiple relations. At an abstract level, the join operation corresponds to a cross-product of tuples in two relations. However, it is often used with a conditional operator (e.g., equality) so that the number of tuples in the output is limited. Let a *term* be the product $(\otimes)$ of a subset of the data objects, e.g., $x_2 \otimes x_5 \otimes x_6$ is a term. The functions F we consider are sums $(\oplus)$ of such terms. Here is an example of such a function on six data objects $F(x_1, \ldots, x_6) = (x_2 \otimes x_5 \otimes x_6) \oplus (x_1 \otimes x_2 \otimes x_4 \otimes x_5) \oplus (x_3 \otimes x_6)$. Typically, the data warehouse function F involves a single term which is a join operation on n relations.

3 The View Management Problem

The *View Management* problem is concerned with the queries issued by the clients of DWH. Let a *view* be the result returned by a query. Intuitively, the views have to be consistent with respect to the data values, and, as time progresses, they have to incorporate the updates that have been applied to the data. More formally, we express the View Management problem as a set of three properties we name Validity, Order Consistency and Up-to-dateness. The first two are safety properties, while the third is a liveness property. They are defined as follows.

– Validity. A view f obtained by a query is such that $f = F(x_1^{[t_1]}, \ldots, x_n^{[t_n]})$, where $\forall i \in [1..n]$, $t_i \in [1, 2, \ldots)$ (the value $x_i^{[t_i]}$ appears in the history of the data object x_i).
Validity states that a query always returns a meaningful value.
– Order Consistency. Let q_1 (resp. q_2) be a query that returns the value $f1 = F(x_1^{[t1_1]}, \ldots, x_n^{[t1_n]})$ (resp. the value $f2 = F(x_1^{[t2_1]}, \ldots, x_n^{[t2_n]})$). If $f1$ appears before $f2$ in the history of DWH, then $\forall i \in [1..n]$, $t1_i \leq t2_i$.
Order consistency means that the values returned by a sequence of queries are consistent with respect the local history of each data object x_i.
– Up-to-dateness. Let us assume an infinite sequence of queries. $\forall i \in [1..n]$, $\forall t \in [1, 2, \ldots)$, there is a query that returns $f = F(\ldots, x_i^{[t']}, \ldots)$ with $t' \geq t$.
Up-to-dateness states that, provided there are enough queries, any update of a data object x_i will be used to compute a view, or will be overwritten by a more recent update.

[2] Let us remark that, due to the properties we assume, the operations $\oplus$ and $\otimes$ define a *ring* mathematical structure.

Specifying a problem as a set of abstract properties is important for several reasons. First, it does not attach the problem to a specific context or a specific set of mechanisms that allow to solve it. Then, it allows the protocol designer to provide a formal proof of its solution: showing that a particular protocol solves the view maintenance problem requires to establish that any of its execution satisfies the Validity, Order Consistency and Up-to-dateness properties.

The updates at the data sources can be handled at the data warehouse in different ways. Depending on how the updates are incorporated into the view at the data warehouse, different notions of consistency of the view have been identified in [11,18]. *Convergence* where the updates are eventually incorporated into the materialized view. *Strong consistency* where the order of state transformations of the view at the data warehouse corresponds to the order of the state transformations at the data sources. *Complete consistency* where every state of the data sources is reflected as a distinct state at the data warehouse, and the ordering constraints among the state transformations at the data sources are preserved at the data warehouse. These notions are different from the ones we have proposed. Our proposed notions are based on when the state of the warehouse is queried. In contrast, the database notions are based on the state of the data warehouse. We feel that the former is more appropriate since it imposes weaker constraints on the maintenance algorithm[3].

4 The Case of a Simple View (Single Term)

This section presents a protocol that solves the view maintenance problem when the function F is a single term. So, without loss of generality, this section considers that $F(x_1, \ldots, x_n) = x_1 \otimes \ldots \otimes x_n$. (The case where F includes several terms is considered in [3].)

4.1 Preliminaries

We consider a distributed system in which each data object x_i is located on a distinct node. So, from now on, x_i is used to denote both a data object and the node where it is located. Moreover, the DWH entity is assumed to be located on another node. The nodes communicate through reliable (not necessarily FIFO) channels. The nodes are assumed to be reliable. We assume an asynchronous system, i.e., there is no bound on processing time and message transfer delays.

There are two extreme solutions. One is to have a DWH node that is "memoryless": each time it is queried, it asks the data sources and computes the value of F according to the values it gets. The other is to maintain a copy of every data x_i at the DWH node. These solutions are highly inefficient, in terms of communication (former solution) or storage (latter solution). That is why an "*incremental*" data view computation has been proposed by several authors [2,6,17,8,14,18,11,

[3] In the presence of an infinite number of queries, it is possible to show that the proposed specification implies strong consistency.

15]. In general, these approaches maintain at the DWH node the last computed value of F, and only compute the corresponding Δ_F when a data object x_i is updated to a new value $x_i \oplus \delta_i$. More precisely, let $F_0 = F(x_1, \ldots, x_i, \ldots, x_n)$. The idea is for DWH to only compute (with the help of the data nodes) the value Δ_F such that

$$F(x_1, \ldots, x_i \oplus \delta_i, \ldots, x_n) = F_0 \oplus \Delta_F.$$

The main problem that needs to be addressed concerns the view maintenance in the presence of concurrent updates to different data objects. If two data objects, e.g., x_i and x_j, are concurrently updated to $x_i \oplus \delta_i$ and $x_j \oplus \delta_j$, the values Δ_F that are computed must ensure that the values obtained by the queries at DWH are always valid, order consistent and up-to-date.

4.2 The Basic Protocol

This section introduces a *basic* (abstract) protocol solving the view maintenance problem. The protocol is abstract [10] in the sense it uses a token that perpetually moves on a directed ring made up of the data nodes. The ring assumption and the perpetual motion of the token are only used to provide a simple presentation of the protocol principles. So, after having presented and proved the protocol expressed in such an abstract way, the simplifying assumptions (ring and perpetual motion of the token) are eliminated to provide a (concrete) more efficient protocol (Section 4.4) that does not require data nodes to know each other, and that is quiescent (i.e., there is no activity when there are neither data update nor queries).

So, let us assume that the data nodes $x_1, \ldots, x_n$ define a ring. For any node x_i, the function next_data gives the data node that follows x_i on the ring.

The "no concurrent updates" case: Let us consider the case where a single data object x_i is updated to $x_i \oplus \delta_i$. As the value $F(x_1, \ldots, x_i, \ldots, x_n)$ is kept by DWH, the aim is to compute Δ_F such that $F(x_1, \ldots, x_i \oplus \delta_i, \ldots, x_n) = F(x_1, \ldots, x_i, \ldots, x_n) \oplus \Delta_F$, and to supply the DWH with it.

A simple idea is the following[4]. Let the token start from x_i and carry the value δ_i. The token goes to x_j=next_data(x_i). When it receives the token, the node x_j computes $x_j \otimes \delta_i$, and sends the token with this new value to x_k=next_data(x_j), and so on. When, the token returns to x_i, it carries the value $x_1 \otimes \ldots \otimes x_{i-1} \otimes \delta_i \otimes x_{i+1} \otimes x_n$, which (as shown by the proof) is Δ_F. So, when it receives the token, the node x_i has only to send the token value to DWH.

The "concurrent updates" case: Allowing concurrent computations of the Δ_F increments corresponding to concurrent updates (e.g., $x_i + \delta_i$ and $x_j + \delta_j$) poses difficult problems. In order to compute the correct Δ_F, each of the following terms $x_i \otimes \delta_j$, $x_j \otimes \delta_i$, and $\delta_i \otimes \delta_j$, must be taken into account exactly once. The concurrency of the computations induced by the updates can produce a

[4] This idea has been used in several protocols, with an implementation that is not ring/token-based.

Δ_F including twice the term $\delta_i \otimes \delta_j$ (the situation is worse if there are more concurrent updates). This is called an *error term*, and a main challenge for the protocols that allow multiple Δ_F computations due to concurrent updates is to appropriately manage the error terms.

Here, we solve the problem posed by concurrent updates with a pipelining technique. More precisely, the token is an array $token[1..n]$ with one entry per data. The entry $token[i]$ plays the role of the simple token of the "no concurrent updates" case as far as x_i updates are concerned. As we will see, this is particularly efficient.

The previous idea is implemented as follows. The token (initially equal to $[\perp, \dots, \perp]$) and a sequence number generator sn (initialized to 0) are initially placed on a data node, and then move perpetually on the ring. The sn variable is used to order the successive values of Δ_F that are computed; they are sent by the data nodes to DWH and will be denoted $\Delta_F^1, \Delta_F^2, \dots$ Initially DWH keeps the value of the initial view, namely $f_0 = F(v_1, \dots, v_n)$ (where v_i is the initial value of x_i). Then, it proceeds incrementally: when it receives Δ_F^x, it computes $f_x = f_{x-1} \oplus \Delta_F^x$ (the sequence numbers allow it to consider the received Δ_F increments in the correct order).

A single term protocol: The protocol is described in Figure 1. It is made up of three parts. Two describe the behavior of a node x_i: what it does when it receives the pair $(token, sn)$, and what it does when x_i is updated. The third part describes the behavior of DWH when it receives a Δ_F value. The lines 14-17 correspond to the previous discussion on the incremental behavior of DWH. Its local variable $next\_sn$ (initialized to 0) allows it to correctly order the Δ_F increments.

In addition to x_i, the manager of x_i maintains a local variable Δ_i (initialized to $\perp$). This variable is used to record updates of x_i between two consecutive visits of the token. More precisely, each time the token leaves the x_i node (line 10), Δ_i is reset to $\perp$ (line 9). Then, Δ_i aggregates the updates of x_i (line 13) until the next visit of the token. (When we consider an update of x_i (lines 11-13), we only consider the case of the $\oplus$ operation, as the $\ominus$ operation can be expressed from $\oplus$ as it is its inverse.)

The behavior of the node x_i when it receives the pair $(token, sn)$ can be decomposed into two parts. In the first part (lines 2-5), the node provides DWH with the next Δ_F value, if necessary. After a full revolution of the token from x_i to itself, the $token[i]$ entry contains the increment Δ_F corresponding to the previous Δ_i multiplied by the value of each other x_j when the token visited it. If Δ_i was null (a $\perp$ value), then Δ_F is null and no message is sent to DWH. Basically, this part (together with the perpetual motion of the token, line 10) ensures the Up-to-dateness property (liveness). The second part (lines 6-10) concerns the token management, and basically ensures the Validity and Order Consistency properties (safety). In this part, the x_i node updates each token entry: (1) $token[i]$ at line 6 which corresponds to the new updates, and (2) $token[j]$, $j \neq i$, at lines 7-8 which corresponds to the effects of x_i on updates of other nodes. Let us consider the case where a single data object x_k is updated (hence $\Delta_k \neq \perp$, while

```
(1) when $token\_sn(token, sn)$ is received by $x_i$:
(2)     let $\Delta_F = token[i]$;
(3)     if $(\Delta_F \neq \bot)$ then $sn \leftarrow sn + 1$;
(4)                                send $incr\ (\Delta_F, sn)$ to DWH
(5)     endif;
(6)     $token[i] \leftarrow \Delta_i$;
(7)     $\forall j \in [1..n], j \neq i$:
(8)             do $token[j] \leftarrow \big(token[j] \otimes (x_i \ominus \Delta_i)\big)$ enddo;
(9)     $\Delta_i \leftarrow \bot$;
(10)    send $token\_sn\ (token, sn)$ to next_data

(11) when $update\ (\delta_i)$ is received by $x_i$:
(12)    $x_i \leftarrow x_i \oplus \delta_i$;
(13)    $\Delta_i \leftarrow \Delta_i \oplus \delta_i$

(14) when $incr\ (\Delta_F, sn)$ is received by DWH:
(15)    wait $(next\_sn = sn)$;
(16)    $f \leftarrow f \oplus \Delta_F$;
(17)    $next\_sn \leftarrow next\_sn + 1$
```

Fig. 1. An Basic Single Term View Maintenance Protocol

$\forall j \neq k:\ \Delta_j = \bot$). Then, starting from x_k, we have $\forall j \neq k:\ token[j] = \bot$, and $token[k] = \Delta_k$. When the token moves along the ring, we have: $\forall j \neq k:\ token[j]$ remains equal to $\bot$, while $token[k]$ is updated at line 8 by each visited node as follows $token[k] \leftarrow \big(token[k] \otimes x_i\big)$. Consequently, when the token returns to the node x_k, we have $token[k] = x_1 \otimes \cdots \otimes x_{k-1} \otimes \Delta_k \otimes x_{k+1} \otimes x_n$, and x_k sends this value to DWH as the next Δ_F.

Remark. The protocol aggregates all the updates $(\delta_i 1, \delta_i 2, \ldots)$ on x_i that occur between two consecutive visits of the token, and considers them as a single update, namely Δ_i. If we do not want to allow the gathering of consecutive updates (δ_i) into a "big" update (Δ_i), each data node x_i can use a list recording its successive δ_i updates, and define Δ_i at line 13 as the first δ_i of the list not yet considered. *End of remark.*

Latency of the basic protocol: We evaluate here the time latency of the protocol as the time that elapses between an update to the time when its effects are incorporated at DWH. We assume that each message takes one time unit, and that processing takes no time. Let us consider the two following extreme cases.

Case 1: a single data object has been updated. In that case, the token has to arrive at the corresponding node, then the token has to complete a turn of the ring. Moreover, an *incr* message has then to be sent. This means that, in the worst case, $2n$ time units are required.

Case 2: each data object has been updated. It easy to see that in that case $2n$ time units are also required. This means that, in that case, the average cost of an update is 2 time units.

Section 4.4 presents improvements that provide an efficient version of the protocol that reduces the cost to n time units in Case (1), while not increasing it in Case (2).

4.3 Correctness Proof

Theorem 1. *The protocol described in Figure 1 solves the* view maintenance *problem.*

Proof We have to show that the protocol satisfies the Validity, Order Consistency and Up-to-dateness properties described in Section 3. The proof uses the following notation: $\forall i \in [1..n]: \ \forall k \in \mathbf{Z}: x_{i+kn} \equiv x_i$.

Let us define a virtual time notion as follows: the time progress is measured as the number of steps performed by the token. Hence, at $t = 1$ the token is in x_1's possession, at $t = 2$, the token is held by x_2, at t the token goes for the $((t \div n) + 1)th$ time to the site site $x_{(t-1)\mathsf{mod}n+1} \equiv x_t$.

Let us rewrite $(x_i^{[t]})_{t \geq 0}$ as the local history of x_i sampled every n time units (n is the time duration for a complete tour of the ring by the token). This means that the semantics of $x_i^{[t]}$ is as follows: for any $k \in \mathbf{N}$, $x_i^{[i+kn]} = x_i^{[i+kn+1]} = \cdots = x_i^{[i+kn+n-1]}$, meaning that, from an external observer point of view, x_i is seen as constant during a tour, and $x_i^{[i+kn]}$ includes all the updates made on x_i during the kth tour of the ring (these updates are locally kept at the node x_i in Δ_i). We also define for convenience of the proof $x_i^{[t]} = x_i^{[0]}, \forall t < 0$.
The proof basically follows from the following claim:
Claim: At step t,

1. The value computed on the DWH based on all messages for steps up to $t-1$ satisfies: $f^{[t-1]} = (x_1 \otimes x_2 \otimes \cdots \otimes x_n)^{[t-1-n]}$.
2. The *token* value received by the data manager of x_i (with $i = (t-1)\mathsf{mod}n+1$) is the array:

$$\begin{aligned}
\mathit{token}[i] &= (x_i^{[t-n]} \ominus x_i^{[t-2n]}) && \otimes(x_1 \otimes \cdots \otimes x_{i-1} \otimes x_{i+1} \otimes \cdots \otimes x_n)^{[t-n]} \\
\mathit{token}[i+1] &= (x_{i+1}^{[t-n+1]} \ominus x_{i+1}^{[t-2n+1]}) && \otimes(x_1 \otimes \cdots \otimes x_{i-1} \otimes x_{i+2} \otimes \cdots \otimes x_n)^{[t-n+1]} \\
&\cdots \\
\mathit{token}[i-2] &= (x_{i-2}^{[t-2]} \ominus x_{i-2}^{[t-n-2]}) && \otimes(x_{i-1})^{[t-2]} \\
\mathit{token}[i-1] &= (x_{i-1}^{[t-1]} \ominus x_{i-1}^{[t-n-1]}).
\end{aligned}$$

Using this claim, the proposed protocol trivially satisfies Validity, since any query returns an $f^{[t]} = \bigotimes_{1 \leq i \leq n} x_i^{[t]}$ for some t.

Order Consistency is ensured because the DWH always updates its state from $f^{[t]}$ to $f^{[t']}$, with $t' \geq t$. This means that if DWH processes the query q_1 before the query q_2 and if $q1$ returns $f^{[t]} = (\bigotimes_{1 \leq i \leq n} x_i^{[t]})$, then $q2$ will return $f^{[t']} = (\bigotimes_{1 \leq i \leq n} x_i^{[t']})$ such that $t' \geq t$.

For Up-to-dateness, let us observe that, the token turning endlessly on the ring, any update is taken into account in the token during one on its turns on the ring, say during tour k. Then, this modification will be committed on DWH at the latest in the value $f^{[(k+1)*n+1]}$, using the first item of the claim.

Proof of the claim:

• *Initialization.* The token is initialized to $[\bot, ..., \bot]$. Since $\bot$ is the zero of the $\oplus$ operation, during the first tour, the value on DWH remains unchanged. As it is initialized to $\bigotimes_{1\leq i\leq n} x_i^{[0]}$, the first item of the claim holds for $1 \leq t \leq n$. The second item is satisfied for $t = 1$ because there are no changes for x_i before $t = 0$.

• *Induction.* Let us suppose that the two items are satisfied until some $t > 0$, and let us show that they hold for $t + 1$. Let $i = (t - 1) \bmod n + 1$.

1. There are two cases corresponding the test at line 3. If $token[i]$ is equal to $\bot$, it means that p_i had no update to commit the previous time it got the token. Hence x_i was not modified at time $t - n$, and no update is needed on the DWH ; therefore, the first item of the claim holds for $t + 1$.

If $token[i]$ contains a value, the x_i data manager sends the value $\Delta_F = token[i]$ to DWH(line 5) with an incremented sequence number. DWH adds it to its current view of the product at line 16. Due to the sequence number synchronization (line 15), and using the induction hypothesis on the token shape, the new value computed at line 16 is :
$f' = f^{[t-1]} \oplus (x_i^{[t-n]} \ominus x_i^{[t-2n]}) \otimes (x_1 \otimes \cdots \otimes x_{i-1} \otimes x_{i+1} \otimes \cdots \otimes x_n)^{[t-n-1]}$
and, by the induction hypothesis on $f^{[t]}$ we get:
$f' = (x_1 \otimes \cdots \otimes x_n)^{[t-n-1]} \oplus (x_i^{[t-n]} \ominus x_i^{[t-2n]}) \otimes (x_1 \otimes \cdots \otimes x_{i-1} \otimes x_{i+1} \otimes \cdots \otimes x_n)^{[t-n]}$.

But, for any x_j ($j \neq i$) the values of x_j are equal at times $t - n$ and $t - n - 1$ (definition of $x_i^{[t]}$), so the previous expression reduces to:
$f' = ((x_1 \otimes \cdots \otimes x_{i-1} \otimes x_{i+1} \otimes \cdots \otimes x_n)^{[t-n]}) \otimes (x_i^{[t-n-1]} \oplus x_i^{[t-n]} \ominus x_i^{[t-2n]})$.

Since $x_i^{[t-n-1]} = x_i^{[t-2n]}$, we conclude that the new value f' computed by DWH is equal to $f^{[t]}$, which proves the first item of the claim.

2. Then, the data manager multiplies each other term by its old value of x_i (lines 7-8), that is the value that does correspond to dates $t-1$, $t-2$, ..., $t-n$. Hence, the token is now made up of the following values:

$$
\begin{aligned}
token[i+1] &= (x_{i+1}^{[t-n+1]} \ominus x_i^{[t-2n+1]}) \otimes (x_1 \otimes \cdots \otimes x_i \otimes x_{i+2} \otimes \cdots \otimes x_n)^{[t-n+1]} \\
&\cdots \\
token[i-2] &= (x_{i-2}^{[t-2]} \ominus x_{i-2}^{[t-n-2]}) \quad \otimes (x_{i-1} \otimes x_i)^{[t-2]} \\
token[i-1] &= (x_{i-1}^{[t-1]} \ominus x_{i-1}^{[t-n-1]}) \quad \otimes (x_i)^{[t-1]}.
\end{aligned}
$$

Finally, the data manager of x_i updates $token[i]$ to $(x_i^{[t]} \ominus x_i^{[t-n]}) = \Delta_i^{[t]}$ (line 6) before sending the token to x_{t+1}(that is x_{i+1}). Since the time increases one tick at each token move, it follows that x_{t+1} will get a token of the same shape, thus proving the second part of the claim. *End of the proof of the claim.*

$\square_{Theorem}$ 1

4.4 An Efficient Protocol

This section presents a version of the basic protocol in which (1) the data nodes are not required to know each other, and (2) each query/update gives rise to a finite number of control messages (this means that the protocol eventually stops sending message if there is no more query or update[5]).

So the goal is to suppress (1) the ring and (2) the perpetual motion of the token. Issue (1) can easily be realized by structuring the system as a star whose center is the DWH node: each data object x_i sends messages only to (and receives only from) DWH. (Interestingly, this makes useless the sequence numbers used in Figure 1.) Issue (2) deals also with efficiency. There are several possible approaches to address it. One is for a data node x_i to query the DWH node to get the token when x_i has been updated. According to the quality of service desired for the up-to-dateness of F, the data nodes can require the token each time their data is updated, or only after some "quantity" of updates have been done ("data update-driven" refreshing of F). Another approach could be for the DWH node to entail a token motion periodically or each time the value of F is queried by its clients ("DWH-driven" refreshing of F).

Here we describe a protocol (Figure 2) that implements the data-driven update approach. When compared with the basic protocol, the main modifications concern DWH.

Behavior of a data node: (Lines 1-12) When x_i is updated, the data node requests the token by sending a message to DWH (line 10). When it receives the token (that now no longer carries a sequence number generator), the data node x_i updates appropriately the token entries (i.e., as in the basic protocol, see lines 1-4), and sends back the token to DWH (line 5).

Behavior of the DWH *node:* (Lines 13-36) DWH manages the following local variables and uses two functions:
- $pending[1..n]$ is a boolean array, such that $pending[i] = true$ means that x_i has required the token (in order the increment Δ_F due to the update Δ_i be taken into account).
- *holder* contains the identifier of the current token owner ($\in [1..n]$). When no token is running, $holder = -1$.
- succ(i) (line 30) is a function that returns the data node that follows the node x_i on the "ring".
- next_holder(i) (lines 31-36), assumes that x_i was the previous token holder, and delivers the next token holder (according to the position on data nodes on the "ring"), or -1 if the token has not been requested by a data node.

When DWH receives a request for the token from x_i (line 13), it creates a token and sends it back to x_i if there is no token (lines 14-16). Otherwise, it records the fact that x_i has required the token (line 17).

[5] The property for a protocol to eventually stop sending control messages when they are no more request (update/query) is sometimes called *quiescence* [5].

```
(1) when $token\_sn(token)$ is received by $x_i$:
(2)      $token[i] \leftarrow \Delta_i$;
(3)      $\forall j \in [1..n], j \neq i$: do $token[j] \leftarrow \big(token[j] \otimes (x_i \ominus \Delta_i)\big)$ enddo;
(4)      $\Delta_i \leftarrow \perp$;
(5)      send $token\_sn$ $(token)$ to DWH;
(6)      $sent\_request_i \leftarrow false$

(7) when $update$ $(\delta_i)$ is received by $x_i$:
(8)      $x_i \leftarrow x_i \oplus \delta_i$;
(9)      $\Delta_i \leftarrow \Delta_i \oplus \delta_i$
(10)     if $(\neg sent\_request_i)$ then send $request$ $(i)$ to DWH;
(11)                                       $sent\_request_i \leftarrow true$
(12)     endif

(13) when $request$ $(i)$ is received by DWH:
(14)     if $(holder = -1)$ then $holder \leftarrow i$;
(15)                             $token \leftarrow [\perp, \ldots, \perp]$ % create a token %
(16)                             send $token\_sn$ $(token)$ to $x_i$
(17)                        else $pending[i] \leftarrow true$
(18)     endif

(19) when $token\_sn$ $(token)$ is received by DWH from $x_i$:
(20)     let $\Delta_F = token[\mathsf{succ}(i)]$;
(21)     if $(\Delta_F \neq \perp)$ then $f \leftarrow f \oplus \Delta_F$;
(22)                             $token[\mathsf{succ}(i)] \leftarrow \perp$
(23)     endif;
(24)     if $(token \neq [\perp, \ldots, \perp])$ then $holder \leftarrow \mathsf{succ}(i)$
(25)                                       else $holder \leftarrow \mathsf{next\_holder}(i)$
(26)     endif;
(27)     if $(holder \neq -1)$ then $pending[holder] \leftarrow false$;
(28)                             send $token\_sn$ $(token)$ to $x_{holder}$
(29)     endif

(30)     function $\mathsf{succ}(i)$ returns $\big((i \bmod n) + 1\big)$

(31)     function $\mathsf{next\_holder}(i)$:
(32)         $y \leftarrow i$;      % $x_i$ was the previous token holder %
(33)         repeat $y \leftarrow \mathsf{succ}(i)$ until $(y = i \ \vee \ pending[x])$ endrepeat;
(34)         if $(y = i)$ then returns $(-1)$
(35)                      else returns $(y)$
(36)         endif
```

Fig. 2. A Quiescent Single Term View Maintenance Protocol

When DWH receives the token from x_i (line 19), it first incrementally updates f with the appropriate value Δ_F if necessary (lines 20-23), exactly as in the basic protocol. Then, DWH determines the next holder of the token (lines 24-26). If $token \neq [\perp, \ldots, \perp]$, then the token has to complete its current turn of the ring (line 24); otherwise, the token skips to a requesting data node (if there is no requesting data node we have $holder = -1$). Finally, according to the value computed for $holder$, DWH sends the token to x_{holder} if $holder \in [1..n]$, or destroys it if $holder = -1$.

It is easy to see that the protocol is quiescent. The proof that it solves the view maintenance problem is similar to the proof of the basic protocol.

5 The General Case

This section addresses the case where the view F is composed of several terms, i.e., $F(x_1, \ldots, x_n) = \bigoplus_x term_x$ where each $term_x$ is a term. The following view will be used in the following as an example to illustrate the underlying principles of the proposed solution:

$$F(x_1, \ldots, x_5) = (x_1 \otimes x_2 \otimes x_3 \otimes x_5) \oplus (x_1 \otimes x_2 \otimes x_4) \oplus (x_3 \otimes x_4).$$

When the view F includes a single term, as we have seen, the main problem that has to be solved is the management of concurrent updates. Basically, this was a *parallelism management* problem. The basic protocol (and its improvement) described in the previous section addressed this issue with an appropriate pipelining technique implemented by a token moving on a ring.

When the view is composed of several terms, the new problem that appears is the following. Considering the previous view F (the initial value of which is f_0), let us look at the data x_4, and assume it is the only data that is updated, namely to $x_4 \oplus \delta_4$. We get:

$$F(x_1, x_2, x_3, x_4 \oplus \delta_4, x_5) = (x_1 \otimes x_2 \otimes x_3 \otimes x_5) \oplus (x_1 \otimes x_2 \otimes (x_4 \oplus \delta_4)) \oplus (x_3 \otimes (x_4 \oplus \delta_4)),$$

i.e., from an incremental computation point of view:

$$F(x_1, x_2, x_3, x_4 \oplus \delta_4, x_5) = f_0 \oplus (x_1 \otimes x_2 \otimes \delta_4) \oplus (x_3 \otimes \delta_4) = f_0 \oplus \Delta 0_F \oplus \Delta 1_F.$$

This means that, not only a single update of a data entails more than one Δ_F computation (namely, here $\Delta 0_F$ and $\Delta 1_F$), but their results have to be *atomically added* to f_0 (adding them separately would produce an intermediate value of f that does not correspond to a valid view with respect to the given specification). Basically, this is a *synchronization* problem (in some sense, this problem is dual with respect to the parallelism management problem due to the concurrent updates).

The solution we propose is the following. It actually generalizes the basic protocol described in Figure 1.

- First, a ring and a token are associated with each term of the view. Hence, a token-based protocol is associated with each term (if the view is made up of a single term, the protocol simplifies and becomes the basic protocol described in Figure 1).

– If a data object x_i belongs to several rings (i.e., it appears in several terms), its updates Δ_i not yet taken into account in the computation of F, require all the corresponding tokens to be simultaneously present at x_i in order the corresponding Δ_F increments (one for each term to which x_i belongs) be computed. In the previous example, both the token associated with the ring including x_1, x_2 and x_4 (second term) and the token associated with the ring including x_3 and x_4 (third term) have to be simultaneously present on the node x_4 for Δ_4 to be taken into account.
  Moreover, as previously, each token carries a sequence number generator to allow DWH to correctly add the Δ_F increments to the current value f, i.e.:
  - in the correct order for each ring considered separately, and
  - at the same time (i.e., atomically) for the Δ_F increments that come from different rings but are due to the same update Δ_i of a data object x_i.

A protocol that follows these principles can be found in [3].

6 Concluding Remarks

This paper has developed a formal model for maintaining views at data warehouses in a distributed asynchronous system. The view maintenance problem associated with distributed data warehouses has been investigated primarily by the database community. In general, the focus has been directed towards an appropriate storage model for supporting fast OLAP queries as well as rapid integration of updates at the data sources into the data warehouse. There is relatively little emphasis to provide a formal definition of the view maintenance problem and formally develop a correct solution. In this paper, we provided a formulation of the view maintenance problem in terms of abstract update and data integration operations and defined the notions of correctness associated with data warehouse views. We also introduced a basic protocol and established its proof of correctness. Then, we presented an efficient version of the proposed protocol by incorporating several improvements. We also included the principles of the solution to a more general view formulation which currently does not have a similar semantics in database systems. However, this general framework could be particularly promising to integrate semi-structured data sources.

To conclude, we would like to point out that this paper is an outcome of interactions between researchers working in the areas of distributed computing and databases, respectively. We believe that such interactions are highly beneficial to both communities. Interestingly, when looking in the past, we can observe such beneficial influences. We cite two: the area of epidemic communication [7,16] that first appeared in the distributed computing area and has then been successfully applied to databases (e.g., [4]), and the domain of atomic broadcast/multicast and group communications [1,12,13].

References

1. Agrawal D., Alonso G., El Abbadi A. and Stanoi I., Exploiting Atomic Broadcast in Replicated Databases. *Proc. of the International Conference on Parallelism (EUROPAR)*, pp. 496–503, August 1997.
2. Agrawal D., El Abbadi A., Singh A. and Yurek T., Efficient Data View Maintenance Warehouses. *Proc. ACM SIGMOD*, pp. 417–427, 1997.
3. Agrawal A., El Abbadi A., Mostéfaoui A., Raynal R. and Roy M., The Lord of the Rings: Efficient Maintenance of Views at Data Warehouses. *IRISA Research Report #1441*, IRISA, Rennes, France. Available at: `http://www.irisa.fr/bibli/publi/pi/2002/1441/1441.html`.
4. Agrawal D., El Abbadi A. and Steinke R.C., Epidemic Algorithms in Replicated Databases. *Proc. ACM PODS*, pp. 161–172, 1997.
5. Aguilera M.K., Chen W. and Toueg S., On Quiescent Reliable Communication. *SIAM Journal of Computing*, 26(6):2040–2073, 2000.
6. Colby L.S., Griffin T., Libkin L., Mumick I.S. and Trickey H., Algorithms for Deferred View Maintenance. *Proc. ACM SIGMOD*, ACM Press, pp. 469–480, 1996.
7. Demers A. *et al.*, Epidemic Algorithms for Replicated Database Maintenance. *Proc. ACM PODC*, ACM Press, pp. 1–12, 1987.
8. Gupta A. and Mumick I.S., Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):3-18, June 1995.
9. Gupta A., Mumick I.S. and Subramanian V.S., Maintaining Views Incrementally. *Proc. ACM SIGMOD*, pp. 157-166, 1993.
10. Hélary J.-M., Mostéfaoui A. and Raynal M., A General Scheme for Token and Tree-Based Distributed Mutual Exclusion Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 5(11):1185-1196, 1994.
11. Hull R. and Zhou G., A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. *Proc. ACM SIGMOD*, pp. 481-492, 1996.
12. Kemme B. and Alonso G., A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems*, 25(3):333-379, 2000.
13. Powell D. (Guest Editor). Special Issue on Group Communication. *Communications of the ACM*, 39(4):50-97, 1996.
14. Rundensteiner E.A., Koeller A. and Zhang X., Maintaining Data Warehouses over Changing Information Sources. *Communications of the ACM*, 43(6):57-62, 2000.
15. Stanoi I., Agrawal D. and El Abbadi A., Modeling and Maintaining Multi-View Data Warehouses. *Proc. 18th. Int. Conference on Conceptual Modeling*, Paris, France, pp. 161-175, 1999.
16. Wuu G. T. and Bernstein A. J., Efficient Solutions to the Replicated Log and Dictionary Problems. *Proc. ACM PODC*, pp. 233-242, 1984.
17. Zhuge Y., Garcia-Molina H., Hammer J. and Widom J., View Maintenance in a Warehousing Environment. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 316-327, 1995.
18. Zhuge Y., Garcia-Molina H. and Wiener J.L., The Strobe Algorithms for Multi-Source Warehouse Consistency. *Proc. Int. Conference on Parallel and Distributed Information Systems*, 1996.

Condition-Based Protocols for Set Agreement Problems

A. Mostéfaoui[1], S. Rajsbaum[2], M. Raynal[1], and M. Roy[1]

[1] IRISA, Université de Rennes, Campus de Beaulieu, 35042 Rennes Cedex, France
[2] HP Research Lab, One Cambridge Center, Cambridge, MA 02139, USA

Abstract. A condition C is a set of input vectors to a problem. A class of conditions that allow to solve k-set agreement in spite of f crashes in an asynchronous system is identified. A k-set agreement protocol that is always safe is described: it is guaranteed to terminate when the input vector belongs to C and it always decides on at most k different values, even if the input vector does not belong to C. While there are simple solutions when $f < k$, it is known that the k-set agreement problem has no solution when $f \geq k$. Thus, the paper identifies classes of conditions that allow to solve this problem even when $f \geq k$. The paper gives evidence that these are the only conditions that allow to solve set agreement, by proving the wait-free case. Two natural concrete conditions that belong to such a class are described. Finally, a more efficient k-set agreement protocol with only linear complexity (does not use snapshots), for any C that allows to solve consensus, when $k \leq f/(n-f)+1$ is presented. This shows how to trade fault-tolerance for agreement precision using the condition based approach.

1 Introduction

Agreement Problems and the Condition-Based Approach. Modern distributed applications have to run efficiently and reliably in complex environments where network components can fail in various ways, and under a range of network and processor loads. To cope with these difficulties, system designers often tailor their applications to run fast under "normal circumstances," and accept to pay the price of expensive recovery procedures, or even manual intervention, in the relatively rare event of unexpected situations happening. Various definitions of "normal circumstances" are made, in terms of synchrony or reliability assumptions about the network. There are, for example, partially synchronous systems where delays and relative processor speeds are bounded [9,10], failure detectors [7,23] that abstract away from the details of how a processor suspects failures without referring to particular synchrony assumptions, or group communication systems that assume the network eventually stabilizes. Consensus algorithms that use typical failure detectors can take an unbounded amount of time to terminate, but in failure-free executions where communication is timely, there are algorithms that terminate using only two rounds of communication, and this is a lower bound [19].

Such optimistic approaches can be used to deal with (i) problems that would be too inefficient to solve under every possible network behavior, or (ii) problems that are not solvable at all. In the first case, a protocol would solve the problem in question fast under normal circumstances, and in other rare circumstances would take longer to terminate. In the second case, a protocol would solve the problem under normal circumstances, and perhaps not terminate at all in other cases. In this paper we consider both scenarios using the optimistic *condition based* approach introduced in [20] to design distributed algorithms for *set agreement* problems. In the condition based approach, "normal circumstances" is understood with respect to possible input configurations of a system. For example, in a voting system it may be expected that only in rare situations the votes proposed to the system are evenly divided, while most of the time one candidate gets a clear majority of the votes. More generally, a *condition* is a set of input configurations to the system. A protocol tailored for a condition C expects that most of the time the system starts in a configuration that belongs to C.

In this paper, we present condition based solutions to k-set agreement problems. Each processor in the system starts the computation with an input value, and after communicating with the others, has to irrevocably decide on one of the input values, such that the set of decided values contains at most k different elements. Thus, an input configuration for this problem can be described by a vector with one entry per process, such that the i-th entry contains the value proposed by the process p_i. A condition C is a set of input vectors over a set of values $\mathcal{V}$. A simple example of a condition includes every input vector over $\mathcal{V}$. However, there is no set agreement solution for C when $|\mathcal{V}| > k$ (see below). In the other extreme there is the condition C' that includes all input vectors with no more than k different values. If only input vectors belonging to C' are possible, it is easy to solve k-set agreement (see Section 3.4). We would like to identify what exactly is the class of conditions that allow to solve k-set agreement in spite of f crashes in an asynchronous system. Moreover, we are interested in protocols that are always safe: processes should *always* decide on at most k different values, even for inputs vectors that do not belong to the condition. Termination is required only for input vectors belonging to the condition. Notice that under this additional requirement, a protocol where each process always decides its initial value does not solve k-set agreement for C'.

In an asynchronous read/write shared memory system of n processes where at most f can crash, k-set agreement protocols have been designed for $f < k$ [8] (i.e, all input vectors over $\mathcal{V}$ are possible). It has then been shown that the k-set agreement problem cannot be solved in systems in which $f \geq k$ when $|\mathcal{V}| > k$ [6,17,27]. Thus, the degree of achievable agreement in an asynchronous system is related to the number of faults tolerated. In this paper we present condition-based k-set agreement protocols for $f \geq k$. We also identify conditions for which very efficient set agreement protocols exist.

An important particular case of k-set agreement is *consensus,* when $k = 1$. The condition-based approach to the consensus problem was studied in [20] where a generic, f-fault tolerant condition based protocol that solves consensus for any condition in a class denoted $\mathcal{C}_f^{wk}$ was designed. It was proved that there is no such protocol for conditions not in $\mathcal{C}_f^{wk}$. In [21] the structure of the class $\mathcal{C}_f^{wk}$ was studied, and it was defined a hierarchy of classes of conditions, each one of some *degree* d of "easiness"

$$\mathcal{C}_f^{[f]} \subset \cdots \subset \mathcal{C}_f^{[d+1]} \subset \mathcal{C}_f^{[d]} \subset \cdots \subset \mathcal{C}_f^{[1]} \subset \mathcal{C}_f^{[0]} = \mathcal{C}_f^{wk}.$$

A generic consensus protocol was described whose efficiency is related to the degree of the condition.

Results. This paper has three contributions. The first is the definition and study of a class of conditions for each pair f, k, denoted $\mathcal{S}_{f,k}$, that allows k-set agreement to be solved. We describe two particular conditions, $C1_{f,k}$ and $C2_{f,k}$, and show that both belong to $\mathcal{S}_{f,k}$. These natural conditions generalize those of [20] for the consensus problem. The first one allows processes to decide on one of the k largest proposed values; this is achieved by considering only input vectors where the first k largest proposed values are proposed more than f times. In contrast, the second condition allows processes to decide on the most common proposed values; here it is required that the difference between the number of k most common proposed input values and the rest is sufficiently greater than f. We prove that

$$\mathcal{S}_{f,1} \subset \mathcal{S}_{f,2} \subset \cdots \subset \mathcal{S}_{f,f-1} \subset \mathcal{S}_{f,f}.$$

The $\subseteq$ containments are fairly easy to show. Proving they are strict is more involved: we use techniques from algebraic topology [15,16] to prove it by showing that $C1_{f,k+1}$ is not in $\mathcal{S}_{f,k}$.

The second contribution is an f-fault tolerant protocol that, given any condition $C \in \mathcal{S}_{f,k}$, solves k-set agreement for C. The protocol is presented in the shared memory model and for simplicity using atomic snapshots [2]. It is generic in the sense that it can be instantiated to work for any $C \in \mathcal{S}_{f,k}$. We show that if (1) the input vector satisfies C, or (2) less than k processes crash (whether or not the input vector satisfies C), then the protocol terminates. Moreover, safety is never violated: no more than k values are ever decided.

We conjecture that the class $\mathcal{S}_{f,k}$ captures exactly the conditions that allow k-set agreement to be solved. That is, that it is possible to show that if there is a k-set agreement protocol for C, then $C \in \mathcal{S}_{f,k}$. As evidence, we show that this is true for $f = n-1$, for $f = 1$ (and hence $k = 1$) and for $k = 1$ (and $f \geq 1$). The proof of the $f = n-1$ case is a direct application of the impossibility result of [6, 17,27], and we show it using the techniques of [15] (to follow the algebraic style used to show the strict containments above, but it is possible to do it with other direct uses of previous approaches, e.g. [3]). The case of $k = 1$ follows because $\mathcal{S}_{f,1}$ are the solvable conditions for consensus $\mathcal{S}_{f,1} = \mathcal{C}_f^{wk}$, and we know from [20] that consensus can be solved for a condition C only if $C \in \mathcal{C}_f^{wk}$. Intuitively, our conjecture says that if k-set agreement is solvable for C, then it is solvable by a protocol that bases its output decisions only on a single snapshot of the input values (taken after waiting for at least $n - f$ processes to show up).

The third contribution is related to efficiency. For the particular case $k = 1$, the previous k-set protocol reduces to the one in [20], and both solve consensus for $C \in \mathcal{S}_{f,1}$, with an $O(n \log n)$ cost. So, the third contribution of the paper is a k-set agreement protocol that, for any f and when instantiated with a condition $C \in \mathcal{S}_{f,1}$, enjoys the following noteworthy property. Let J be the input vector: (1) If $J \in C$, it solves the consensus problem and its cost is $O(n)$; (2) If $J \notin C$, the number of decided values is $\leq f/(n - f) + 1$ (and its cost is then $O(n)$ or $O(n \ \log n)$ according to the actual value of J). This shows that it is possible to reduce the time complexity of solving consensus under normal circumstances by sacrificing agreement precision in other cases. Let us notice that, interestingly, this k-set agreement protocol always solves consensus when $J \in C$ or $f < n/2$, and then yields a consensus protocol whose cost is always $O(n)$. This improves on the consensus protocol described in [21], that requires $(2n + 1) \ \lceil \log_2(\lceil (f - d)/2 \rceil + 1) \rceil$ shared memory read/write operations per process, for any $C \in \mathcal{C}_f^{[d]}$. In other words, this shows that the hierarchy of [21] collapses for $f < n/2$.

Related Work. The idea of considering restricted set of inputs to a problem is very natural and has appeared in various contexts, like on-line algorithms, adaptive sorting, etc. Agreement problems with a restricted set of inputs vectors were considered in [25,26], where possibility and impossibility results in a shared memory system and a hierarchy of problems that can be solved with up to f failures but not for $(f + 1)$ failures are introduced. More generally, an approach for designing algorithms in situations where there is some information about the typical conditions that are encountered when the respective problem is solved is presented in [4]. In this paper the consensus problem in a synchronous setting is analyzed taking as parameter the difference between the number of 0's and 1's in the input vector.

The foundation underlying the proposed condition-based approach can be formalized using topology (e.g., [14]). Our setting is similar to that of the previous topology papers, except that those consider *decision tasks* where processes have to terminate always, with an output vector satisfying the task specification. In general, the study of f-fault tolerant decision tasks requires higher dimensional topology (except for the case of $f = 1$ which uses only graphs [5]), and leads to undecidable characterizations [12,13] (NP-Hard for $f = 1$ [5]). Our conjecture would imply that the characterization of the k-set agreement conditions is decidable.

A result similar to our $f = n - 1$ characterization has been independently developed by Attiya and Avidor[1]. Their model is slightly different because they assume that input vectors outside of the condition can never occur. The proof of their characterization uses new notions of knowledge to derive combinatorial topology arguments for the upper and the lower bounds. In contrast, the lower bound derived here uses algebraic topology, and the upper bound is obtained using a generic algorithm.

A full version of this paper appears in [22].

2 The Condition-Based Approach

Computation Model. We consider a standard asynchronous shared-memory system with n, $n > 1$, processes where at most f, $0 \leq f < n$, processes may crash. The shared memory consists of single-writer, multi-reader atomic registers. In addition to the shared memory, each process has a local memory. The subindex i is used to denote p_i's local variables.

The shared memory is organized into arrays. The j-th entry of an array $X[1..n]$ can be read by any process p_i with an operation $\mathsf{read}(X[j])$. Only p_i can write to the i-th component, $X[i]$, and it uses the operation $\mathsf{write}(v, X[i])$ for this. The notation $\mathsf{collect}(X)$ is an abbreviation for: $\mathsf{read}(X[j])$ for all j in arbitrary order. Hence, it defines a non-atomic operation that returns an array of values $[a_1, \ldots, a_n]$ such that a_j is the value returned by $\mathsf{read}(X[j])$.

We assume that processes can take atomic snapshots of any of the shared arrays: $\mathsf{snapshot}(X)$ allows a process p_j to atomically read the content of all the registers of the array X. This assumption is made without loss of generality, only to simplify the description of our algorithms, since it is known that atomic snapshots can be wait-free implemented from single-writer multi-reader registers. However, there is a cost in terms of efficiency: the best implementation has $O(n \log n)$ complexity [2].

The k-Set Agreement Problem. The *k-Set Agreement* problem has been introduced by Chaudhuri [8] to study whether the number of choices allowed to processes (namely, k) is related to the maximal number of processes that can crash (f). More precisely, there is a set of values $\mathcal{V}$ that can be proposed by the processes, $|\mathcal{V}| > k$ and $\perp \notin \mathcal{V}$Each process starts an execution with an arbitrary input value from $\mathcal{V}$, the value it proposes, and all correct processes have to decide on a value such that (1) any decided value has been proposed, and (2) no more than k different values are decided. The classic *consensus* problem is k-set agreement for $k = 1$. As mentioned in the introduction, k-set agreement can be solved if and only if $f < k$.

The Condition-Based Approach. The proposed values in an execution are represented as an *input vector,* such that the i-th entry contains the value $v \in \mathcal{V}$ proposed by p_i, or $\perp$ if p_i did not take any steps in the execution. We usually denote with I an input vector with all entries in $\mathcal{V}$, and with J an input vector with some entries equal to $\perp$. If at most f processes may crash, we consider only input vectors J with at most f entries equal to $\perp$, called *views.* Let $\mathcal{V}^n$ be the set of all possible input vectors with all entries in $\mathcal{V}$, and $\mathcal{V}^n_f$ be the set of all the vectors with at most f entries equal to $\perp$. For $I \in \mathcal{V}^n$, let $\mathcal{I}_f$ be the set of possible views, i.e., the set of all input vectors J with at most f entries equal to $\perp$, and such that I agrees with J in all the non-$\perp$ entries of J. For a set C, $C \subseteq \mathcal{V}^n$, let $\mathcal{C}_f$ be the union of the $\mathcal{I}_f$'s over all $I \in C$. Thus, in the set agreement problem, every vector $J \in \mathcal{V}^n_f$ is a possible input vector.

The *condition-based* approach consists of considering a subset C of $\mathcal{V}^n$, called a *condition*, that represents common input vectors for a particular distributed problem. We are interested in conditions C that, when *satisfied,* i.e., when the proposed input vector belongs to $\mathcal{C}_f$, make the problem solvable despite f process crashes. Notice that if a problem is solvable for a condition C, then it is solvable for any C' contained in C: the same protocol works. In this paper the problem considered is k-set agreement. We assume $f \geq k$, since as explained above when $f < k$, k-set agreement is solvable for $\mathcal{V}^n$, hence for any condition C. More precisely, we say that an *f-fault tolerant protocol solves the k-set agreement problem for a condition C* if in every execution whose input vector J belongs to $\mathcal{V}^n_f$, the protocol satisfies the following properties:

– Validity: A decided value is a proposed value.
– k-Agreement: No more than k different values are decided.
– (k, C)-Termination : If (1) $J \in \mathcal{C}_f$ and no more than f processes crash, or (2.a) less than k processes crash, or (2.b) a process decides, then every correct process decides.

The first two are the validity and agreement requirements of the standard set agreement problem, and are independent of a particular condition C; they should hold even if the input pattern does not belong to C. The third requirement, requires termination under "normal" operating scenarios, including inputs belonging to C, and runs where less than k processes crash, a situation where k-set agreement is solvable. Part (1), requires termination even in executions where some processes crash initially and their inputs are unknown to the other processes. This is represented by a view J with $\perp$ entries for those processes. Termination is required if it is possible that the full input vector belongs to C; i.e., if J can be extended to an input vector $I \in C$. Part (2) defines two well-behaved scenarios where a protocol should terminate even if the input vector does not belong to C.

3 Conditions for Set Agreement

Here we define the class $\mathcal{S}_{f,k}$ of conditions, for each pair f, k. In the next section we prove that any $C \in \mathcal{S}_{f,k}$ allows k-set agreement to be solved with f faults. We also show here that $\mathcal{S}_{f,1} \subset \mathcal{S}_{f,2} \subset \cdots \subset \mathcal{S}_{f,f-1} \subset \mathcal{S}_{f,f}$. For the case of $k = 1$, we know from [20] that consensus can be solved for C only if $C \in \mathcal{S}_{f,1}$. We prove a similar impossibility for the wait-free case.

The class $\mathcal{S}_{f,k}$ consists of all conditions C whose views $\mathcal{C}_f$ can be colored with input values, such that (i) the color assigned to a view J is one of the inputs appearing in J, and (ii) a set of views that can be ordered by containment have at most k different colors assigned. We next describe two perspectives of this definition called acceptability and legality (following the style of [20,21]). We will use the first to derive protocols, and we use the second to show lower bounds. Either of these descriptions shows that deciding if a condition C belongs to $\mathcal{S}_{f,k}$ is computable, in contrast to the situation for arbitrary decision problems [12,13]. Although we do not know if this can be done in polynomial time, except for the case of $k = 1$[20].

3.1 The Class $\mathcal{S}_{f,k}$

We use the following notation. For vectors $J1, J2 \in \mathcal{V}_f^n$, $J1 \leq J2$ if $\forall k: J1[k] \neq \bot \Rightarrow J1[k] = J2[k]$, and we say that $J2$ *contains* $J1$. Let $\#_x(J)$ denote the number of entries of J whose value is x, with $x \in \mathcal{V} \cup \{\bot\}$.

Acceptability of a Condition. Given a condition C and values for k and f, acceptability is defined in terms of a coloring function S described above, and a predicate P that indicates when is S defined, that have to satisfy three properties in order that a protocol can be designed. The intuition for the first property, $\mathrm{T}_{C \to P}$ is that the predicate P allows a process p_i to test if a decision value can be computed from its local view. The intuition that underlies the second property, $\mathrm{V}_{P \to S}$, is clear: a decided value has to be a proposed value. The intuition that underlies the third property, $\mathrm{A}_{P \to S}$, is as follows. If several processes get the local views $J_1, J_2, \ldots, J_\ell$, ordered by containment, and such that $P(J_1), \ldots, P(J_\ell)$ are satisfied, these processes have to decide at most k different values. Formally:

– Property $\mathrm{T}_{C \to P}$: $\forall J \in \mathcal{V}_f^n: \ J \in \mathcal{C}_f \Rightarrow P(J)$.
– Property $\mathrm{V}_{P \to S}$: $\forall J \in \mathcal{V}_f^n: \ P(J) \Rightarrow S(J) =$ a non-$\bot$ value of J.
– Property $\mathrm{A}_{P \to S}$:

$$\forall J_1, \ldots, J_\ell \in \mathcal{V}_f^n: \ \left\{ \begin{array}{l} J_1 \leq J_2 \leq \cdots \leq J_\ell \\ \bigwedge_{i=1}^{\ell} P(J_i) \end{array} \right\} \Rightarrow |\{S(J_i) \ \textbf{s.t.} \ i \in [1..\ell]\}| \leq k.$$

Definition 1. *A condition C is (f,k)-*acceptable *if there exist a predicate P and a function S satisfying the properties $\mathrm{T}_{C \to P}$, $\mathrm{A}_{P \to S}$ and $\mathrm{V}_{P \to S}$ for f and k. Any such P, S are said to be* associated *with C, f and k.*

When clear from the context, we omit mentioning the parameter f. Notice that when $k = 1$, the (f,k)-acceptability definition reduces to the consensus acceptability definition of [20].

Legality of a Condition. In [20], we described an equivalent formulation of $(f,1)$-acceptability, in terms of a graph, $Gin(C,f)$ (close to the graph defined in [5] for $f = 1$): Its vertices are $\mathcal{C}_f$, i.e., the input vectors I of C plus all their views $J \in \mathcal{I}_f$ for every I in C. Two views $J1, J2 \in \mathcal{C}_f$ are connected by an edge if $J1 \leq J2$. (Hence, two vertices $I1$, $I2$ of C are connected if their Hamming distance $d(I1,I2) \leq f$.) The graph $Gin(C,f)$ is made up of one or more connected components, namely, $G_1, \ldots, G_x$. A condition C is $(f,1)$-*legal* if, for each connected component of $Gin(C,f)$, all the vertices that belong to this component have at least one input value in common.

For the k-set agreement problem, we need to generalize from a graph, which is a one dimensional structure, to a *complex,* which is a higher dimensional structure: instead of joining pairs of vertices to form edges, we need to join sets of vertices to form *simplexes.* To motivate the generalization, think of the $(f,1)$-legality definition, as requiring the existence of a map δ from $Gin(C,f)$ to a graph $Gout(C,f)$ consisting of one vertex for each element of $\mathcal{V}$, and no edges.

The map δ chooses the input value in common of a component. It is easy to see that δ sends vertices of $Gin(C, f)$ to vertices of $Gout(C, f)$ in a way that $\delta(J)$ is a value of J, and such that two vertices which are connected by an edge are mapped to the same vertex (a vertex is a singleton simplex). Such a map is called *simplicial* because it sends a set of vertices that form a simplex into a set of vertices that form a simplex (intuitively, it sends connected pieces of one complex to connected pieces of another complex).

Let $\mathcal{K}in(C, f, k)$ be the complex whose vertices are $\mathcal{C}_f$, and whose simplexes are all sets of at most $k+1$ vertices, $\{J_1, J_2, \ldots, J_\ell\}$, that can be ordered by containment, i.e., $J_1 \leq J_2 \leq \cdots \leq J_\ell$. In particular, every vertex is itself a simplex. Let $\mathcal{K}out(C, f, k)$ be the complex whose vertices are $\mathcal{V}$, and whose simplexes are all subsets of $\mathcal{V}$ with at most k different values (so it depends only on $\mathcal{V}$ and k).

Definition 2. *A condition C is* (f, k)-legal *if there exists a simplicial map δ from $\mathcal{K}in(C, f, k)$ to $\mathcal{K}out(C, f, k)$ such that $\delta(J)$ is a value that appears in J.*

It is straightforward to check that both definitions are equivalent:

Lemma 1. *A condition C is (f, k)-acceptable if, and only if, it is (f, k)-legal.*

Thus, we define $\mathcal{S}_{f,k}$ to be the set of all (f, k)-acceptable conditions, or equivalently, the set of all (f, k)-legal conditions, and we have:

$$\mathcal{S}_{1,f} \subseteq \mathcal{S}_{2,f} \subseteq \cdots \subseteq \mathcal{S}_{f-1,f} \subseteq \mathcal{S}_{f,f}.$$

3.2 Two Conditions

This section presents two conditions and proves them be in $\mathcal{S}_{f,k}$. They can be seen as generalizations of conditions stated in [20] for the consensus problem.

Condition $C1$. For each pair (f, k) we define a condition $C1_{f,k}$. For this we assume the set $\mathcal{V}$ is ordered. Its aim is to allow a process to decide the greatest value it has seen (the smallest value could be used instead). Let us denote a_i the i-th greatest value of a vector J; e.g. a_1=max(J) the greatest value and a_2 is the second greatest value of J. We have $a_1 > a_2 > \cdots > a_\ell$, where ℓ is the number of different values of $\mathcal{V}$ that are in J. If $\ell < k$, we assume for notational convenience that $\#_{a_{\ell+1}} = \cdots = \#_{a_k} = 0$. With these notations $C1_{f,k}$ can be stated as follows:

$$\boxed{I \in C1_{f,k} \quad \textit{iff} \quad \textstyle\sum_{i=1}^{k} \#_{a_i}(I) > f \ .}$$

Let $J \in \mathcal{I}_f$. The genericity parameters associated with $C1_{f,k}$ are:

- $P1(J) \equiv \sum_{i=1}^{k} \#_{a_i}(J) > f - \#_\perp(J)$.
- $S1(J) = \max(J)$.

Theorem 1. *The k-set agreement problem is solvable for $C1_{f,k}$ (i.e., $C1_{f,k} \in \mathcal{S}_{f,k}$).* (Proof in [22].)

Condition $C2$. For a vector J with ℓ different input values (different from $\bot$), let x_1 denote the occurrence number of its most common value, x_2 the occurrence number of its second most common value, and so on. Thus we have $x_1 \geq x_2 \geq \cdots \geq x_\ell$. When $\ell < (k+1)$, it will be convenient to set $x_{\ell+1} = x_{\ell+2} = \cdots = x_{k+1} = 0$.

With the previous notations, $C2$ for f and k (denoted $C2_{f,k}$) is stated as follows for any vector $I \in \mathcal{V}^n$ (recall that I has no entry equal to $\bot$):

$$\boxed{I \in C2_{f,k} \quad \textit{iff} \quad \sum_{i=1}^{k} x_i - k\ x_{k+1} > f\ .}$$

The underlying intuition is the following. We would like any process p_i to decide the value it sees the most often and no more than k values should be decided. Hence, at least one of the k most common values of I has to appear more than its $(k+1)$-th most common value despite the occurrence of up to f crashes. In this context, the pigeonhole principle ensures that at least one value from the k most common values of I appears at least $(\sum_{i=1}^{k} x_i - f)/k$ times. The condition $(\sum_{i=1}^{k} x_i - f)/k > x_{k+1}$ guarantees a safe decision. Let $J \in \mathcal{I}_f$. The parameters associated with $C2_{f,k}$ are:

- $P2(J) \equiv \sum_{i=1}^{k} x_i - k\ x_{k+1} > f - \#_\bot(J)$.
- $S2(J) = a: \ \#_a(J) = x_1$.

Theorem 2. *The k-set agreement problem is solvable for $C2_{f,k}$ (i.e., $C2_{f,k} \in \mathcal{S}_{f,k}$).* (Proof in [22].)

3.3 Proving $\mathcal{S}_{f,k} \neq \mathcal{S}_{f,k+1}$

As we have seen above, $\mathcal{S}_{f,k} \subseteq \mathcal{S}_{f,k+1}$. Here we prove that this containment is strict, by showing that $C1_{f,k+1}$ (which belongs to $\mathcal{S}_{f,k+1}$) is not in $\mathcal{S}_{f,k}$. Then Lemma 1 implies the claim. We start by proving some topological properties of the complex $\mathcal{K}in(C1_{f,k}, f, k)$. Namely, that it has some acyclic parts. A complex $\mathcal{K}$ is *acyclic* if it has no "holes", and it is *q-acyclic* if it has no "holes" up to dimension q; for $q = 0$ no "holes" means connected. Precise definitions appear in the appendix.

Let $\mathcal{K}in(C1_{f,k}, f, k)\,\{\mathcal{U}\}$ be the subcomplex of $\mathcal{K}in(C1_{f,k}, f, k)$ generated by the condition $C1_{f,k}\,\{\mathcal{U}\}$, which contains all input vectors of $C1_{f,k}$ with input values from a set $\mathcal{U}$, $\mathcal{U} \subseteq \mathcal{V}$. Notice that if $|\mathcal{U}| \leq k$, then $C1_{f,k}\,\{\mathcal{U}\} = \mathcal{U}^n$. The following technical result is proved in the appendix using techniques of [16].

Lemma 2. *Let $C = \mathcal{U}^n$. Then $\mathcal{K}in(C, f, k)$ is $(f-1)$-acyclic.*

Let $\mathcal{P}$ be the function that assigns to each simplex S of $\mathcal{K}out(C1_{f,k+1}, f, k{+}1)$ the (non-empty) subcomplex $\mathcal{K}in(C1_{f,k+1}, f, k{+}1)\,\{\mathcal{U}\}$ of $\mathcal{K}in(C1_{f,k+1}, f, k{+}1)$, where $\mathcal{U}$ is the set of input values in S. Then, it follows directly from Lemma 2, that $\mathcal{P}$ is an acyclic carrier [15]:

Lemma 3. *$\mathcal{P}$ is an acyclic carrier, namely: (1) $\mathcal{P}(S^m)$ is non-empty and $(m-1)$-acyclic; (2) $S \subseteq S'$ implies $\mathcal{P}(S) \subseteq \mathcal{P}(S')$.*

Lemma 4. *$C1_{f,k+1}$ is not in $\mathcal{S}_{f,k}$.* (Proof in [22].)

3.4 The Wait-Free Lower Bound

Let $k \leq f$, and $f = n - 1$. Assume that there is a k-set agreement protocol for a condition C. We show that then $C \in \mathbb{S}_{f,k}$ by proving that every input vector $I \in C$ contains at most k different input values (similar to the algebraic proof in [15] or its combinatorial version in [3]). This will imply the result, because any decision function δ that for each view J of $\mathcal{C}_f$ chooses some value of J trivially satisfies the legality (or the acceptability) definition. This is due to the fact that the union of a set of views of $\mathcal{C}_f$ ordered by containment is also a view of $\mathcal{C}_f$, and hence they can contain at most k different values overall.

Assume for contradiction that C contains I, an input vector with at least $k+1$ different input values. Pick a set U of $k+1$ different values of I. Let $\mathcal{S}^k$ be the complex that consists of a simplex S^k with $k+1$ vertices v_i, one for each value in U, and all faces (subsets) of S^k. For a simplex S in $\mathcal{S}^k$, let $val(S)$ be the union of the values labeling vertices of S. Let I' be the face of I with input values in U. Consider $\dot{\xi}_f(\mathcal{I}')$, the complex of all possible views of I'.

For $i \leq k$ there is an acyclic carrier Σ' from $\mathcal{S}^k$ to $\dot{\xi}_f(\mathcal{I}')$ that assigns to each i-dimensional S^i the subcomplex $\Sigma'(S^i) = \dot{\xi}_i(\mathcal{I}')\left\{val(S^i)\right\}$, the subcomplex of $\dot{\xi}_f(\mathcal{I}')$ spanned by inputs in the set $val(S^i)$. This carrier is acyclic by Lemma 3 because $i \leq k \leq f$. Therefore, we know from topology, that there is a chain map σ' from $C(\mathcal{S}^k)$ to $C(\dot{\xi}_f(\mathcal{I}'))$ carried by Σ'.

Let $\mathcal{P}$ map each input simplex to the complex of final states of the protocol reachable from it [17]. We know (e.g. [17,15]) that $\mathcal{P}$ is an acyclic carrier from $\dot{\xi}_f(\mathcal{I}')$ to the protocol complex, denoted also $\mathcal{P}$. So, there is a chain map σ from $C(\dot{\xi}_f(\mathcal{I}'))$ to $C(\mathcal{P})$ carried by $\mathcal{P}$, called an *algebraic span* [15]. Thus, $\sigma(S)$ is a chain with process ids belonging to S.

Considering the chain map δ corresponding to the decision map of the protocol, and the chain map π corresponding to the projection from $\mathcal{O}$ to $\mathcal{V}$ (the complex consisting of a simplex with one vertex per value in I, plus all its faces), and the chain map π' corresponding to a projection from $\mathcal{V}$ to $\mathcal{S}^k$. We have

$$C(\mathcal{S}^k) \xrightarrow{\sigma'} C(\dot{\xi}_f(\mathcal{I}')) \xrightarrow{\sigma} C(\mathcal{P}) \xrightarrow{\delta} C(\mathcal{O}) \xrightarrow{\pi} C(\mathcal{V}) \xrightarrow{\pi'} C(\mathcal{S}^k).$$

Let $\phi : C(\mathcal{S}^k) \to C(\mathcal{S}^k)$, be the composition of σ', σ, δ, π and π'. Let Φ be the the trivial acyclic carrier from $\mathcal{S}^k$ to itself. Notice that Φ carries ϕ, by the validity requirement of set agreement. The identity chain map ι is also carried by Φ. Thus ϕ and ι are equal, since Φ preserves dimension [22]. Now, S^k of $\mathcal{S}^k$ is labeled with $k+1$ different sets, and its mapped by ι to itself. Thus, $\phi(S^k) = S^k$, and there must be an execution with $k+1$ different values, one value from each V_i (otherwise $\delta \circ \pi$ is zero in dimension k). We have shown

Theorem 3. *If there is a wait-free k-set agreement protocol for a condition C then C does not contain vectors with more than k different input values. Therefore, C is (f,k)-legal.*

4 A Condition-Based k-Set Agreement Protocol

Figure 1 presents a protocol that solves k-set agreement for any condition $C \in \mathcal{S}_{f,k}$. This protocol assumes P and S have been instantiated to correspond with the parameters associated with C.

The protocol uses a fixed, deterministic function F, that returns $F(J)$, a non-$\bot$ value of the vector J to which it is applied (any such function will do, such as max). An implementation of F that does not depend on ordering the input values is the following one. Let Π_k be a predetermined set of k processes. $F(J)$ considers the values of the entries of J corresponding to the processes of Π_k, and outputs the first of them (according to the process order) that is different from $\bot$. The function F will be applied only to vectors J that contain at most $(k-1)$ entries equal to $\bot$, and such that they can be ordered by containment. Thus, F is applied to a set of at most k such vectors, and hence it satisfies the following property:

Property 1. Let J_1, $J_2, \dots, J_\ell$ be a set of vectors, $J_1 \le J_2 \le \cdots \le J_\ell$, and such that each contains at most $(k-1)$ entries equal to $\bot$. Then, $\{F(J_1), \dots, F(J_\ell)\}$, contains at most k different values.

Protocol description. A process p_i starts the k-set agreement protocol by invoking *k-Set_Agreement*(v_i) where v_i is the value it proposes. It terminates when it executes the statement **return** which provides it with a decided value (at line 5, 8 or 11). A process p_i first writes its input to a shared array V (line 1). Then p_i reads V until it sees at least $n-f$ input values, and once this happens it takes a snapshot of V to create its local view V_i (line 3). This process guarantees that all local views contain at least $n-f$ non-$\bot$ entries, and can be ordered by containment. From this local view V_i, p_i computes its decision estimate w_i (line 4), using the parameters P and S associated with the condition C, and makes w_i visible to the other processes by writing it into the shared array $D[i]$ (line 5). Thus, w_i will be different from $\bot$ and $\top$ if the predicate P is evaluated to true, and in this case p_i will decide (line 5). By writing w_i to D, p_i helps the other processes that might not have evaluated P to true to decide (this is the case when the input vector $\notin C$). The write statement line 6 informs the other processes that p_i has executed the first part of the protocol but was not able to decide whereas the default initial value $\bot$ means that either p_i has not yet executed its protocol or p_i is crashed. The remaining of the protocol is the "best effort termination" part. Here p_i enters a loop (line 7) during which it looks for a decision estimate (e.g., a non-$\bot$ and non-$\top$ value by p_j in $D_i[j]$) that allows it to decide or possibly a situation where $\#_\bot(D_i) < k$. Notice that the latter case is the only place where k is used. Then p_i builds a local view Y_i of the input vector (lines 9-10). These views can be ordered by containment. Finally, p_i decides on the value $F(Y_i)$. Note that if p_i terminates at line 11, due to lines 9-10 and the test line 7, the array Y_i has at most $(k-1)$ entries equal to $\bot$. Property 1 combined with the function F and the linearizability of the $\mathsf{snapshot}(D)$ invocations at line 7 ensures that at most k different values can be decided at line 11. The proof will show a stronger property, namely, $\alpha + \beta \leq k$ where α (resp. β) is the number of values decided at lines 5 and 8 (resp. line 11.)

```
Function $k$-Set_Agreement($v_i$)

(1)  write($v_i, V[i]$);
(2)  repeat $V_i \leftarrow$ collect($V$) until $(\#_\perp(V_i) \leq f)$;
(3)  $V_i \leftarrow$ snapshot($V$);
(4)  if $P(V_i)$ then $w_i \leftarrow S(V_i)$ else $w_i \leftarrow \top$ endif;
(5)  if $w_i \neq \top$ then write($w_i, D[i]$); return($w_i$)
(6)     else write($\top, D[i]$);
(7)          repeat $D_i \leftarrow$ snapshot($D$) until $\big((\exists\ D_i[j] \neq \perp, \top) \vee (\#_\perp(D_i) < k)\big)$;
(8)          if $(\exists\ D_i[j] \neq \perp, \top)$ then return($D_i[j]$)
(9)                 else $\forall j$: if $(D_i[j] = \top)$ then $Y_i[j] \leftarrow$ read($V[j]$)
(10)                                  else $Y_i[j] \leftarrow \perp$   endif;
(11)                     return($F(Y_i)$)
(12)         endif
(13) endif
```

Fig. 1. A Generic Condition-Based k-Set Agreement Protocol

Theorem 4. *For any condition $C \in \mathcal{S}_{f,k}$, the k-Set_Agreement protocol satisfies the* Validity*, k-*Agreement *and (k, C)-*Termination *properties.* (Proof in [22].)

5 An Efficient Set Agreement Protocol

The protocol of Figure 1 solves k-set agreement for any condition $C \in \mathcal{S}_{f,k}$. In particular, for $k = 1$, it solves consensus for $C \in \mathcal{S}_{f,1}$, as in [20]. The cost of the condition-dependent part of the protocol can be considered to be $O(n \log n)$ steps, since this is the known complexity [2] of implementing the snapshot operation of line 3 with read/write operations. Here, we show how to trade fault-tolerance for agreement precision using only $O(n)$ steps, namely, in this case, it is possible to solve k-set agreement for $k = \lfloor f/(n-f) \rfloor + 1$. It follows from this trading that there is a consensus protocol that requires only $O(n)$ steps when $f < n/2$ or the input vector $J \in C_f$ with $C \in \mathcal{S}_{f,1}$. As indicated in the introduction, this improves upon the results of [21].

5.1 An Efficient Set Agreement Protocol for Consensus Conditions

Such a protocol trading fault-tolerance for agreement precision is described in Figure 2. It has been designed with a structure as close as possible to the structure of the protocol described in Figure 1. Hence, it is self-explanatory (the additional shared register W is initialized to $[\perp, \ldots, \perp]$). It assumes that P and S have been instantiated to correspond with the parameters associated with a given condition $C \in \mathcal{S}_{f,1}$. It always solves k-set agreement for $k = \lfloor f/(n-f) \rfloor + 1$, but if the input vector J actually belongs to C_f, then it satisfies a stronger agreement property, namely, it solves consensus. More explicitly, the protocol satisfies the following specification:

Specification 1 (Set Agreement + Consensus for inputs in $C \in \mathcal{S}_{f,1}$)

– Validity: *A decided value is a proposed value.*
– Set Agreement: *No more than $\frac{f}{n-f}+1$ values are decided. Moreover, if $J \in C$ then no two processes decide different values.*
– Termination: *If (1) $J \in C$ and there are at most f crashes, or (2) no more than $\frac{f}{n-f}$ processes crash, or (3) a process decides, then all correct processes decide.*

```
Function $Set\_Agreement(v_i)$

(1)  write($v_i, V[i]$);
(2)  repeat $V_i \leftarrow$ collect($V$) until $(\#_\perp(V_i) \le f)$;
(3)  if $P(V_i)$ then $w_i \leftarrow S(V_i)$ else $w_i \leftarrow \top$ endif;
(4)  write($w_i, W[i]$);
(5)  repeat $W_i \leftarrow$ collect($W$) until $(\#_\perp(W_i) \le f)$;
(6)  if (the same $w \ne \perp, \top$ appears $(n-f)$ times in $W_i$)
(7)     then write($w, D[i]$); return($w$)
(8)     else write($\top, D[i]$);
(9)          repeat $D_i \leftarrow$ snapshot($D$) until $((\exists\ D_i[j] \ne \perp, \top) \vee (\#_\perp(D_i) \le \frac{f}{n-f}))$;
(10)         if $(\exists\ D_i[j] \ne \perp, \top)$ then return($D_i[j]$)
(11)                        else $\forall j$: if $(D_i[j] = \top)$ then $Y_i[j] \leftarrow$ read($V[j]$)
(12)                                                           else $Y_i[j] \leftarrow \perp$   endif;
(13)                             return($F(Y_i)$)
(14)         endif
(15) endif
```

Fig. 2. A Condition-Based Set Agreement Protocol for $C \in \mathcal{S}_{f,1}$

The protocol described in Figure 1 uses a strong operation at line 3 (snapshot) that is suppressed from the protocol of Figure 2. This has no effect as long as the input vectors belong to the condition with which the protocol is instantiated. However, if this is not the case the agreement property is no longer guaranteed, because there is no order between the views. This is why a second phase serving as a filter (lines 4 and 5) is added to the protocol of Figure 2. Consequently, the first protocol allows to decide as soon as the predicate P is evaluated to true whereas the second protocol waits for the values collected during the second phase to decide a value that appeared sufficiently many times. Of course, in the latter case, the "quality" of the decision depends on the number of failures; in particular, if $f < n/2$ the problem it solves is always consensus.

It is important to see that this protocol solves consensus without snapshot operations when the input vector $J \in C_f$ (the snapshot operation at line 3 of Figure 1 has been eliminated). This shows that the protocol is particularly efficient in "normal circumstances". More generally, Figure 3 depicts the maximal number k of values decided by this protocol, according to the ratio $x = f/n$, i.e., the percentage of processes that may crash. More precisely, $k = \lfloor \frac{x}{1-x} \rfloor + 1$. As we can see, when less than a majority of processes are allowed to crash, the protocol guarantees consensus (and, as indicated before, if $J \in C_f$ without a snapshot). Then, the degradation of its "quality of service" (increase of k) is more and more severe when the ratio f/n increases.

Theorem 5. *The Set_Agreement protocol satisfies the* Validity, Set Agreement *and* Termination *properties defined in Specification 1.* (Proof in [22].)

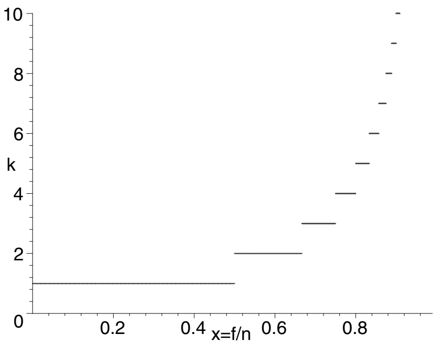


Fig. 3. Maximal Size of the Decision Set

References

1. Attiya H. and Avidor Z., Wait-Free n-Consensus When Inputs are Restricted. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, These proceedings.
2. Attiya H. and Rachman O., Atomic Snapshots in O($n \log n$) Operations. *SIAM Journal on Computing,* 27(2):319-340, 1998.
3. Attiya H. and Rajsbaum S., The Combinatorial Structure of Wait-free Solvable Tasks. To appear in *SIAM Journal on Computing,* 2002.
4. Berman P. and Garay J., Adaptability and the Usefulness of Hints. *6th European Symposium on Algorithms,* Springer-Verlag LNCS #1461, pp. 271-282, 1998.
5. Biran O., Moran S. and Zaks S., A Combinatorial Characterization of the Distributed 1-Solvable Tasks. *Journal of Algorithms*, 11:420-440, 1990.
6. Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM STOC*, pp. 91-100, 1993.
7. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
8. Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.
9. Dolev D., Dwork C. and Stockmeyer L., On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, 1987.

10. Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
11. Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
12. Gafni E. and Koutsoupias E., Three-Processor Tasks Are Undecidable. *SIAM Journal of Computing,* 28(3):970-983, 1999.
13. Herlihy M.P. and Rajsbaum S., On the Decidability of Distributed Decision Tasks. *Proc. 29th ACM STOC*, pp. 589-598, 1997.
14. Herlihy M. and Rajsbaum S., New Perspectives in Distributed Computing. *Invited Talk, Proc. 24th Int. Symposium on Mathematical Foundations of Computer Science (MFCS'99)*, Springer-Verlag LNCS #1672, pp. 170-186, 1999.
15. Herlihy M. and Rajsbaum S., Algebraic Spans. *Mathematical Structures in Computer Science,* 10(4):549-573, 2000.
16. Herlihy M., Rajsbaum S. and Tuttle M., Synchronous Round Operators, 2000.
17. Herlihy M. and Shavit N., The Asynchronous Computability Theorem for t-Resilient Tasks. *Proc. 25th ACM STOC*, CA, pp. 111-120, 1993.
18. Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 12(3):463-492, 1990.
19. Keidar I. and Rajsbaum S., On the Cost of Fault-Tolerant Consensus When There Are No Faults - A Tutorial, *SIGACT News, DC Column*, 32(2):45-63, 2001.
20. Mostéfaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Proc. 33rd ACM STOC*, pp. 153-162, 2001.
21. Mostéfaoui A., Rajsbaum S., Raynal M. and Roy M., A Hierarchy of Conditions for Consensus Solvability. *Proc. PODC'01*, pp. 151-160, 2001.
22. Mostéfaoui A., Rajsbaum S., Raynal M. and Roy M., Condition-Based Protocols for Set Agreement Problems. *Research Report #1464*, IRISA, University of Rennes, France, 2002. `http://www.irisa.fr/bibli/publi/pi/2002/1464/1464.html`.
23. Mostefaoui A. and Raynal M., k-Set Agreement with Limited Accuracy Failure Detectors. *Proc. PODC'99*, Portland (OR), pp. 143-152, 2000.
24. Mostefaoui A. and Raynal M., Randomized k-Set Agreement. *Proc. 13th th ACM Symp. on Parallel Algorithms and Architectures (SPAA'01)*, pp. 291-297, 2001.
25. Taubenfeld G., Katz S. and Moran S., Impossibility Results in the Presence of Multiple Faulty Processes. *Information and Computation*, 113(2):173-198, 1994.
26. Taubenfeld G. and Moran S., Possibility and Impossibility Results in a Shared Memory Environment. *Acta Informatica*, 35:1-20, 1996.
27. Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible: the Topology of Public Knowledge. *Proc. 25th ACM STOC*, pp. 101-110, 1993.

Distributed Agreement and Its Relation with Error-Correcting Codes*

R. Friedman[1], A. Mostéfaoui[2], S. Rajsbaum[3], and M. Raynal[2]

[1] Department of Computer Science, The Technion, Haifa, Israel,
`roy@cs.technion.ac.il`,
[2] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France,
`{achour,raynal}@irisa.fr`,
[3] HP Research Lab, Cambridge, MA 02139, USA and Inst. Matem. UNAM, Mexico,
`Sergio.Rajsbaum@hp.com`

Abstract. The *condition based* approach identifies sets of input vectors, called *conditions*, for which it is possible to design a protocol solving a distributed problem despite process crashes. This paper investigates three related agreement problems, namely consensus, interactive consistency, and k-set agreement, in the context of the condition-based approach. In consensus, processes have to agree on one of the proposed values; in interactive consistency, they have to agree on the vector of proposed values; in k-set agreement, each process decides on one of the proposed values, and at most k different values can be decided on. For both consensus and interactive consistency, a direct correlation between these problems and error correcting codes is established. In particular, crash failures in distributed agreement problems correspond to erasure failures in error correcting codes, and Byzantine and value domain faults correspond to corruption errors. It is also shown that less restrictive codes can be used to solve k-set agreement, but without a necessity proof, which is still an open problem.

Keywords: Asynchronous Distributed System, Code Theory, Condition, Consensus, Crash Failure, Distributed Computing, Erroneous Value, Error-Correcting Code, Fault-Tolerance, Hamming Distance, Interactive Consistency.

1 Introduction

Context of the paper. Agreement problems are among the most fundamental problems in designing and implementing reliable applications on top of an asynchronous distributed environment prone to failures [3,21]. Among these problems, *consensus* has been the most widely studied. Specifically, in consensus, each process proposes a value, and all processes have to agree on the same proposed value. In the *interactive consistency* problem (initially stated in [30] for the case of Byzantine failures) each process proposes a value, and processes have to agree on the same vector, such that the i-th entry of the vector contains the

* As decided by the program committee, this paper results from the merging of [17] and [24], which were developed separately.

value proposed by process p_i if it is correct. Interactive consistency is at least as difficult as consensus: a solution to interactive consistency can be used to solve consensus.

Given the importance of agreement problems in distributed computing, it is remarkable that they cannot be solved in an asynchronous system if only one process can fail, and only by crashing. More precisely, it can be shown that for any protocol that tries to ensure agreement, there is an infinite run in which no process can decide. The first such result is the FLP impossibility for consensus in a message passing system [15], which has been extended to many other agreement problems and distributed models [3,21].

Over the years, researchers have investigated ways of circumventing these impossibility results. The first approach for overcoming the impossibility result considers weaker agreement problems, such as approximate agreement [13], set agreement [10], and randomized solutions [5]. The second approach considers stronger environments that are only partially asynchronous [12,14] and/or have access to failure detectors [11,18]. The third, *condition based* approach, is the focus of this paper [23,25,26,27,34,35]. This approach consists of restricting the set of possible input configurations to a distributed problem. An input configuration can be represented by a vector whose entries are the individual processes' input values in an execution. It has been shown [23] that in asynchronous environments, consensus is solvable despite f failures when the set of allowed input vectors obey certain conditions, which are shown to be necessary and sufficient. This area is also related to the work of [8], which developed an approach for designing algorithms that can utilize some information about the typical conditions that are likely to hold when they are invoked.

The condition based approach can serve two complementary purposes. It can first be viewed as an optimization tool. That is, it allows to identify scenarios where it is always possible to guarantee termination of agreement protocols, yet design the protocols in a way that will prevent them from reaching unsafe decisions even if these scenarios (conditions) where not met. As can be seen from [23, 25,26,27], such scenarios or conditions, are likely to be common in practical systems. Second, as it becomes evident from this work (Section 5), one can utilize the conditions as a guideline to augmenting the environment with optimal levels of synchrony that are cheap enough to support, yet guarantee the solvability of the corresponding agreement problem. In these cases, it is then possible to employ highly efficient protocols that terminate in at most two communication steps even when there are failures. Thus, in comparing this approach to failure detectors, the latter can be viewed as an abstraction that encapsulates the minimal synchrony assumptions on the environment in terms of the ability to detect failures in order to solve agreement problems. On the other hand, a result of this paper shows that the condition based approach can be viewed as capturing minimal requirements on the number of synchronous communication links to solve these problems.

Content of the paper. This paper takes the condition based approach a step further, and establishes a direct relation between error-correcting codes and agreement problems, based on the notion of condition based agreement. In particular, in this paper we obtain the following results. First, we show that the conditions that allow interactive consistency to be solved despite f_c crashes and f_e value domain faults is exactly the set of error correcting codes capable of recovering from f_c erasures and f_e corruptions. Second, we prove that consensus can be solved despite f_c crash failures iff the condition corresponds to a code whose Hamming distance is $f_c + 1$ and Byzantine consensus can be solved despite f_b Byzantine faults iff the Hamming distance of the code is $2f_b + 1$. (There is in fact an additional requirement needed to satisfy the validity property of consensus, as discussed later in the paper.) Third, we show a code that allows solving k-set agreement in both the benign and Byzantine failure models using a simple protocol, but were not able to prove the necessity of the code.

The paper also presents several interesting results that are derived from the main results mentioned above. Namely, we show that by exploring error correcting codes, we can find the parameters needed by our protocols to solve interactive consistency and consensus. On the other hand, coding theory may benefit from this connection. As a simple example, we show that there are no perfect codes that tolerate erasure failures, for otherwise it would have violated the consensus impossibility results. Moreover, we discuss the practical implications of our work to the design choices of distributed consensus algorithms in mixed environments, and introduce the notion of cluster-based failure detectors. Furthermore, our results imply that coding theory can serve as a guideline to efficient deployment and utilization of sparse synchronous communication lines, as promoted, for example, by the *wormholes* approach [36].

The fact that agreement problems are solvable when the input vectors are limited to error correcting codes is not surprising. In particular, error correcting techniques were used in [6] as basic building blocks in constructions for computing arbitrary functions in synchronous systems. Thus, the main challenges of this work include finding algorithms that provide safety always and termination in all favorable circumstances. Additionally, the discussion of the cost of providing agreement, validity, and termination in all possible scenarios shed an important insight into these problems. Finally, the fact that these conditions are necessary for consensus is somewhat surprising, since the problem definition of consensus allows for initial bi-valent configurations. The interesting insight that comes out of the proofs is that these conditions are the minimal requirement to avoid initial bi-valent configurations, and this is exactly why they are required.

Road map. The paper is made up of six sections. Section 2 introduces the computation model and defines the main agreement problems we are interested in (namely, consensus and interactive consistency). Section 3 addresses condition-based interactive consistency and shows that the conditions that allow to solve this problem are exactly error-correcting codes. Section 4 provides a code-based characterization of the conditions that allow to solve consensus. Then, Section 5 focuses on the practical implications of the previous results. Finally, Section 6 concludes the paper. More details and proofs of theorems can be found in [17, 24].

2 Model and Problem Statement

In this paper we assume a standard asynchronous shared memory model or message passing model [3,21] according to our needs, and consider the typical notions of *processes*, *local histories*, *executions*, and *protocols*. We also assume a fixed set of n processes trying to solve an *agreement problem.* In agreement problems, each process has an initial input value, and must decide on an output value. In our model, in each execution at most $f_c < n$ processes can fail by *crashing.* Additionally, up to $f_e < n/2$ processes may suffer *value domain* errors [32] and up to $f_b < n/3$ incur *Byzantine* errors, depending on the circumstances. We sometimes simply write f to denote the total number of failures. A process that suffers a value domain error, also known as *value-faulty*, behaves as if it had a different input value than the one actually given to it, but must otherwise obey the protocol. On the other hand, a process that suffers a Byzantine error, also known as *Byzantine process*, behaves in an arbitrary manner. A process that does not crash and does not suffer any error is called *correct*; otherwise, it is *faulty.*

A universe of values $\mathcal{V}$ is assumed, together with a default value $\perp$ not in $\mathcal{V}$. In the *consensus* problem, each process p_i proposes a value $v_i \in \mathcal{V}$ (the input value of that process), and has to decide on a value (the output value), such that the following properties are satisfied:

– C-Agreement. No two different values are decided.
– C-Termination. A process that does not crash decides.
– C-Validity. A decided value v is a proposed value.

For Byzantine failures, we modify the requirements to be:

– BC-Agreement. No two different values are decided by correct processes.
– BC-Termination. A correct process decides.
– BC-Validity. If all proposed values are the same value v, then the value decided by correct processes is v.

The *interactive consistency* (IC) problem is defined as follows. Each process p_i proposes a value $v_i \in \mathcal{V}$ (the input value), and has to decide a vector D_i (the output value), such that the following properties are satisfied:

– IC-Agreement. No two different vectors are decided.
– IC-Termination. A process that does not crash decides.
– IC-Validity. Any decided vector D is such that $D[i] \in \{v_i, \perp\}$, and is v_i if p_i does not crash.

It is easy to see that, as noted in the Introduction, the IC problem is at least as hard as consensus, and hence unsolvable even if at most one process can crash. Also, it is possible to view the collection of input values to each of these problems as an input vector to the problem. We are interested in conditions on these input vectors that allow the IC and *consensus* problems to be solved despite process failures.

3 Condition-Based Interactive Consistency

3.1 Notation

Let the *input vector* associated with an execution be a vector J, such that $J[i]$ contains the value $v_i \in \mathcal{V}$ proposed by p_i or $\perp$ if p_i crashes initially and does not take any steps. Let $\mathcal{V}^n$ be the set of all possible vectors (of size n) with all entries in $\mathcal{V}$. We typically denote by I a vector in $\mathcal{V}^n$ and by J a vector that may have some entries equal to $\perp$, and hence in $\mathcal{V}^n_{f_c}$, the set of all the n-vectors over $\mathcal{V}$ with at most f_c entries equal to $\perp$. For vectors $J1, J2 \in \mathcal{V}^n_{f_c}$, $J1 \leq J2$ if $\forall k : \ J1[k] \neq \perp \Rightarrow J1[k] = J2[k]$. We define two functions:

- $\#_x(J) =$ number of entries of J whose value is x, with $x \in \mathcal{V} \cup \{\perp\}$.
- $d_{\not\perp}(I, J) =$ number of corresponding non-$\perp$ entries that differ in I and J.

When I has no entry equal to $\perp$, we have $d(I, J) = \#_\perp(J) + d_{\not\perp}(I, J)$. Where $d(I, J)$ is the Hamming distance, i.e., total number of entries where I and J differ. Given a vector $I \in \mathcal{V}^n$,

$$I_{f_c,f_e} \ = \ \{J \mid \#_\perp(J) \leq f_c \ \wedge \ d_{\not\perp}(I, J) \leq f_e \ \}$$

and for a subset C of $\mathcal{V}^n$,

$$\mathcal{C}_{f_c,f_e} \ = \ \bigcup_{I \in C} I_{f_c,f_e}.$$

Thus I_{f_c,f_e} represents a sphere of vectors centered at I and including the vectors J whose distances $\#_\perp(J)$ and $d_{\not\perp}(I, J)$ are bounded by f_c and f_e, respectively. Also, $\mathcal{C}_{f_c,f_e}$ is the union of the spheres centered in vectors of C.

3.2 The CB_IC Problem

As indicated in the Introduction, the idea of the condition-based approach is considering sets of input configurations for which a particular agreement problem can be solved. Such sets C are called conditions. In the IC problem, as in other agreement problems, a condition C is a subset of $\mathcal{V}^n$. It is assumed that C represents input configurations that are common in practice. Hence, it is required that the protocol terminates whenever the input configuration belongs, or could have belonged to C. That is, if the input vector is J (some processes may have crashed initially), termination is required if $J \leq I$ for some $I \in C$, since it is possible that the processes that crashed initially had inputs that would complete J into a vector $I \in C$. Similarly, termination is required if some (at most f_e) non-$\perp$ values of J can be changed to obtain a vector J', such that $J' \leq I$ for $I \in C$, since it is possible that the corresponding processes are value-faulty.

More precisely, here follows a condition-based version of the interactive consistency problem. Each process p_i proposes a value $v_i \in \mathcal{V}$ and has to decide a vector D_i. We say that an (f_c, f_e)*-fault tolerant protocol solves the* CB_IC *problem for a condition* C, if in every execution whose proposed vector J belongs to $\mathcal{V}^n_{f_c}$, the protocol satisfies the following properties:

- CB_IC-Agreement. No two different vectors are decided.
- CB_IC-Termination. If (1) $J \in \mathcal{C}_{f_c,f_e}$ and no more than f_c processes crash, or (2.a) no process crashes, or (2.b) a process decides, then every crash-correct process decides.
- CB_IC-Validity. If $J \in \mathcal{C}_{f_c,f_e}$, then the decided vector D is such that $J \in D_{f_c,f_e}$ with $D \in C$.

The agreement property states that there is a single decision, even if the input vector is not in C, guaranteeing "safety" always. The termination property requires that the processes that do not crash must decide at least when the circumstances are "favorable." Those are (1) when the input could have belonged to C, as explained above, (provided there are no more than f_c crashes during the execution), and (2) under normal operating conditions. The validity property eliminates trivial solutions by relating the decided vector and the proposed vector. It states that, when the proposed vector belongs to at least one sphere defined by the condition, the center of such a sphere is decided, which is one of the possible actual inputs that could have been proposed. To simplify the notation we do not allow $\perp$ entries in the decided vector (the same results apply).

3.3 The Interactive Consistency Conditions

We define here a set of conditions for which there is a solution to the CB_IC problem. Then, the next section presents a protocol that solves CB_IC for any condition in this set. As it can be seen from the definitions below, this set is quite restricted. Nevertheless, in the following section we prove that those are the only conditions for which the CB_IC problem can be solved.

Similarly to the approach used in [23], we define the set of conditions in two equivalent ways, called acceptability and legality. We start with acceptability, which is useful to derive protocols. We then consider legality, which is useful to prove impossibility results.

Acceptability is defined in terms of a predicate P and a function S. Given a condition C and an input vector $J \in \mathcal{V}^n_{f_c}$ proposed by the processes, $P(J)$ has to hold when $J \in \mathcal{C}_{f_c,f_e}$ to allow a process to decide at least in those cases. Then, $S(J)$ provides the process with the corresponding decision vector. To meet these requirements, P and S have to satisfy the following properties.

- Property $\mathrm{T}_{C\to P}$: $I \in C \Rightarrow \ \forall J \in I_{f_c,f_e} : \ P(J)$,
- Property $\mathrm{A}_{P\to S}$:
$$\forall J1, J2 \in \mathcal{V}^n_{f_c} : \quad (J1 \leq J2) \wedge P(J1) \wedge P(J2) \Rightarrow S(J1) = S(J2),$$
- Property $\mathrm{V}_{P\to S}$:
$$\forall J \in \mathcal{V}^n_{f_c} : \quad P(J) \ \Rightarrow \ S(J) = I \text{ such that } I \in C \ \wedge \ J \in I_{f_c,f_e}.$$

Definition 1. *A condition C is (f_c, f_e)-*acceptable *if there exist a predicate P and a function S satisfying the three previous properties.*

The following is a more geometric version of acceptability (where $d(,)$ is Hamming distance).

Definition 2. *A condition C is (f_c, f_e)-*legal *if for all distinct $I1, I2 \in C$, $d(I1, I2) \geq 2f_e + f_c + 1$.*

We will prove that the set of (f_c, f_e)-acceptable conditions is the same as the set of (f_c, f_e)-legal conditions, and this is precisely the set of conditions for which there exists an (f_c, f_e)-fault tolerant protocol that solves CB_IC.

3.4 A Shared Memory CB_IC Protocol

We show in this section that for any (f_c, f_e)-acceptable condition C there is a (f_c, f_e)-fault tolerant protocol that solves CB_IC. Such a protocol is described in Figure 1, which needs to be instantiated with parameters P and S associated to C. Interestingly, the protocol is very similar to the condition-based consensus protocol presented in [23], illustrating the relation between consensus and CB_IC.

Computation Model

We consider a standard asynchronous system made up of $n > 1$ processes, $p_1, \ldots, p_n$, that communicate through a single-writer, multi-reader shared memory, and where at most f_c, $1 \leq f_c < n$, processes can crash [3,21].

We assume the shared memory is organized into arrays. The j-th entry of an array $X[1..n]$ can be read by any processes p_i with an operation $\mathsf{read}(X[j])$. Only p_i can write to the i-th component, $X[i]$, and it uses the operation $\mathsf{write}(v, X[i])$ for this.

To simplify the description of our algorithms, we assume two primitives, collect and snapshot. The collect primitive is a non-atomic operation which can be invoked by any process p_i. It can only be applied to a whole array $X[1..n]$, and is an abbreviation for $\forall j$: **do** $\mathsf{read}(X[j])$ **enddo**. Hence, it returns an array of values $[a_1, \ldots, a_n]$ such that a_j is the value returned by $\mathsf{read}(X[j])$. The processes can take atomic snapshots of any of the shared arrays: $\mathsf{snapshot}(X)$ allows a process p_j to atomically read the content of all the registers of the array X. This assumption is made without loss of generality, since atomic snapshots can be wait-free implemented from single-writer multi-reader registers (although there is a cost in terms of efficiency: the best known simulation has $O(n \log n)$ time complexity [2]).

Each shared register is initialized to a default value $\perp$. In addition to the shared memory, each process has a local memory. The subindex i is used to denote p_i's local variables.

Protocol

A process p_i starts by invoking *SM_*CB_IC (v_i) with some $v_i \in \mathcal{V}$. It terminates when it executes the statement **return** (line 7, 9 or 10) which provides it with a decision vector. The shared memory is made up of two arrays of atomic registers, $V[1..n]$ and $W[1..n]$ (the aim of $V[i]$ is to contain the value proposed by p_i, while the aim of $W[i]$ is to contain the vector p_i suggests to decide on or $\top$ if p_i cannot decide by itself). The protocol has a three part structure.

- Local view determination: lines 1-3. A process p_i first writes into $V[i]$ the value it proposes (line 1). Then, it reads the array V until it gets a vector (V_i) including at least $(n - f_c)$ proposed values.
- Wait-free condition-dependent part: lines 4-5. If $P(V_i)$ holds, p_i computes its view $w_i = [a_1, \dots, a_n]$ of the decided vector. If p_i cannot decide, it sets w_i to $\top$. p_i also writes w_i in the shared register $W[i]$ to help the other processes decide.
- Termination part: lines 6-11. If $w_i \neq \top$, then p_i unilaterally decides the vector w_i. If it cannot decide by itself ($w_i = \top$) p_i waits until it knows (1) either that another process p_j has unilaterally decided, (2) or that no process can unilaterally decides. In the first case, it decides p_j's suggested vector, while in the second case it decides the full vector of proposed values.

```
**Function** $SM\_CB\_IC$ $(v_i)$

(1)  write($v_i, V[i]$);
(2)  **repeat** $V_i \leftarrow$ collect($V$) **until** $(\#_\perp(V_i) \le f_c)$ **endrepeat**;
(3)  $V_i \leftarrow$ snapshot($V$);
(4)  **if** $P(V_i)$ **then** $w_i \leftarrow S(V_i)$ **else** $w_i \leftarrow \top$ **endif**;
(5)  write($w_i, W[i]$);
(6)  **if** $(w_i \neq \top)$
(7)     **then return** $(w_i)$
(8)     **else repeat** $W_i \leftarrow$ collect($W$)
                 **until** $(\exists W_i[j] \neq \perp, \top) \vee (W_i = [\top, \dots, \top])$
           **endrepeat**;
(9)        **if** $(\exists W_i[j] \neq \perp, \top)$ **then return** $(W_i[j])$
(10)                                  **else return** (collect($V$))
(11)       **endif**
(12) **endif**
```

Fig. 1. A Shared Memory CB_IC Protocol

Theorem 1. *The protocol described in Figure 1 is an (f_c, f_e)-fault tolerant protocol that solves the* CB_IC *problem for a condition C, when it is instantiated with P, S associated to C, and C is (f_c, f_e)-acceptable.*

Proof Follows directly from the next three Lemmas. $\square_{Theorem\ 1}$

In the following J denotes the vector actually proposed by the processes.

Lemma 1. CB_IC-Validity. *If $J \in \mathcal{C}_{f_c,f_e}$, then the decided vector D is such that $J \in D_{f_c,f_e}$ with $D \in C$.*

Proof There are three cases according to the line (7, 9 or 10) at which a process decides.

- p_i decides at line 7. In that case, $P(V_i)$ held at line 4. Moreover, $V_i \leq J$ and $\#_\bot(V_i) \leq f_c$, from which we conclude that if $J \in \mathcal{C}_{f_c,f_e}$, we also have $V_i \in \mathcal{C}_{f_c,f_e}$. It then follows from $\mathrm{V}_{C \to P}$ that $S(V_i) = I$ with $I \in C$ and $J \in I_{f_c,f_e}$.
- p_i decides at line 9. In that case, p_i decides a vector decided by another process at line 7. CB_IC-Validity follows from the previous item.
- p_i decides at line 10. In that case $W = [\top, \dots, \top]$, i.e., no process has suggested a decision vector. Hence, for any p_j, $P(V_j)$ was false. Combining this with the $\mathrm{T}_{C \to P}$ property, we get that the input vector J does not belong to $\mathcal{C}_{f_c,f_e}$, and CB_IC-Validity trivially follows. $\square_{Lemma\ 1}$

Lemma 2. CB_IC-Termination. *If (1) $J \in \mathcal{C}_{f_c,f_e}$ and no more than f_c processes crash, or (2.a) no process crashes, or (2.b) a process decides, then every crash-correct process decides.*

Proof We consider the three cases separately.

- Let us assume that $J \in \mathcal{C}_{f_c,f_e}$ and no more than f_c processes crash.
  Let p_i be a crash-correct process. Let us first observe that, as at most f_c processes crash, p_i cannot block forever at line 2. Moreover, let V_i the local view obtained by p_i. As $J \in \mathcal{C}_{f_c,f_e}$, $V_i \leq J$ and $\#_\bot(V_i) \leq f_c$, we have $V_i \in \mathcal{C}_{f_c,f_e}$, i.e., $\exists D \in C$ such that $V_i \in D_{f_c,f_e}$. It then follows from the $\mathrm{T}_{C \to P}$ property that $P(V_i)$ holds. Consequently, $w_i \,/\!\!=\top$ and p_i decides at line 7.
- Let us assume that no process crashes and $P(V_i)$ holds for no process p_i. (Hence $J \notin \mathcal{C}_{f_c,f_e}$; otherwise, the previous item would apply.) As there is no crash, no process blocks forever at line 2. Moreover, as $P(V_i)$ holds for no process p_i and no process crashes, W becomes eventually equal to $[\top, \dots, \top]$. Consequently, no process blocks forever at line 8, and each process executes line 10 and terminates.
- Let us assume that some process (p_j) decides. In that case, as p_j executed line 2, we conclude that at least $(n - f_c)$ processes have deposited the value they propose into V, and consequently, no crash-correct process can block forever at line 2.
  Let us consider the case where p_j decides at line 7. Let p_i be a crash-correct process that does not decide at line 7. As no crash-correct process p_i blocks forever at line 2, p_i benefits from the vector deposited by p_j into $W[j]$ to decide at line 9.
  Let us consider the case where p_j decides at line 9. In that case, some process p_k deposited a vector into $W[k]$. As we have seen just previously, all crash-correct processes decide.

Let us finally consider the case where p_j decides at line 10. Then $W = [\top, \dots, \top]$, and consequently, no process decided at line 7 or 9. But, as $W = [\top, \dots, \top]$, any crash-correct process eventually exits line 8 and then decides at line 10.

$\square_{Lemma\ 2}$

The following corollary follows from the proof of the previous theorem.

Corollary 1. *Either all processes that decide do it at lines 7/9, or at line 10.*

Lemma 3. CB_IC-Agreement. *No two different vectors are decided.*

Proof Let us consider two processes p_i and p_j that decide. By Corollary 1, there are two cases. Moreover, if a process decides at line 9, it decides a vector decided by another process at line 7. So, in the following we only consider decisions at line 7 or 10.

– Both p_i and p_j decide at line 7. In that case, their local views V_i and V_j are such that $P(V_i)$ and $P(V_j)$ hold. Since the snapshot invocations can be ordered, these local views are ordered. Without loss of generality let us consider $V_i \leq V_j$. As $V_i \leq V_j$ and both $P(V_i)$ and $P(V_j)$ hold, we conclude from the $\mathrm{A}_{P\to S}$ property that $S(V_i) = S(V_j)$.
– Both p_i and p_j decide at line 10. In that case, it follows from the protocol text that they decide the same vector (made up of the n proposed values).

$\square_{Lemma\ 3}$

3.5 Characterizing the Interactive Consistency Conditions

A Characterization

In this section we prove the opposite of Theorem 1: if there is a (f_c, f_e)-fault tolerant protocol that solves CB_IC for C, then C must be (f_c, f_e)-acceptable. The proof extends ideas initially proposed in [9,29] for $f = 1$. We start by establishing a bridge from legality to acceptability.

Lemma 4. *An (f_c, f_e)-legal condition is (f_c, f_e)-acceptable.*

Proof Let C be an (f_c, f_e)-legal condition. We show that there are a predicate P and a function S satisfying the properties $\mathrm{T}_{C\to P}$, $\mathrm{A}_{P\to S}$ and $\mathrm{V}_{P\to S}$ defined in Section 3.3.

As C is (f_c, f_e)-legal, for any two distinct input vectors $I1$ and $I2$ we have $d(I1, I2) \geq 2f_e + f_c + 1$, from which we conclude that $\mathcal{C}_{f_c,f_e}$ is made up of non-intersecting spheres, each centered at a vector I of C. Let us define P and S as follows:

– $P(J)$ holds iff J belongs to a sphere (i.e., $\exists I \in C : \ J \in I_{f_c,f_e}$),
– $S(J)$ outputs the center I of the sphere to which J belongs.

The properties $\mathrm{T}_{C\to P}$ and $\mathrm{V}_{P\to S}$ are trivially satisfied by these definitions. Let us consider the property $\mathrm{A}_{P\to S}$. If $P(J1)$ holds, $J1$ belongs to a sphere centered at $I1 \in C$ (i.e., $J1 \in I1_{f_c,f_e}$). Moreover, due to definition of S, we have $S(J1) = I1$. Similarly for $J2$, if $P(J2)$ holds, $J2$ belongs to a sphere centered at $I2 \in C$ (i.e., $J2 \in I2_{f_c,f_e}$) and $S(J2) = I2$.

If $I1 \neq I2$, we have $d(I1, I2) \geq 2f_e + f_c + 1$ from the legality definition. It follows that there is at least one non-$\perp$ entry in which $J1$ and $J2$ differ. Consequently, in that case, we cannot have $J1 \leq J2$.

Let us now consider the additional assumption stated in $\mathrm{A}_{P\to S}$, namely, $J1 \leq J2$. From the previous observation, we conclude that, if $P(J1)$ and $P(J2)$ hold and $J1 \leq J2$, then $J1$ and $J2$ belong to the same sphere, and consequently have the same center I. Hence, $S(J1) = S(J2) = I$. $\square_{Lemma\ 4}$

Lemma 5. *If there is an (f_c, f_e)-fault tolerant protocol that solves* CB_IC *for C, then C must be (f_c, f_e)-legal.* (Proof in [24].)

We can now prove our main result.

Theorem 2. *The* CB_IC *problem for a condition C is (f_c, f_e)-fault tolerant solvable iff C is (f_c, f_e)-legal.*

Proof By Lemma 5 if the CB_IC problem for a condition C is (f_c, f_e)-fault tolerant solvable then C is (f_c, f_e)-legal. By Lemma 4 C is (f_c, f_e)-acceptable. By Theorem 1 the CB_IC problem for C is (f_c, f_e)-fault tolerant solvable. $\square_{Theorem\ 2}$

Correspondence with Error-Correcting Codes

A one-to-one correspondence. Consider an error-correcting code (ECC) problem where a sender wants to reliably send words to a receiver through an unreliable channel. Each word is represented by a sequence of k digits from an alphabet $\mathcal{A}$, and is called a *codeword*. A *code* consists of a set of codewords. The channel can erase or alter digits. The problem is to design a code that allows the receiver to recover the word sent from the word it receives (the received word can contain erased digits[1] and modified digits). We assume all codewords are of the same length, n. The ECC theory has been widely studied and has applications in many diverse branches of mathematics and engineering (see any textbook, e.g., [4]).

Although its goal is different, the CB_IC problem can actually be associated in a one-to-one correspondence with the ECC problem. More precisely, we have the following correspondences:

- alphabet $\mathcal{A}$/set of values $\mathcal{V}$,
- codeword/ vector of the condition,
- codeword length/number of processes (n),

[1] An erasure occurs when the received digit does not belong to the alphabet $\mathcal{A}$. Such a digit is replaced by a default value ($\perp$).

– code/condition,
– erasure/process crash (upper bounded by f_c),
– alteration/erroneous proposal (upper bounded by f_e),
– word received/proposed input vector,
– decoding/deciding.

On the necessary and sufficient condition. We have stated in Theorem 2 that the CB_IC problem is (f_c, f_e)-fault tolerant solvable for a condition C iff $\forall I1, I2 \in C$: $(I1 \not\equiv I2) \Rightarrow d(I1, I2) \geq 2f_e + f_c + 1$. The "corresponding" code theory theorem (whose proof appears in any texbook on error-correcting codes, e.g., Theorem 5.17 in [4], pages 96-97) is stated as follows:

> "A code C is t *error/e erasure* decoding iff its minimal Hamming distance is $\geq 2t + e + 1$."

In this sense, CB_IC and ECC are equivalent problems:

Theorem 3. *The* CB_IC *problem is* (f_c, f_e)*-fault tolerant solvable for a condition* C *iff* C *is* f_e error/f_c erasure *decoding.*

4 A Coding Theory-Based Approach to Consensus

An input vector can actually be seen as a codeword made up of n digits (one per process) that encode the decision value. This observation [17] leads to another way to characterize the set of conditions that allow to solve the consensus problem. The discussion that follows uses the message passing model. As shown in [17], the results hold also in the shared memory model. So, similarly to [23], the initial input values of all processes is seen as a vector of $\mathcal{V}^n$ (the set of all possible input vectors). We say that an *f-fault tolerant protocol solves the consensus problem for a given condition C* if it guarantees the following properties[2]:

– CB_C-Validity. If the input vector belongs to the condition C, then a decided value is a proposed value.
– CB_C-Agreement. If the input vector belongs to the condition C, then no two different values are decided.
– CB_C-Guaranteed Termination. If at most f processes fail and the input vector belong to C, then every correct process p_i eventually decides.

For Byzantine failures we use the following definition of validity and agreement:

– CB_BC-Validity. If all input values are v, then only v can be decided by a correct process.
– CB_BC-Agreement. If the input vector belongs to the condition C, then no two different values are decided by correct processes.

[2] Notice that this condition-based definition of the consensus problem is weaker than the one that we introduced in [23]. The definition in [23] requires validity and agreement to hold even when the input vector does not belong to the condition, (i.e., C-Validity and C-Agreement), and termination to additionally hold whenever there are no failures or a process decides. Section 4.2 discusses these issues.

4.1 Characterizing the Input Vectors with Codes

We define the initial configuration c of the system as *bi-valent* if there are two executions σ_1 and σ_2 that start with c such that the decision value in σ_1 is v_1, the decision value in σ_2 is v_2, and $v_1 \neq v_2$. Otherwise, the configuration is *univalent.*

In our characterization, each allowed input vector must always lead the system to the same decision value. In other words, the initial configuration of the system is not allowed to be bi-valent. Thus, we can treat the set of allowed input vectors as codewords coding the possible decision values. Clearly, in the case of consensus, since the decision value has to be unique, the code maps words from $\mathcal{V}^n$ to values in $\mathcal{V}$. Also, due to the validity requirement of consensus, we must limit ourselves to codes in which at least one of the digits of every codeword corresponding to a value $v \in \mathcal{V}$ has to be v. Thus we have:

Definition 3. *A d-*admissible code *is a mapping $C : \mathcal{V}^n \to \mathcal{V}$ such that the Hamming distance of every two codewords coding different values in $\mathcal{V}$ is at least d and at least one of the digits in each codeword mapped to a value $v \in \mathcal{V}$ is v.*

Solving Consensus with *d*-Admissible Codes

A simple generic protocol for solving the consensus problem using d-admissible codes in both the crash failure and Byzantine failure models is presented in Figure 2 (this protocol combines protocols described in [17,23]). V_i is the vector of initial values heard so far by process p_i; *code* is a set of codewords defining the allowed input vectors.

```
Function MP_Consensus($v_i$)

(1)  $\forall j$ : send VAL1($v_i, i$) to $p_j$;
(2)  wait until (at least $(n - f)$ VAL1 msgs have been delivered);
(3)  $\forall j$ : do if VAL1($v_j, j$) has been delivered then $V_i[j] \leftarrow v_j$
(4)                                              else $V_i[j] \leftarrow \perp$ endif
(5)       enddo;
(6)  $w_i \leftarrow$ match ($V_i$,code);
(7)  if $w_i \neq \perp$ then return($w$) endif
```

Fig. 2. A Message-Passing Consensus Protocol

For the crash failures model with at most $f = f_c$ failures, it is sufficient to use an $(f_c + 1)$-admissible code, while for the Byzantine failures model with at most $f = f_b$ failures we have to use a $(2f_b + 1)$-admissible code. Also, the protocol uses a subroutine called `match` to check whether the digits received so far can be matched to any codeword. It returns $\perp$ if no matching was found; otherwise, it returns the value of the decoded word corresponding to the codeword matched.

More precisely, each digit that was not received from some process is treated as the special value $\perp$ in the vector of received digits. Then, all `match` has to do is to check if the distance of this vector is at most f from some legal codeword. If it is, `match` returns the value mapped to by this codeword. Otherwise, `match` returns $\perp$. Note that by the specific codes that we use, the same rule applies in both benign and Byzantine failures, and in both cases, if there are at most f failures, we are guaranteed to find such a codeword. Also, it is possible that there would be several possible codewords to chose from, but in this case they all have to be mapped to the same value, so any of them can be picked. The `match` routine is the equivalent of the predicate P and function S introduced in [23]. The exact implementation of the `match` routine is outside the scope of this paper. Here we simply rely on coding theory to guarantee that it exists.

It is not hard to show [17] that the previous protocol satisfies validity, agreement, and termination when the input vectors are indeed codewords of some $(f_c + 1)$-admissible (or $(2f_b + 1)$-admissible) code. Section 4.2 discusses the implications of satisfying validity and agreement when the input vectors are not always codewords, and termination when the input vectors are not codewords but there are no failures.

Necessity of *d*-Admissibility for Solving Consensus

The consensus problem considered in Theorem 4 (resp. Theorem 5) is defined by the following three properties: CB_C-Agreement (resp.CB_BC-Agreement), CB_C-Guaranteed Termination and CB_C-Validity (resp. CB_BC-Validity).

Theorem 4. *If a condition C allows to solve consensus in the crash failure model (with at most f_c failures), then C consists of codewords of an $(f_c + 1)$-admissible code.*

Proof To prove the theorem, we first point the reader to the proof of the consensus impossibility result as stated in [3]. In that proof, it was shown that if there is a bi-valent initial configuration, then consensus cannot be solved. That proof is for the case of a single failure, but the case of f_c failures is completely analogous. So, all that we have to show is that if the initial input vectors allowed by the condition are not words of an $(f_c + 1)$-admissible code, then there has to be a bi-valent initial configuration.

Assume, by way of contradiction, that C does not correspond to an $(f_c + 1)$-admissible code and there is no bi-valent initial configuration. Thus, there are two allowed univalent initial configurations c_1 and c_2 that differ in less than $f_c + 1$ processes, yet each one leads to a different value v_1 and v_2 respectively. Denote by $P' = p_{i_1}, \ldots, p_{i_k}$ $(k \leq f_c)$ the processes that differ between c_1 and c_2. Hence, there is an execution σ_1 of the protocol that starts at c_1 in which all processes in P' fail before managing to take any action. All other processes must decide in σ_1 on value v_1 without receiving any message from any process in P'. Let p_j be one of the processes that decides in σ_1 on v_1, HP_{p_j} be the history prefix of p_j at the moment it decides, and $CH_{p_j}(\sigma_1, HP_{p_j})$ be its causal history at that point.

Since the network latency is unbounded, we can create another execution σ_2 that starts in c_2, no process fails during σ_2, but $CH_{p_j}(\sigma_1, HP_{p_j})$ is also a causal history of p_j in σ_2. Given the determinism of p_j, it must also decide in σ_2 on the same value v_1. Since the protocol is assumed to solve consensus, all processes must decide v_1. Thus, either c_2 leads to v_1, or c_2 is bi-valent. A contradiction. $\square_{Theorem}$ 4

Theorem 5. *If a condition C allows to solve consensus in the Byzantine failure model (with at most f_b failures), then C consists of codewords of a $(2f_b + 1)$-admissible code.*

Proof To prove the theorem, we first note that the proof in [3] that consensus is not solvable if there is an initial bi-valent configuration is also valid for the Byzantine case. Thus, all that we have to show is that if the initial input vectors allowed by the condition are not words of a $(2f_b + 1)$-admissible code, then there has to be a bi-valent initial configuration.

Assume by way of contradiction that C does not correspond to a $(2f_b + 1)$-admissible code and there is no bi-valent initial configuration. Thus, there are two allowed univalent initial configurations c_1 and c_2 that differ in less than $2f_b + 1$ processes, yet each one leads to a different value v_1 and v_2 respectively. We can divide the set of processes whose initial state is different in c_1 and c_2 into two subsets, P' and P'' such that $|P'| \leq |P''| \leq f$. Due to termination, there is an execution σ_1 of the protocol that starts at c_1 in which all processes in P' crash before managing to take any action. All other processes must decide in σ_1 on value v_1 without receiving any message from any process in P'. Let p_j be one of the processes in $N \setminus P''$ that decides in σ_1 on v_1, HP_{p_j} be the history prefix of p_j at the moment it decides, and $CH_{p_j}(\sigma_1, HP_{p_j})$ be its causal history at that point.

Since the network latency is unbounded, we can create the following execution σ_2 that starts in c_2. In σ_2, all processes in P'' suffer the Byzantine failure that makes them behave as if their initial configuration was as in c_1, but otherwise obey the protocol. No process crashes during σ_2. Due to the unbounded message latency, all messages of processes in P' are delayed enough so that $CH_{p_j}(\sigma_1, HP_{p_j})$ is also a causal history of p_j in σ_2. Given the determinism of p_j, it must also decide in σ_2 on the same value v_1. Since the protocol is assumed to solve consensus, all correct processes must decide v_1. Thus, either c_2 leads to v_1, or c_2 is bi-valent. A contradiction. $\square_{Theorem}$ 5

4.2 Strict Condition-Based Consensus

In order to weaken the dependency of the possible solutions on whether the input vectors are indeed codewords, we can make any of the validity, agreement, and termination requirements more strict. Here we discuss the consequences of doing so.

Agreement

In order to ensure that agreement holds even when the input vectors are not codewords (i.e., C-Agreement), it is possible to augment the protocol described in Figure 2 with additional checks that verify that all decided values are the same. The modified protocol is depicted in Figure 3. It guarantees agreement whenever $f = f_c < n/2$ in the benign failures model and when $f = f_b < n/3$ in the Byzantine model. The protocol also guarantees CB-C-Validity and CB-C-Guaranteed Termination.

```
Function MP_Consensus($v_i$)

(1)  $\forall j$ : send VAL1($v_i, i$) to $p_j$;
(2)  wait until (at least $(n-f)$ VAL1 msgs have been delivered);
(3)  $\forall j$ : do if VAL1($v_j, j$) has been delivered then $V_i[j] \leftarrow v_j$
(4)                                               else $V_i[j] \leftarrow \perp$ endif
(5)      enddo;
(6)  $w_i \leftarrow$ match ($V_i$,code);
(7)  $\forall j$ : send VAL2($w_i, i$) to $p_j$;
(8)  repeat  wait for a new VAL2($w_j, j$) message;
(9)          if VAL2($w, -$) with the same non-$\perp$ value $w$ has
(10)           been received from at least $(n-f)$ processes
(11)           then return($w$)
(12)         endif
(13) endrepeat
```

Fig. 3. An Always Safe Message-Passing Consensus Protocol

Validity

The previous protocol (Figure 3) might not satisfy the C-Validity property when the input vector is not a codeword of an admissible code (it only satisfies CB_C-Validity). A slightly stronger definition of admissibility allows to overcome this problem, namely:

Definition 4. *A* strongly d-admissible code *is a mapping* $C : \mathcal{V}^n \to \mathcal{V}$ *such that the Hamming distance of every two codewords coding different values in* $\mathcal{V}$ *is at least* d *and at least* d *of the digits in each codeword mapped to a value* $v \in \mathcal{V}$ *are* v.

We say that an *f-fault tolerant protocol solves the* strict *consensus problem for a given condition* C if it guarantees C-Validity, C-Agreement, and CB-C-Guaranteed Termination. Clearly, the protocol of Figure 3 solves strict consensus for strongly $(f+1)$-admissible codes.

Theorem 6. *A condition C allows to solve* strict *consensus in the crash failure model (with at most f_c failures), iff C consists of codewords of a* strongly *(f_c+1)-admissible code.*

Proof First, note that the proof of Theorem 4 holds here as well. The only thing we need to show is that C-Validity cannot be guaranteed unless the code satisfies the property that each word is mapped to a value that appears in at least $f_c + 1$ of its digits. Assume by way of contradiction that there is a protocol $\mathcal{P}$ that solves strict consensus for a condition C that includes an allowed input vector V in which no value appears more than $k \leq f_c$ times. In other words, every execution of $\mathcal{P}$ that starts with V has to terminate.

As discussed before, since $\mathcal{P}$ solves consensus for C, V must be an initial univalent configuration. Consider an execution of $\mathcal{P}$ that starts with V and decides some value v. Thus, every execution of $\mathcal{P}$ that starts with V must decide v. Since v only appears $k \leq f_c$ times in V, the execution E in which all processes whose initial value is v crash before sending any message must also terminate with a decision value v. Denote the set of corresponding processes by S. Therefore, there exists an execution E' in which the input vector is the same as V except for all processes in S for which the input value is different, and during E' all processes in S crash immediately. For processes outside S, E' is indistinguishable from E, and therefor E' also terminates with a decision value v. However, this violates C-Validity. $\square_{Theorem}$ 6

Note that instead of using strongly d-admissible codes, we could use a weaker definition of validity that only requires it to hold if either all initial values are the same, or when there are no failures. Such a definition is used in any case for the Byzantine failure model.

It is shown in [17] that strongly $(f + 1)$-admissible codes are the same as f-acceptable codes in [23]. In particular, Condition C_1 in [23] requires that the most popular value in a vector in C_1 appear at least $f + 1$ times more than the second most popular value in the same vector. Clearly, any two vectors that lead to different decision values and obey this condition must differ in at least $f + 1$ places, and thus the Hamming distance of the code is $f + 1$.

Condition C_2 in [23] requires that the largest value in a vector in C_2 appear at least $f + 1$ times. For any condition defined on values from some range $\mathcal{V}$, any vector mapped to a value $v \in \mathcal{V}$ must include at least $f + 1$ v entries and no entries larger than v. Consider two vectors V_1 and V_2 leading to different decision values v and u such that $v > u$. Thus, V_1 must include at least $f + 1$ v entries, while V_2 does not include any v entries, which means that they differ in at least $f + 1$ entries. In other words, the Hamming distance of the code is $f + 1$.

Termination

To guarantee termination in the crash failure model with $f_c < n/2$, both (1) when the input vector is a codeword and (2) when the input vector is not a codeword and there are no failures or a process decides, it is possible to use the protocols described in [23]. At present, we have no solution for the Byzantine model.

4.3 k-Set Consensus

The augmented definition of k-set consensus when there are conditions on the input vectors is:

- CB_K-Validity. If the input vector belongs to the condition C, then a decided value is a proposed value.
- CB_K-Agreement. If the input vector belongs to the condition C, then at most k distinct values are decided.
- CB_K-Guaranteed Termination. If at most f processes fail and the input vector belongs to the condition C, then eventually every correct process p_i decides.

For the case of Byzantine faults we use the following definitions of validity and agreement:

- CB_KB-Validity. If the initial value of all processes is the same, then every correct process that decides has to decide on this value.
- CB_KB-Agreement. If the input vector belongs to the condition C, then at most k distinct values are decided by the correct processes.

Next, we define (d, k)-admissible codes:

Definition 5. *A (d, k)-admissible code is a mapping $C : \mathcal{V}^n \to \mathcal{V}$ such that: (a) for every codeword w in C, all codewords whose Hamming distance from w is less than d are mapped to at most k different values, and (b) at least one of the digits in each codeword mapped to a value $v \in \mathcal{V}$ is v.*

Given the above definition, we claim that k-set consensus can be solved if the input vectors are codewords of a $(f_c + 1, k)$-admissible code in the crash failures model, and $(2f_b + 1, k)$-admissible code in the Byzantine model. Note that by slightly changing the behavior of the `match` routine in the protocol in Figure 2, we can use the protocol without any additional modifications to solve k-set agreement for the above codes. The only difference is that now `match` checks whether the Hamming distance of the vector of received digits from any codeword is at most f_c (f_b). If the answer is yes, `match` returns the value pointed to by the closest of these codewords, breaking symmetry arbitrarily. Otherwise, `match` returns $\perp$.

5 Practical Implications of the Results

5.1 Benefiting from Error-Correcting Code Theory

Following the results of Sections 3 and 4, we can use known *t error/e erasure* correcting codes to define conditions suited to the CB_IC and *consensus* problems. This is illustrated below with a simple example using a linear code. Interestingly, linear codes can be composed. An additional benefit of using coding theory is that some codes have been proved to be maximal. Using such a code gives a condition that contain as many input vectors as possible.

From a code ... Let us consider the following linear code C, namely, the binary $[n, M, d]$-code where $n = 6$ is the length of a codeword, $M = 2^k = 8$ is the number of codewords (k being the length of the words to be encoded) and $d = 3$ is the code minimal Hamming distance. As $d = 3$, C can detect and correct one error. The set C of codewords can be obtained from a generator matrix G made up of k linearly independent size n vectors. Considering a particular generator matrix G, we get the following set C of 8 codewords:

0 0 0 0 0 0	1 1 1 0 0 0	0 1 0 1 0 1	1 0 1 1 0 1
1 0 0 1 1 0	0 1 1 1 1 0	1 1 0 0 1 1	0 0 1 0 1 1

These codewords can also be defined from a check matrix A. This matrix is obtained from G: it is such that $A\ G^T = 0$ (G^T is the transpose of G, and 0 is a $k \times k$ zero matrix). We have:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Let w be a received word. It is a codeword if its syndrome is equal to the 0 vector (the syndrome of a vector w is the value Aw^T, where w^T is the transpose of w). When $A\ w^T\ /= 0$, the syndrome value defines the altered position of w.

... To a condition. Let us now consider a system made up of $n = 6$ processes. The previous 8 vectors define a condition C on the vocabulary $\mathcal{V} = \{0, 1\}$. Due to Definition 2 and Theorem 2, C can cope with $f_c = 2$ process crashes and $f_e = 0$ erroneous proposals (or alternatively, $f_c = 0$ and $f_e = 1$). The matrix A provides a simple way to define the condition C for the CB_IC problem:

$$C = \{I \text{ such that } \mathit{syndrome}(I) = 0\}$$

where $\mathit{syndrome}(I) = A\ I^T$. It is possible to show that the following acceptability parameters (predicate P and function S) can be associated with such a condition C:

$$P(J) = \exists I : \text{ such that } J \in I_{f_c,f_e}\ \wedge\ \mathit{syndrome}(I) = 0,$$
$$S(J) = I \text{ such that } J \in I_{f_c,f_e}\ \wedge\ \mathit{syndrome}(I) = 0.$$

The use of the *syndrome* function allows for an efficient determination of the input vectors I defining a legal condition. As the CB_IC problem directly considers input vectors of size n (each corresponding to a codeword), the generator matrix G is useless for this problem.

5.2 Benefiting from Distributed Computing Results

Let a code C, $|C| > 1$, be (f_c, f_e)-*perfect* if the spheres whose centers are the vectors I of C define a partition of $\mathcal{V}^n_{f_c}$, i.e.,

$$\forall I1, I2 \in C : I1 \neq I2 : \; I1_{f_c,f_e} \cap I1_{f_c,f_e} = \emptyset \qquad \text{and} \qquad \mathcal{V}^n_{f_c} = \bigcup_{I \in C} I_{f_c,f_e}.$$

When $f_c = 0$, the notion of (f_c, f_e)-*perfect* code is the coding theory notion of *perfect* code. Interestingly, perfect codes are known. (They are rare, and have the property not to correctly decode a received word with more than f_e errors.) The following is a simple example of using distributed computing to prove a result in coding theory. It states that code perfection does not exist when a code system has to cope with digit erasures.

Theorem 7. *There is no (f_c, f_e)-perfect code for $f_c \geq 1$, $f_e < n$.*

Proof Let us assume that such an (f_c, f_e)-perfect code C does exist. Then, due to the definition of (f_c, f_e)-perfection, we have $\mathcal{V}^n_{f_c} = \bigcup_{I \in C} I_{f_c,f_e}$, i.e., all the possible input vectors are included. As C is (f_c, f_e)-perfect, no two spheres intersect. Consequently, C is (f_c, f_e)-legal, i.e., (f_c, f_e)-acceptable. This means that there is a CB_IC protocol that always terminates when the number of crashes is $\leq f_c$. Consequently, in every execution all decided vectors are equal. Moreover, at least two different vectors are decided, since the code is non-trivial. This is a version of consensus that is known to be unsolvable in the presence of crashes (an implication of FLP [15], e.g. [22]). It follows that C is not $(f_c, 0)$-perfect. $\Box_{Theorem}$ 7

Another trivial result that stems from this work is that error correcting data, e.g., parity bits (and in general `xor`), cannot be computed in asynchronous environments prone to failures.

5.3 Agreement in Mixed Environments

The practical implication of the previous coding-based characterization is the ability to solve consensus in mixed environments. That is, assume that a set of n processes is split into clusters, where in each cluster the communication is synchronous enough so that consensus can be solved, but between clusters the system is asynchronous. Clusters can correspond to different LANs, or a single large LAN can be arbitrarily divided into several clusters for scalability purposes. With such a division into clusters, it is possible to have all nodes of a single cluster initially decide on one value, and use that value as their input value to the global consensus problem, which will be run among all processes using the protocol similar to the one described in Figure 2. If the size of the smallest cluster is at least $f_c + 1$ (resp., $2f_b + 1$), we can solve consensus in the global system despite f_c crash failures (resp., f_b Byzantine failures).

A shortcoming of this approach is that if clusters become disconnected, the previous protocol can block until they reconnect again. Another shortcoming of the above scheme is that it requires a high degree of redundancy in the system. However, if we look at error correcting codes, for both erasure and alteration errors, there are more efficient codes. For example, parity can be used to overcome one digit erasure with only one extra digit, while Hamming code can correct one digit flip with an overhead of $\log(n)$ digits. But, in both cases, some digits in each codeword depend on many other digits in the same word. Practically speaking, given a code, if some digit b depends on the values of some other digits $b_1, \ldots, b_l$, it indicates that process b needs synchronous communication links with $b_1, \ldots, b_l$. Thus, it would be interesting to find codes that present a good tradeoff between the number of digits each digit depends on, and the digit overhead for error correction. That is, small overhead implies the ability to solve consensus with small hardware redundancy, while low dependency between digits means that it can be applied more easily to real settings, since it requires weaker synchrony assumptions. Looking at linear codes might be a good direction for this [4,7].

Let us notice that Pfitzmann and Waidner have shown how to solve Byzantine agreement for any number of faults in the presence of a reliable and secure multicast during a precomputation phase [31]. Also, Fitzi and Maurer showed how to obtain Global Broadcast in the presence of up to $n/2$ Byzantine failures based on a Local Broadcast service [16]. However, none of these works draws any relation from agreement to error correction.

5.4 Cluster-Based Failure Detectors

The above discussion indicates that it is possible to solve consensus despite a *small number of failures* using failure detectors that provide the accuracy and completeness properties of $\diamond\mathcal{W}$ only among members of clusters. Such failure detectors need not guarantee anything about failure suspicions of processes outside the cluster. Formally, we assume that processes are divided into non-overlapping clusters, and augment the definitions of accuracy and completeness given in [11] as follows:

- Strong c-Completeness. Eventually, every process that fails is permanently suspected by every non-faulty process in the same cluster.
- Weak c-Completeness. Eventually, every failed process is permanently suspected by some non-faulty process in the same cluster.
- Eventual Strong c-Accuracy. There is a time after which no non-faulty process is suspected by any non-faulty process in the same cluster.
- Eventual Weak c-Accuracy. There is a time after which some non-faulty process is not suspected by any non-faulty process in the same cluster.

As discussed in [11], guaranteeing one of these properties is trivial. The difficult problem (impossible in completely asynchronous systems) is guaranteeing a combination of one of the accuracy requirements with one of the completeness requirements. A failure detector belongs to the class c-$\diamond\mathcal{W}$ if it guarantees Weak c-Completeness and Eventual Weak c-Accuracy. Similarly, a failure detector belongs to the class c-$\diamond\mathcal{S}$ if it guarantees Strong c-Completeness and Eventual Weak c-Accuracy.

Clearly, it is possible to simulate a failure detector in c-$\diamond\mathcal{S}$ from a failure detector in c-$\diamond\mathcal{W}$ by running within each cluster the simulation given in [11] for simulating $\Diamond\mathcal{S}$ from $\Diamond\mathcal{W}$. It is thus possible to solve consensus among members of the same cluster using c-$\Diamond\mathcal{S}$ and any of the $\Diamond\mathcal{S}$-based consensus protocols (e.g., [11,19,20,28,33]). Similarly to the discussion above, each process can use the decision value of its cluster as its input value in the global consensus protocol of Figure 2. We call this the *direct cluster based approach.*

On the other hand, it is easy to derive a failure detector in $\Diamond\mathcal{W}$ from a failure detector in c-$\Diamond\mathcal{W}$. Specifically, assume that each process is equipped with a failure detector FD from the class c-$\Diamond\mathcal{W}$.

Theorem 8. *A failure detector FD' that adopts the failure suspicions of FD for processes inside the cluster, but never suspects any process outside the cluster is in $\Diamond\mathcal{W}$.*

Proof Note that $FD' \in c$-$\Diamond\mathcal{W}$, since it behaves the same as FD for processes inside the same cluster. Clearly, Weak c-Completeness is stronger than Weak Completeness. This is because the latter only requires that eventually, every failed process be permanently suspected by some non-faulty process, but different failed processes can be suspected by different non-faulty processes.

Also, for each cluster, FD guarantees that there it at least one non-faulty process that is not suspected by any non-faulty process within the cluster. Moreover, by the construction of FD', this process is not suspected by any process outside the cluster, and thus FD' is in $\Diamond\mathcal{W}$. $\Box_{Theorem}$ 8

Consequently, it is possible to employ Theorem 8 to simulate a failure detector in $\Diamond\mathcal{W}$, and use it to solve consensus with any of the previously cited protocols. However, we argue that the direct cluster based approach is more efficient and scalable. That is, the direct cluster based approach only requires failure detection (heartbeats) among nodes of the same cluster. Specifically, there is no need for long haul failure detection, and heartbeats are exchanged only among a small set of close nodes. In contrast, the simulation of $\Diamond\mathcal{S}$ from $\Diamond\mathcal{W}$ given [11] requires many long haul message exchanges. Moreover, with the direct cluster based approach, all rounds of $\Diamond\mathcal{W}$-based protocols are executed between a small set of well connected processes. Given that consensus can be used as a building block for solving other problems in distributed computing this can serve as a basis for a scalable solution to these problems as well.

As before, the downside of this scheme is that if a single cluster becomes disconnected from the rest of the network, this might prevent the global consensus from terminating until that cluster reconnects. Conversely, existing protocols for solving consensus (with respect to the entire set of nodes) that rely on $\Diamond\mathcal{S}$ can overcome up to $\lceil n/2 \rceil - 1$ crash failures.

6 Discussion

This paper has investigated principles that underlie distributed computing by establishing links between agreement problems and error correcting codes. The results that have been presented draw a correlation between crash failures in distributed computing and erasures in error correcting, and between value domain faults and Byzantine failures in distributed computing and digits corruptions in error correcting. In particular, it has been shown that condition-based interactive consistency and error correcting are two facets of the same problem. Similar conditions are shown to be sufficient and necessary for solving consensus. Yet, the conditions for consensus are on one hand less restrictive, since in consensus each decision value may be coded by more than one codeword. On the other hand, the conditions for consensus include a restriction on the occurrence of the decoded value in the digits of the corresponding codewords, which is due to the validity requirement of consensus. (This requirement is implicit in interactive consistency.) For k-set agreement, we only showed sufficient conditions, which are less restrictive due to the more relaxed agreement requirement of the problem. Showing necessary error-correcting based conditions for k-set agreement is still an open issue, although some advancements have been made in giving topological characterizations for such conditions [1,27]. Interestingly enough, the same protocol, instantiated with the appropriate conditions, can be used to solve all three problems [27].

The paper has also discussed some interesting tradeoffs related to whether the agreement, validity, and termination requirements should be preserved only when the input belongs to the condition. Specifically, we showed that in order to always obtain validity (i.e., even when the input vector does not belong to the condition), it is necessary to use more restrictive conditions, while to obtain always agreement and also termination when there are no failures, we need to enrich the basic protocols. We have also discussed the implications of our results in terms of applying limited synchrony technologies like wormholes [36] and cluster based failure detectors in a clever way.

Another interesting result that comes from this paper is that interactive consistency is harder than consensus in the sense that it requires more restrictive conditions. This echos the known result that interactive consistency requires perfect failure detectors [18], while consensus can be solved with unreliable failure detectors (this suggests that it might not be a good idea to solve consensus via interactive consistency). It would be interesting to find a more direct linkage between the strength of conditions and failure detectors. In particular, if one can unify this with topological characterization of such conditions, it might enable viewing failure detectors as topological tweaks on the environment that enable agreement to be solved.

Finding a linkage between coding theory and agreement problems in distributed computing seems to be both an intellectually challenging task and a practically relevant aim. Coding theory is an area that has been extensively studied. By applying results from coding theory, it might be possible to find simpler proofs to existing results, and ideally, even to obtain new results in distributed computing. At the same time, the paper has shown that it is possible to draw simple proofs for results in error correcting codes by reduction to agreement problems.

Our work leaves a few additional interesting open problems. For example, can the results be generalized under a unified framework for general agreement problems? Is there a distributed computing problem that is a counterpart of error detection codes similarly to the fact that CB_IC is the counterpart of error correction? More generally, given a condition C, which agreement problem does C allow to solve? And at what cost (see [25] for a related result in the case of consensus)?

References

1. Attiya H. and Avidor Z., Wait-Free n-Consensus When Inputs are Restricted. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, These proceedings.
2. Attiya H. and Rachman O., Atomic Snapshots in $O(n \log n)$ Operations. *SIAM Journal on Computing,* 27(2):319–340, 1998.
3. Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* McGraw-Hill, 451 pages, 1998.
4. Baylis J., *Error-Correcting Codes: a Mathematical Introduction.* Chapman & Hall Mathematics, 219 pages, 1998.
5. Ben-Or M., Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proc. 2nd ACM Symposium on Principles of Distributed Computing (PODC'83)*, pp. 27–30, Montréal, 1983.
6. Ben-Or M., Goldwasser S., and Wigderson A., Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *Proc. 20th ACM Symposium on Theory of Computing (STOC'88)*, pp. 1–10, 1988.
7. Berlekamp E.R., *Algebraic Coding Theory* (Second Edition). Aegean Park Press, 1984.
8. Berman P. and Garay J., Adaptability and the Usefulness of Hints. *6th European Symposium on Algorithms (ESA'98),* Springer-Verlag LNCS #1461, pp. 271–282, Venice (Italy), 1998.
9. Biran O., Moran S. and Zaks S., A Combinatorial Characterization of the Distributed 1-Solvable Tasks. *Journal of Algorithms*, 11:420–440, 1990.
10. Chaudhuri S., More *Choices* Allow More *Faults:* set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.
11. Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.
12. Dolev D., Dwork C. and Stockmeyer L., On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, 1987.
13. Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E., Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM*, 33(3):499–516, 1986.
14. Dwork C., Lynch N. and Stockmeyer L., Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
15. Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.
16. Fitzi M. and Maurer U., From Partial Consistency to Global Broadcast. *Proc. 32rd ACM Symposium on Theory of Computing (STOC'00)*, pp. 494–503, 2000.
17. Friedman R., A Simple Coding Theory-Based Characterization of Conditions for Solving Consensus. *Technical Report CS-2002-06 (Revised Version)*, Department of Computer Science, The Technion, Haifa, Israel, June 30, 2002.
18. Hélary J.-M., Hurfin M., Mostéfaoui A., Raynal M. and Tronel F., Computing Global Functions in Asynchronous Distributed Systems with Perfect Failure Detectors. *IEEE Trans. on Parallel and Distributed Systems*, 11(9):897–910, 2000.

19. Hurfin M., Mostefaoui A. and Raynal M., A Versatile Family of Consensus Protocols Based on Chandra-Toueg's Unreliable Failure Detectors. *IEEE Transactions on Computers*, 51(4):395–408, 2002.
20. Hurfin M. and Raynal M., A Simple and Fast Asynchronous Consensus Protocol Based on a Weak Failure Detector. *Distributed Computing*, 12(4):209–223, 1999.
21. Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.
22. Moses Y. and Rajsbaum S., The Unified Structure of Consensus: a Layered Analysis Approach. *Proc. 17th ACM Symp. on Principles of Distributed Computing (PODC'98)*, pp. 123–132, 1998. To appear *SIAM Journal on Computing.*
23. Mostefaoui A., Rajsbaum S. and Raynal M., Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Proc. 33rd ACM Symposium on Theory of Computing (STOC'01)*, pp. 153–162, Crete (Greece), 2001.
24. Mostefaoui A., Rajsbaum S. and Raynal M., Asynchronous Interactive Consistency and its Relation with Error-Correcting Codes. *Research Report #1455*, IRISA, University of Rennes, France, April 2002, 16 pages. (Also Brief Announcement in *PODC'02.*) http://www.irisa.fr/bibli/publi/pi/2002/1455/1455.html.
25. Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., A Hierarchy of Conditions for Consensus Solvability. *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, ACM Press pp. 151–160, Newport (RI), 2001.
26. Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Efficient Condition-Based Consensus. *8th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO'01)*, Carleton Univ. Press, pp. 275–291, Val de Nuria (Catalonia, Spain), 2001.
27. Mostefaoui A., Rajsbaum S., Raynal M. and Roy M., Condition-Based Protocols for Set Agreement Problems. *Proc. 16th Symposium on Distributed Computing (DISC'02)*, These proceedings.
28. Mostefaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int. Symposium on Distributed Computing (DISC'99)*, Bratislava (Slovaquia), Springer-Verlag LNCS 1693, pp. 49–63, 1999.
29. Moran S. and Wolfstahl Y., Extended Impossibility Results for Asynchronous Complete Networks. *Information Processing Letters,* 26:145–151, 1987.
30. Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.
31. Pfitzmann B. and Waidner M., Unconditional Byzantine Agreement for any Number of Faulty Processors. *Proc. of the 9th Int. Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag LNCS 577, pp. 339–350, 1992.
32. Powell D., Failures Mode Assumptions and Assumption Coverage. *Proc. 22th IEEE Fault-Tolerant Computing Symposium (FTCS'92)*, IEEE Society Press, pp. 386–395, Boston (MA), 1992.
33. Schiper A., Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10:194–157, 1997.
34. Taubenfeld G., Katz S. and Moran S., Impossibility Results in the Presence of Multiple Faulty Processes. *Information and Computation*, 113(2):173–198, 1994.
35. Taubenfeld G. and Moran S., Possibility and Impossibility Results in a Shared Memory Environment. *Acta Informatica*, 35:1–20, 1996.
36. Veríssimo P., Traveling Through Wormholes: Meeting the Grand Challenge of Distributed Systems. *Proc. Int. Workshop on Future Directions in Distributed Computing (FuDiCo)*, pp. 144–151, Bertinoro (Italy), June 2002.

On the Stability of Compositions of Universally Stable, Greedy Contention-Resolution Protocols

D. Koukopoulos[1], M. Mavronicolas[2], S. Nikoletseas[1], and P. Spirakis[1]⋆

[1] Department of Computer Engineering & Informatics, University of Patras and Computer Technology Institute (CTI), Riga Fereou 61, P. O. Box 1122, 261 10 Patras, Greece. {nikole,koukopou,spirakis}@cti.gr

[2] Department of Computer Science, University of Cyprus, 1678 Nicosia, Cyprus. mavronic@ucy.ac.cy

Abstract. A distinguishing feature of today's large-scale platforms for distributed computation and communication, such as the *Internet,* is their *heterogeneity,* predominantly manifested by the fact that a wide variety of *communication protocols* are simultaneously running over different distributed hosts. A fundamental question that naturally poses itself concerns the preservation (or loss) of important correctness and performance properties of the individual protocols when they are *composed* in a large network. In this work, we specifically address stability properties of greedy, contention-resolution protocols operating over a *packet-switched* communication network.

We focus on a basic adversarial model for packet arrival and path determination for which the time-averaged arrival rate of packets requiring a single edge is no more than 1. *Stability* requires that the number of packets in the system remains bounded, as the system runs for an arbitrarily long period of time. It is known that several commonly used contention-resolution protocols, such as LIS (*Longest-in-System*), SIS (*Shortest-in-System*), NTS (*Nearest-to-Source*), and FTG (*Furthest-to-Go*) are *universally stable* in this setting – that is, they are stable over all networks. We investigate the preservation of universal stability under compositions for these four greedy, contention-resolution protocols. We discover:

- The composition of any two protocols among SIS, NTS and FTG is universally stable.
- The composition of LIS with any of SIS, NTS and FTG is *not* universally stable: we provide interesting combinatorial constructions of networks over which the composition is unstable when the adversary's injection rate is at least 0.519.
- Through an involved combinatorial construction, we significantly improve the current state-of-the-art record for the adversary's injection rate that implies instability for FIFO protocol to 0.749. Since 0.519 is significantly below 0.749, this last result suggests that the potential for instability incurred by the composition of *two* universally stable protocols may be *worse* than that of some *single* protocol that is not universally stable.

⋆ This work has been partially supported by the IST Program of the European Union under contract numbers IST-1999-14186 (ALCOM-FT) and IST-2001-33116 (FLAGS).

1 Introduction

1.1 Motivation-Framework

Heterogeneous Networks. A key feature of contemporary large-scale platforms for distributed communication and computation, such as the *Internet,* is their *heterogeneity.* Heterogeneity comes around in many different flavors. For example, the specifics of how the computers in different parts of the network are connected (directly or indirectly) with each other, and the properties of the links that foster the interconnection, is difficult to characterize uniformly. Second, different traffic sources over the Internet (due to varying mechanisms for supporting different classes and qualities of service) result in a heterogeneous mix of traffic traces. Third but not least, although, conceptually, the Internet uses a unified set of protocols, in practice each protocol has been implemented with widely varying features (and of course bugs). (See the recent interesting article by Floyd and Paxson [7] for an extended discussion on the heterogeneity of Internet.) Thus, heterogeneity is a crucial feature that makes it difficult to model, verify and analyze the behavior of such large-scale communication networks.

Objectives. In this work, we embark on a study of the impact of heterogeneity of distributed systems on their correctness and performance properties. More specifically, we wish to pose the general question of which correctness and performance properties of individual, different modules of a distributed system are maintained and which are not when such modules are *composed* into a larger, heterogeneous distributed system. We choose, as a test-bed, the case of distinct *communication protocols* that are simultaneously running on different hosts in a distributed system. We ask, in particular, which (and how) stability properties of greedy, contention-resolution protocols operating over a *packet-switched* communication network are maintained under composition of such protocols.

Framework of Adversarial Queueing Theory. We consider a packet-switched communication network in which packets arrive dynamically at the nodes with predetermined paths, and they are routed at discrete time steps across the edges. We focus on a basic adversarial model for packet arrival and path determination that has been recently introduced in a pioneering work by Borodin *et al.* [3], under the name *Adversarial Queueing Theory.* Roughly speaking, this model views the time evolution of a packet-switched communication network as a game between an *adversary* and a *protocol.* At each time step, the adversary may inject a set of packets into some nodes. For each packet, the adversary specifies a simple path (including an *origin* and *destination*) that the packet must traverse; when the packet arrives to destination, it is absorbed by the system. When more than one packets wish to cross a queue at a given time step, a *contention-resolution* protocol is employed to resolve the conflict. A crucial parameter of the adversary is its *injection rate* r, where $0 < r < 1$. Among the packets that the adversary injects in any time interval I, at most $\lceil r|I| \rceil$ can have paths that contain any particular edge. Such a model allows for adversarial injection of packets, rather than for injection according to a randomized, oblivious process (cf. [4]).

Table 1. Contention-resolution protocols considered in this paper. (**US** stands for universally stable)

Protocol name	Which packet it advances:	**US**
Shortest-in-System (SIS)	The most recently injected packet into the network	$\surd$
Longest-in-System (LIS)	The least recently injected packet into the network	$\surd$
Furthest-to-Go (FTG)	The furthest packet from its destination	$\surd$
Nearest-to-Source (NTS)	The nearest packet to its origin	$\surd$
First-In-First-Out (FIFO)	The earliest arrived packet at the queue	**X**

Stability. *Stability* requires that the number of packets in the system remains bounded, as the system runs for an arbitrarily long period of time. Naturally, achieving stability in a packet-switched communication network depends on the *rate* at which packets are injected into the system, and on the employed contention-resolution protocol. Till our work, the study of stability has focused on *homogeneous* networks, that is, on networks in which the same contention-resolution protocol is running at all queues. In this work, we embark on a study of the effect of composing contention-resolution protocols on the stability of the resulting system. (By *composition* of contention-resolution protocols, we mean the simultaneous use of different such protocols at different queues of the system.)

Greedy Contention-Resolution Protocols. We consider only *greedy* protocols– ones that always advance a packet across a queue (but one packet at each discrete time step) whenever there resides at least one packet in the queue. The protocol specifies which packet will be chosen. We study five greedy protocols (all of which enjoy simple implementations): Say that a protocol is *stable* [3] on a given network if it induces a *bounded* number of packets in the network against any adversary with injection rate less than 1. (Note that the bound may depend on parameters of the network.) The first four of these protocols (namely, SIS, LIS, FTG and NTS) are *universally stable*– each is stable on *all* networks [1, Section 2.1]. In contrast, FIFO (one of the most popular queueing disciplines, because of its simplicity) is not universally stable [1, Theorem 2.10].

Approach. We consider all combinations of two from the four universally stable protocols, and we examine whether the corresponding composition is universally stable. We either show that it is, or we demonstrate a network and an adversary (with some specific injection rate less than 1) such that the composition is not stable on the network (against the adversary). In addition, in order to qualitatively evaluate how unstable are the compositions that turn out not to be universally stable, we also consider the FIFO protocol, which is known not be universally stable; we measure the instability of the composition against that of FIFO by establishing the best lower bound we can on the adversary's injection rate that implies instability for the composition and for FIFO, and we compare the two resulting lower bounds.

1.2 Contribution

Summary of Results. In this work, we initiate the study of the stability properties of heterogeneous networks with compositions of greedy contention-resolution protocols, such as SIS, LIS, FTG, NTS and FIFO, running on top of them. Our results are three-fold; they are summarized as follows:

– We establish universal stability for compositions of any two among the (universally stable) SIS, FTG, and NTS protocols (Theorem 1).
– We establish that, surprisingly, the composition of LIS with any of SIS, NTS and FTG is *not* universally stable (Theorem 2).
  To show this, we provide interesting combinatorial constructions of networks, for each queue of which we specify the contention-resolution protocol to be used, so that the composition of the protocols is unstable if the injection rate of the adversary is at least 0.519.
– We establish a new lower bound on the instability threshold of the FIFO protocol. More specifically, we provide an involved combinatorial construction of a network containing only FIFO queues and an adversary with injection rate 0.749 that result to instability (Theorem 3).
  This result not only significantly improves the current record (that is, the lowest known) instability threshold for FIFO [5, Theorem 3.1]. More importantly perhaps, it provides, as we argue, a standard for evaluating the lower bound (0.519) on the instability thresholds we established for the not universally stable compositions (Theorem 2). Since 0.519 is substantially less than 0.749 (in the climax [0.1]), and since lowering the instability threshold for FIFO has undergone a series of subsequent improvements in recent papers in the literature [1,8,5] culminating to the 0.749 shown in this work, these together may modestly suggest that composing two universally stable protocols may, surprisingly, turn out to exhibit more unstable behavior than a single protocol that is already known to not be universally stable (such as FIFO).

The combinatorial constructions of networks and adversaries that we have employed for showing that certain compositions of universally stable protocols are not universally stable significantly extend ones that appeared before in [1,3, 5]. In more detail, some of the tools we devise in order to obtain constructions of networks and adversaries that imply improved bounds are the following:

– We employ combinatorial constructions of networks with multiple "parallel" paths between a *common* origin and destination; we judiciously use such paths for the simultaneous injection of various non-overlapping flows.
– We introduce and use the technical notions of *investing flow* and *short intermediate flow*; these are some special cases of packet flows that we use in our adversarial constructions that consist of inductive *phases*. Roughly speaking, an investing flow injects packets in a phase which will remain in the system till the beginning of the next phase, in order to guarantee the induction hypothesis for the next phase; on the other hand, short intermediate flows

consist of packets injected on judiciously chosen paths of the network and their role is to block *all* packets of the investing flows (so that the latter will indeed remain in the system).

1.3 Related Work and Comparison

Composing Protocols and Objects. The issue of composing distributed protocols (resp., objects) to obtain other protocols (resp., objects), and the properties of the resulting (*composed*) protocols (resp., objects), has a rich record in Distributed Computing Theory (see, e.g., [10]). For example, Fernández *et al.* [6] study techniques for the composition of (identical) *causal* DSM systems from smaller modules each individually satisfying *causality*. Herlihy and Wing [9] establish that a composition of *linearizable* memory objects (possibly distinct), each managed by its own protocols, preserves linearizability. In the community of Security Protocols, the statement that security is not compositional is considered to be folklore [11].

Adversarial Queueing Theory. The model of *Adversarial Queueing Theory* was developed by Borodin *et al.* [3] as a more realistic model that replaces traditional stochastic assumptions made in Queueing Theory (cf. [4]) by more robust, worst-case ones. Subsequently, the Adversarial Queueing Theory model, and corresponding stability and instability issues, received a lot of interest and attention (see, e.g., [1,2,5,8,12]).

Stability and Instability Results. The universal stability of SIS, LIS, NTS and FTG was established by Andrews *et al.* [1, Section 2.1]. The instability of FIFO (on a specific network) was first established by Andrews *et al.* [1, Theorem 2.10]. Lower bounds of 0.85, 0.84 and 0.8357 on the instability threshold of FIFO (in the model of Adversarial Queueing Theory) were presented before by Andrews *et al.* [1, Theorem 2.10], Goel [8] and Diaz *et al.* [5, Theorem 3]. To the best of our knowledge, no previous work addressed the stability and instability properties of networks consisting of queues using multiple contention-resolution protocols.

Summary. For purpose of completeness and comparison, we summarize, in Table 2, all results shown in this work and in [1] that provide bounds on stability and instability properties of the universally stable, greedy contention-resolution protocols, and their compositions, that we considered.

Table 2. Range of injection rates for which the composition of the two protocols is unstable on some network. We denote **US** the universally stable compositions

	LIS	SIS	NTS	FTG
LIS	**US** ([1])			
SIS	[0.519,1] (Thm. 2)	**US** ([1])		
NTS	[0.519,1] (Thm. 2)	**US** (Thm. 1)	**US** ([1])	
FTG	[0.519,1] (Thm. 2)	**US** (Thm. 1)	**US** (Thm. 1)	**US** ([1])

2 Preliminaries

The definition of a *bounded adversary* $\mathcal{A}$ of rate (r, b) (where $b \geq 1$ is a natural number and $0 < r < 1$) in the Adversarial Queueing Theory model [3] requires that for any edge e and any interval I, the adversary injects no more than $r|I| + b$ packets during I that require edge e at their time of injection. Such a model allows for adversarial injection of packets that are "bursty" using the integer $b > 0$. Say that a packet p *requires* an edge e at time t if e lies on the path from its position at time t to its destination.

This definition for the adversary is used in Section 3 for proving that specific compositions of protocols are universally stable. On the other hand, when we consider adversarial constructions for proving instability of compositions of specific protocols (Section 4) and FIFO protocol (Section 5) in which we want to derive lower bounds, using an adversary with zero "burstiness" (that is, taking $b = 0$) results in more simplified proofs. Thus, for these purposes, we say that an adversary $\mathcal{A}$ has injection rate r if for every $t \geq 1$, every interval I of t steps, and every edge e, it injects no more than $r|t|$ packets during I that require edge e at the time of their injection. Clearly, an instability result for an adversary with no burstiness ($b = 0$) applies also to an adversary that may use burstiness ($b \geq 0$). Also, for simplicity, and in a way similar to that in [1], we omit floors and ceilings and sometimes count time steps and packets roughly. This only results to loosing small additive constants while we gain in clarity.

In order to formalize the behavior of a network under the Adversarial Queueing model, we use the notions of *system* and *system configuration.* A triple of the form (G, $\mathcal{A}$, P) where G is a network, $\mathcal{A}$ is the adversary and P is the used protocol on the network queues is called a system. Furthermore, the configuration C^t of a system (G, $\mathcal{A}$, P) in every time step t is a collection of sets $\{S_e^t : e \epsilon G\}$, such that S_e^t is the set of packets waiting in the queue of the edge e at the end of step t. If the current system configuration is C^t, then we can go to the system configuration C^{t+1} for the next time step as follows: (i) Addition of new packets to some of the sets S_e^t, each of which has an assigned path in G, and (ii) for each non-empty set S_e^t deletion of a single packet $p \epsilon S_e^t$ and its insertion into the set S_f^{t+1} where f is the edge following e on its assigned path (if e is the last edge on the path of p, then p is not inserted into any set.) The time evolution of the system is a sequence of such configurations $C^1, C^2, \ldots$, such that for all edges e and all intervals I, no more than $r|I| + b$ packets are introduced during I with an assigned path containing e. An execution of the adversary's construction on a system (G, $\mathcal{A}$, P) determines the time evolution of the system configuration.

In the constructions of executions in Sections 4 and 5, we split time into *phases.* In each phase, we study the evolution of the *system configuration* by considering corresponding *time rounds.* For each phase, we inductively prove that the number of packets of a specific subset of queues in the system increases in order to guarantee instability. This inductive argument can be applied repeatedly, thus showing instability. In addition, our constructions use networks that can be split into two symmetric parts. Thus, the inductive argument needs to be applied twice to establish increase in the number of packets residing at *two* different queues.

Also, in order to make our inductions work, we assume that there is a sufficiently large number of packets s_0 in the initial system configuration. This will imply instability results for networks with an *empty* initial configuration, as established by Andrews *et al.* [1, Lemma 2.9].

3 Universally Stable Compositions of Universally Stable Protocols

In order to prove the following theorem we make the following assumptions and definitions. Let $0 < \epsilon < 1$ be a real number. We assume that $r = 1 - \epsilon$, m is the number of network edges, and d is the length of the longest simple directed path in the network. Let us now define a sequence of numbers by the recurrence $k_j = \frac{mk_{j-1}+mb}{\epsilon}$, where $k_1 = \frac{mb}{\epsilon}$. Our techniques here are motivated by corresponding techniques in Andrews *et al.* [1].

Theorem 1. *If the used queueing disciplines in the system are a)* SIS *and* FTG*, then the system (*G*,* $\mathcal{A}$*,* SIS*,* FTG*) is stable, no queue ever contains more than* k_d *packets and no packet spends more than* $\frac{1}{\epsilon}(db + \sum_{i=1}^{d} k_i)$ *steps in the system, while if they are b)* NTS *and* FTG *or c)* SIS *and* NTS*, then there are never more than* k_d *packets in the system and no queue contains more than* $\frac{1}{\epsilon}(k_{d-1} + b)$ *packets, where* d *is the length of the longest simple directed path in* G *and* k_i *is an appropriately defined sequence of numbers.*

Proof. (*Sketch*) In order to prove this theorem, we first show the following two lemmas:

Lemma 1. *Let* p *be a packet waiting in a queue* e *at time* t *and suppose there are currently* $k - 1$ *other packets in the system requiring* e *that have priority over* p*. Then* p *will cross* e *within the next* $\frac{k+b}{\epsilon}$ *steps if the queueing discipline in queue* e *is* SIS*,* NTS *or* FTG*.*

Lemma 2. *When a packet* p *crossing its path arrives at the* j^{th} *edge on its path, there are at most* $k_j - 1$ *other packets requiring to traverse this edge with priority over* p*, if the used queueing protocol is a)* SIS *or* NTS*, b)* SIS *or* FTG*, and c)* NTS *or* FTG*.*

Lemma 1 claims that if a packet p waits in a queue e at time t and there are currently $k-1$ other packets in the system requiring to traverse edge e that have priority over p, then p will cross e within the next $\frac{k+b}{\epsilon}$ steps either the queueing discipline of queue e is SIS or NTS or FTG. In Lemma 2, we specialize Lemma 1 taking into account the distance of the queue e, in which packet p arrives at time t crossing its specified path, in relation to the first queue in its path. In this case, we prove that if queue e has distance j from the first queue on $p's$ path then there are at most $k_j - 1$ other packets in the system requiring to traverse the edge e that have priority over p when the system queues have as protocols (a) SIS or FTG, (b)NTS or FTG and (c) SIS or NTS.

Then based on these two lemmas, we prove this theorem using contradiction. Firstly, let us assume that there are $k_d + 1$ packets at some time all requiring the same edge. Then, the packet with the lowest priority of the $k_d + 1$ packets contradicts Lemma 2. Combining both lemmas, a packet p takes at most $\frac{k_j+b}{\epsilon}$ steps to cross the j^{th} edge on its path. Therefore, the upper bound for delay is $D = \frac{db+\sum_{i=1}^{d} k_i}{\epsilon}$. No packet spends more than D steps in the system. □

4 Unstable Compositions of Universally Stable Protocols

In this section, we show that the composition of LIS protocol with any of SIS, NTS and FTG protocols is *not* universally stable. Before proceeding to the adversary constructions for proving instability we give two basic definitions.

Definition 1. *We denote X_i the set of packets that are injected by the adversary into the system in the i_{th} round of a phase. These packet sets are characterized as investing flows because the number of their packets that will remain into the system at the end of the phase in which they have been injected, will be a portion of the packets that will be used as the initial ones in the next phase ensuring the reproduction of the induction hypothesis.*

Definition 2. *We denote $S_{i,k}$ the k_{th} set of packets the adversary injects into the system in the i_{th} round of a phase. These packet sets are characterized as short intermediate flows because their only purpose is to block other packet sets by using suitable paths.*

Theorem 2. *Let $r \geq 0.519$. There is a network N_1 and an adversary $\mathcal{A}$ of rate r, such that the system (N_1,$\mathcal{A}$,Queueing Disciplines) is unstable, if the used queueing disciplines are a)* LIS *and* SIS*, b)* LIS *and* NTS *or c)* LIS *and* FTG.

Proof. (*Sketch*) **Part a)** Consider the network N_1 in Figure 1.
Induction Hypothesis: At the beginning of phase j, there are s_j packets that are queued in the queues $f_4^{'}, f_7^{'}, f_8^{'}$ (in total) requiring to traverse the edges e_0, f_1, f_2, f_4, f_7.
Induction Step: At the beginning of phase $j + 1$ there will be more than s_j packets that will be queued in the queues f_4, f_7, f_8, requiring to traverse the edges $e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$.

The intuition behind this adversary construction and network topology is basically the preservation of the newly injected investing flows in each round of a phase inside the network until the end of the phase during which they have been injected. The tools we use to achieve this goal are the injections of short intermediate flows ($S_{i,k}$) and the introduction of parallel edges (f_4, f_5, f_6 and f_7, f_8, f_9) in the network topology. Although the short intermediate flows do not contribute actually to the number of packets that will be used as the initial flow

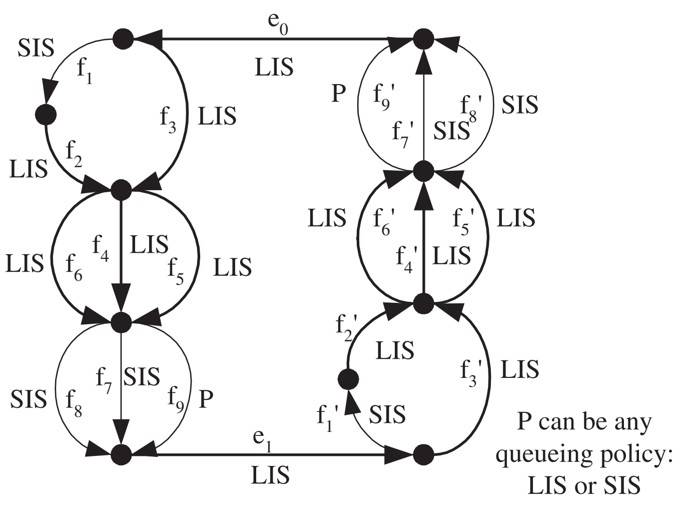


Fig. 1. The network N_1 running a composition of LIS and SIS

for the next phase, their role is essential as they are used for blocking investing flows of packets in consecutive rounds. Moreover, some of them have another role, too. They block short intermediate flows that have been injected in certain rounds in order they can be used for blocking investing flows in the next round of the one in which they have been injected. Also, an important point we should mention is the use of the short intermediate flow that is injected in the first round of a phase j ($S_{1,1} - flow$) to block a portion of the initial packets that are in the system at the beginning of the phase (s_j) in order these packets to be used twice for blocking investing flow X_1 to the first and the next round of the current phase. As far as concerns the parallel edges, the purpose of their presence in the network topology is to be guaranteed that the paths of the packet flows injected in the same round do not overlap. The first set of parallel edges f_4, f_5, f_6 is used in order short intermediate flows that are injected at the same round to not overlap, while the second set of parallel edges f_7, f_8, f_9 is used mainly in order the paths of investing flows and the paths of short intermediate flows that are injected at the same round to not overlap.

We will construct an adversary $\mathcal{A}$ such that the induction step will hold. Proving that the induction step holds, we ensure that the induction hypothesis will hold at the beginning of phase $j+1$ for the symmetric edges with an increased value of s_j, $s_{j+1} > s_j$. In order to prove that the induction step works, we should consider that there is a large enough number of packets s_j in the initial system configuration. During phase j the adversary plays four rounds of injections. The sequence of injections is as follows:

Round 1: It lasts s_j time steps. *At the beginning of this round*, there are s_j packets ($S - flow$) in the queues $f_4^{'}, f_7^{'}, f_8^{'}$ (in total) requiring to traverse the edges e_0, f_1, f_2, f_4, f_7.

Adversary's behavior. During this round the adversary injects in queue e_0 a set X_1 of $|X_1| = rs_j$ packets wanting to traverse the edges $e_0, f_3, f_4, f_7, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$. At the same time, the adversary injects a set $S_{1,1}$ of $|S_{1,1}| = rs_j$ packets in queue f_1 that require to traverse only the edge f_1.

At the end of this round, there are rs_j packets of $S-flow$ in queue f_1 wanting to traverse the edges f_1, f_2, f_4, f_7. Also, there is a set X_1 of $|X_1| = rs_j$ packets in queue e_0 wanting to traverse the edges $e_0, f_3, f_4, f_7, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$.

Round 2: It lasts rs_j time steps. *Adversary's behavior:* During this round the adversary injects a set X_2 of $|X_2| = r^2s_j$ packets in queue e_0 requiring to traverse the edges $e_0, f_3, f_4, f_8, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$. In addition, the adversary injects a set $S_{2,1}$ of $|S_{2,1}| = r^2s_j$ packets in queue f_2 wanting to traverse the edges f_2, f_5, f_7.

At the end of this round, there are rs_j packets of X_1-flow in queue f_4 wanting to traverse the edges $f_4, f_7, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$. Also, there are r^2s_j packets of X_2-flow in queue e_0 requiring to traverse the edges $e_0, f_3, f_4, f_8, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$ and r^2s_j packets of $S_{2,1}-flow$ in queue f_2 wanting to traverse the edges f_2, f_5, f_7.

Round 3: It lasts r^2s_j time steps. *Adversary's behavior:* During this round the adversary injects a set X_3 of $|X_3| = r^3s_j$ packets in queue f_4 wanting to traverse the edges $f_4, f_9, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$. Also, the adversary injects a set $S_{3,1}$ of $|S_{3,1}| = r^3s_j$ packets in queue f_5 wanting to traverse the edges f_5, f_7. Finally, the adversary injects a set $S_{3,2}$ of $|S_{3,2}| = r^3s_j$ packets in queue f_2 wanting to traverse the edges f_2, f_6, f_8.

At the end of this round, there are rs_j packets of $X_1 - flow$ in queues f_4, f_7 in total ($rs_j - r^2s_j$ packets in queue f_4 and rs_j packets in queue f_7) wanting to traverse the edges $f_4, f_7, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$. Also, in queue f_4 there are r^2s_j packets of $X_2 - flow$ requiring to traverse the edges $f_4, f_8, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$ and r^3s_j packets of $X_3 - flow$ wanting to traverse the edges $f_4, f_9, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$. Moreover, there are r^3s_j packets of $S_{3,1} - flow$ in queue f_5 requiring to traverse the edges f_5, f_7 and r^3s_j packets of $S_{3,2} - flow$ in queue f_2 requiring to traverse the edges f_2, f_6, f_8.

Round 4: It lasts r^3s_j time steps. *Adversary's behavior:* During this round, the adversary injects a set X_4 of $|X_4| = r^4s_j$ packets in queue f_3 wanting to traverse the edges $f_3, f_4, f_7, e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$. Besides the short intermediate flows $(S_{3,1}, S_{3,2})$ that have been injected into the system during the previous round and still remain into the system at the beginning of this round, the system configuration at the beginning of round 4 also consists of the investing flows X_1, X_2, X_3 that have been injected into the system during the previous rounds. During this round, these investing flows along with the investing flow that is injected into the system during this round $(X_4 - flow)$ continue to remain into the system independently of the adversarial injection rate r because they are blocked by the short intermediate flows $S_{3,1}, S_{3,2}$.

At the end of this round, the number of packets in queues f_4, f_7, f_8 requiring to traverse the edges $e_1, f_1^{'}, f_2^{'}, f_4^{'}, f_7^{'}$ is

$$s_{j+1} = |X_1| + |X_2| + |X_3| + |X_4| = rs_j + r^2s_j + r^3s_j + r^4s_j \tag{1}$$

In order to have instability, we must have $s_{j+1} > s_j$. Therefore from (1), we should have $rs_j + r^2 s_j + r^3 s_j + r^4 s_j > s_j$, i.e. $r \geq 0.519$. This argument can be repeated for an infinite number of phases ensuring that the number of packets at the end of a phase will be bigger than at the beginning of the phase.

Part b) This part of the theorem can be proved similarly to the previous part. One difference here is the replacement of the protocol of queues that use SIS by NTS. The topology of the used network N_2 and the adversary construction for proving instability of the system (N_2,$\mathcal{A}$,LIS, NTS) are similar to the first part of the theorem. Especially, short intermediate flows have the same blocking effects as before over investing flows because their injection in the same queues as before are enough to guarantee their priority over investing flows when they conflict in queues that use NTS.

Part c) This part of the theorem can be proved similarly to the previous parts. Thus, the construction of the adversary $\mathcal{A}$ for proving instability of the system (N_3,$\mathcal{A}$,LIS,FTG) is similar to the other parts. As far as concerns the network topology a difference here is that the queues, that have as protocol SIS in the first part, are now using FTG. Another difference that concerns the used network topology is that it contains additional paths that start at FTG queues and have sufficient lengths, such that the injected short intermediate packet flows have the same blocking effects over the injected investing packet flows, as in the proofs of the previous parts, when they conflict in queues that use FTG. □

5 A Lower Bound for FIFO Instability

In this section, we present an adversary construction that lowers significantly the injection rate bound for which FIFO is unstable to 0.749 on the network that we consider in Figure 2.

Theorem 3. *Let $r \geq 0.749$. There is a network $\mathcal{N}$ and an adversary $\mathcal{A}$ of rate r, such that the ($\mathcal{N}$, $\mathcal{A}$,* FIFO*) system is unstable.*

Proof. (*Sketch*) We consider the network $\mathcal{N}$ in Figure 2.
Induction Hypothesis: At the beginning of phase j, there are s_j packets that are queued in the queues $e_0, f_3', f_4', f_5', f_6'$ (in total) requiring to traverse the edges e_0, f_1, f_3, f_5, all these packets are able to depart from their initial edges to the symmetric part of the network (f_1, f_3, f_5) as a continuous flow in s_j time steps, and the number of packets that are queued in queues f_4', f_6' is bigger than the number of packets that are queued in queues f_3', f_5'.
Induction Step: At the beginning of phase $j+1$ there will be more than s_j packets (s_{j+1} packets) that will be queued in the queues f_3, f_5, f_4, f_6, e_1 (in total) requiring to traverse the edges e_1, f_1', f_3', f_5', all of which will be able to depart from their initial edges to the symmetric part of the network (f_1', f_3', f_5') in s_{j+1} time steps as a continuous flow and the number of packets that will be queued in queues f_4, f_6 will be bigger than the number of packets that will be queued in queues f_3, f_5.

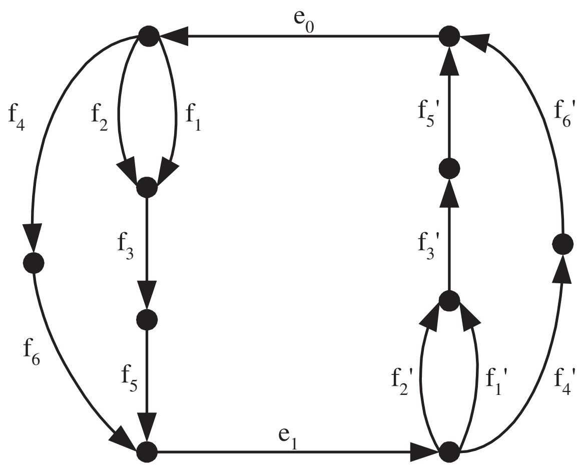


Fig. 2. FIFO network $\mathcal{N}$

Notice that our inductive argument claims that if at the beginning of phase j all s_j packets, that are queued in queues $e_0, f_3^{'}, f_4^{'}, f_5^{'}, f_6^{'}$ requiring to traverse the edges e_0, f_1, f_3, f_5, manage to traverse their initial edges in s_j time steps as a continuous flow, then at the beginning of phase $j+1$ all s_{j+1} packets, that will be queued in queues f_3, f_5, f_4, f_6, e_1 requiring to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$, will be able to traverse their initial edges in s_{j+1} time steps as a continuous flow. This argument guarantees the reproduction of the induction hypothesis in queues f_3, f_5, f_4, f_6, e_1 even if there are flows (in particular in queues f_3, f_4, f_5) that do not want to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$, the packets of which are regularly spread among the packets that want to traverse these edges. Furthermore, this argument implies the third part of the inductive argument, which claims that if at the beginning of phase j, the number of packets that are queued in queues $f_4^{'}, f_6^{'}$ is bigger than the number of packets that are queued in queues $f_3^{'}, f_5^{'}$, then at the beginning of phase $j+1$ the number of packets that will be queued in queues f_4, f_6 will be bigger than the number of packets that will be queued in queues f_3, f_5. This happens because in the first round of the adversary's construction we inject packets in queue $f_4^{'}$ and if the third part of the induction hypothesis does not hold, then we cannot guarantee that all the initial s_j packets will depart their initial edges to the edges f_1, f_3, f_5 in s_j time steps as a continuous flow. However, we include it into the induction hypothesis for readability reasons.

We will construct an adversary $\mathcal{A}$ such that the induction step will hold. Proving that the induction step holds, we ensure that the induction hypothesis will hold at the beginning of phase $j+1$ for the symmetric edges with an increased value of s_j packets, $s_{j+1} > s_j$. From the induction hypothesis, initially, there are s_j packets (called $S - flow$) in the queues $e_0, f_3^{'}, f_4^{'}, f_5^{'}, f_6^{'}$ requiring to traverse the edges e_0, f_1, f_3, f_5. In order to prove the induction step, it is assumed that there is a set S with a large enough number of $|S| = s_j$ packets in the initial system configuration. During phase j the adversary plays three rounds of injections. The sequence of injections is as follows:

Round 1: It lasts s_j time steps. *Adversary's behavior:* During these steps, the adversary injects a set X of $|X| = rs_j$ packets in queue $f_4^{'}$ wanting to traverse the edges $f_4^{'}, f_6^{'}, e_0, f_2$, f_3, f_5, e_1, $f_1^{'}, f_3^{'}, f_5^{'}$ and a set S_1 of $|S_1| = rs_j$ packets in f_1 wanting to traverse f_1.

At the end of this round, all the packets of the set X are queued in e_0, while in queue f_1 remains a set S_{rem} of $|S_{rem}| = \frac{rs_j}{r+1}$ packets from the set S and a set $S_{1,rem}$ of $|S_{1,rem}| = \frac{r^2 s_j}{r+1}$ packets from the set S_1 mixed on a proportion equal to their initial proportion of their sizes (fair mixing property).

Round 2: It lasts rs_j steps. *Adversary's behavior:* The adversary injects a set Y of $|Y| = r^2 s_j$ packets in queue $f_4^{'}$ requiring to traverse the edges $f_4^{'}, f_6^{'}, e_0, f_4$, f_6, e_1, $f_1^{'}, f_3^{'}, f_5^{'}$. At the same time, the adversary injects a set S_2 of $|S_2| = r^2 s_j$ packets in f_2 wanting to traverse f_2, a set S_3 of $|S_3| = r^2 s_j$ packets in f_3 wanting to traverse f_3, and a set S_4 of $|S_4| = r^2 s_j$ packets in f_5 wanting to traverse f_5.

At the end of this round all the packets of the set Y are queued in queue e_0. Also, a set X_{rem,f_2} of $|X_{rem,f_2}| = \frac{r^2 s_j}{r+1}$ packets from the set X and a set S_{2,rem,f_2} of $|S_{2,rem,f_2}| = \frac{r^3 s_j}{r+1}$ packets from the set S_2 remain in queue f_2 mixed on a proportion equal to their initial proportion of their sizes. Furthermore a set X_{rem,f_3} of $|X_{rem,f_3}| = \frac{r^3 s_j + rs_j}{(r+1)(r^2+r+2)}$ packets from the set X, a set S_{rem,f_3} of $|S_{rem,f_3}| = \frac{r^3 s_j + rs_j}{(r+1)(r^2+r+2)}$ packets from the set S and a set $S_{3,rem}$ of $|S_{3,rem}| = \frac{r^4 s_j + r^2 s_j}{r^2+r+2}$ packets from the set S_3 remain in queue f_3 mixed on the proportion of the sizes with which they arrive in queue f_3 during this round. Finally, $\frac{r^4 s_j + r^2 s_j}{(r^2+r+2)(r^3+r^2+2r+2)}$ packets from the set X, $\frac{r^4 s_j + r^2 s_j}{(r^2+r+2)(r^3+r^2+2r+2)}$ packets from the set S, and $\frac{r^5 s_j + r^3 s_j}{r^3+r^2+2r+2}$ packets from the set S_4 remain in queue f_5 mixed on the proportion of the sizes with which they arrive in queue f_3 during this round.

Round 3: It lasts $r^2 s_j$ time steps. *Adversary's behavior:* During this round the adversary injects a set S_5 of $|S_5| = r^3 s_j$ packets in queue f_4 requiring to traverse the edge f_4 and a set Z of $|Z| = r^3 s_j$ packets in queue f_6 requiring to traverse the edges $f_6, e_1, f_1^{'}$, $f_3^{'}, f_5^{'}$.

At the end of this round a set Y_{rem} of $|Y_{rem}| = \frac{r^3 s_j}{r+1}$ packets from the set Y and a set $S_{5,rem}$ of $|S_{5,rem}| = \frac{r^4 s_j}{r+1}$ packets from the set S_5 remain in queue f_4 mixed on a proportion equal to their initial proportion of their sizes. Furthermore, a set Y_{rem,f_6} of $|Y_{rem,f_6}| = \frac{r^4 s_j}{(r+1)(r^2+r+1)}$ packets from the set Y and a set Z_{rem,f_6} of $|Z_{rem,f_6}| = \frac{r^5 s_j}{r^2+r+1}$ packets from the set Z are queued in queue f_6 mix on the proportion of the sizes with which they arrive in queue f_6 during this round.

The total number of packets in queue f_3 at the beginning of round 3 is

$$|T_1| = |X_{rem,f_3}| + |S_{rem,f_3}| + |S_{3,rem}| = \frac{(r^5 + r^4 + 3r^3 + r^2 + 2r)s_j}{(r+1)(r^2+r+2)} \quad (2)$$

However, $|T_1| \geq r^2 s_j, \forall r$. Thus, a number of $X_{rem,f_3}, S_{rem,f_3}, S_{3,rem}$ packets will remain in queue f_3 at the end of round 3. This number is $|T_2| = \frac{(2r-r^2-r^4)s_j}{(r+1)(r^2+r+2)}$. From this number $|S_{3,rem,f_3}| = \frac{(2r^2-r^3-r^5)s_j}{(r^2+r+2)^2}$ packets belong to the set S_3. In addition, the total number of packets that are in f_2 at the end of round 2 is $|T_3| = r^2 s_j$ that is equal to the round's time duration. Thus, all the packets that belong to the set X and were in f_2 at the end of round 2 ($|X_{rem,f_2}|$) traverse now f_2 and arrive to f_3 where they are blocked.

In order to have instability the number of packets that are queued in f_3, f_4, f_5, f_6, e_1 requiring to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$ at the end of this round, s_{j+1}, should be more than the initial s_j packets that were queued in the system in corresponding queues at the beginning of round 1. Therefore, it should hold

$$|Z| + |Y| + |X_{rem,f_2}| + |X_{rem,f_3}| + |X_{pass,f_3}| - |T_{round_3}| > s_j \quad (3)$$

Substituting in (3), we take

$$r^3 s_j + \frac{r^2 s_j}{r+1} + \frac{r^3 s_j + r s_j}{(r+1)(r^2+r+2)} + \frac{r^4 s_j + r^2 s_j}{(r^2+r+2)(r^3+r^2+2r+2)} > s_j \quad (4)$$

This holds for $r \geq 0.749$. Now in order to conclude the proof we prove:

Lemma 3. *For $r \geq 0.686$, the number of packets that remain in queues f_3, f_5 ($Q(f_3), Q(f_5)$) at the end of round 3 is less than or equal to the number of packets that remain in queues f_4, f_6 at the end of round 3 ($Q(f_4), Q(f_6)$).*

Notice that we have, till now, managed to reproduce the induction hypothesis in queues f_3, f_5, f_4, f_6, e_1 but with some packet flows (in particular in queues f_4, f_3, f_5) having empty spaces (packets that don't want to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$). In order for the induction step to work we must show that all the packets in these queues will manage to depart to the symmetric part of the network ($f_1^{'}, f_3^{'}, f_5^{'}$) in s_{j+1} time steps as a continuous flow. As we have shown above all the packets that are queued in queue e_1 want to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$ and their flow is continuous without empty spaces (packets that do not want to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$). Also, from Lemma 3, the number of packets in queues f_4, f_6 is bigger than the number of packets in queues f_3, f_5. Furthermore, the packets that are queued in queue f_6 can be seen as a continuous flow that wants to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$, while the set of packets in queue f_4 consists of packets that want to traverse the edges $e_1, f_1^{'}, f_3^{'}, f_5^{'}$ (Y_{rem}) and packets that are injections which require to traverse a single edge ($S_{5,rem}$), which can be considered as empty spaces. Because of that we should show that all the Y_{rem} packets manage to leave the edge f_4 during the s_{j+1} time steps. We formally establish:

Lemma 4. *For any injection rate r, all the Y_{rem} packets manage to leave the edge f_4 during s_{j+1} time steps.*

We have so far established two sufficient constraints on r for instability, namely that $r \geq 0.749$ and $r \geq 0.686$. Clearly, taking $r \geq \max\{0.749, 0.686\} = 0.749$ suffices for instability of the network $\mathcal{N}$ in the constructed execution. This concludes our proof. □

6 Discussion and Directions for Further Research

Our work opens up the study of the stability and instability properties of heterogeneous communication networks with multiple contention-resolution protocols running on top of them. A fundamental question that arises in this setting is whether there exists a structural explanation of the differences we have observed regarding the stability of different compositions of universally stable protocols. Is there any deep reason for the composition of LIS with another contention-resolution protocol to be unstable? Also, for the unstable compositions of pairs of protocols, can we characterize the graphs on which the composition is unstable? A corresponding characterization for networks on which a *single* greedy protocol is running (as opposed to a composition of greedy protocols) in terms of *graph minors* has been developed in [1, Section 3.2].

References

1. M. Andrews, B. Awerbuch, A. Fernandez, J. Kleinberg, T. Leighton, and Z. Liu, "Universal Stability Results for Greedy Contention-Resolution Protocols," *Journal of the ACM,* Vol. 48, No. 1, pp. 39–69, January 2001.
2. M. Andrews, A. Férnandez, A. Goel and L. Zhang, "Source Routing and Scheduling in Packet Networks," *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science,* pp. 168–177, October 2002.
3. A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan and D. Williamson, "Adversarial Queueing Theory," *Journal of the ACM,* Vol. 48, No. 1, pp. 13–38, January 2001.
4. H. Chen and D. D. Yao, *Fundamentals of Queueing Networks,* Springer, 2000.
5. J. Diaz, D. Koukopoulos, S. Nikoletseas, M. Serna, P. Spirakis and D. Thilikos, "Stability and Non-Stability of the FIFO Protocol," *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 48–52, July 2001.
6. A. Férnandez, E. Jiménez and V. Cholvi, "On the Interconnection of Causal Memory Systems," *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing,* pp. 163–170, July 2000.
7. S. Floyd and V. Paxson, "Difficulties in Simulating the Internet," *IEEE/ACM Transactions on Networking,* Vol. 9, No. 4, pp. 392–403, August 2001.
8. A. Goel, "Stability of Networks and Protocols in the Adversarial Queueing Model for Packet Routing," *Networks,* Vol. 37, No. 4, pp. 219-224, 2001.
9. M. P. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems,* Vol. 12, No. 3, pp. 463–492, 1990.
10. N. Lynch, *Distributed Algorithms,* Morgan Kaufmann, 1996.
11. J. Mitchell, Email Communication to I. Lee, April 2002.
12. P. Tsaparas, *Stability in Adversarial Queueing Theory,* M.Sc. Thesis. Department of Computer Science, University of Toronto, 1997.

Transformations of Self-Stabilizing Algorithms

Kleoni Ioannidou

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
ioannidu@cs.toronto.edu

Abstract. In this paper, we are interested in transformations of self-stabilizing algorithms from one model to another that preserve stabilization. We propose an easy technique for proving correctness of a natural class of transformations of self-stabilizing algorithms from any model to any other. We present a new transformation of self-stabilizing algorithms from a message passing model to a shared memory model with a finite number of registers of bounded size and processors of bounded memory and prove it correct using our technique. This transformation is not wait-free, but we prove that no such transformation can be wait-free. For our transformation, we use a new self-stabilizing token-passing algorithm for the shared memory model. This algorithm stabilizes in $O(n \log^2 n)$ rounds, which improves existing algorithms.

1 Introduction

There is a variety of different distributed models. Because of their different characteristics, we need to design different algorithms that solve a problem, one for each model. A standard approach to avoid this is to design general methods of transforming algorithms from one model to another [13,4]. A transformation of algorithms from one model to another takes, as input, an algorithm of the first model and produces, as output, an algorithm in the second model. These algorithms solve the same problem if they have the same interaction with the user interface. The sequence of operations that describes this interaction in an execution is the *trace* of the execution [13,4]. A transformation is correct if, for any algorithm, the trace produced in any execution of the transformed version can be produced in an execution of the original version [13,4].

We will restrict our attention to systems where transient faults may occur. These are failures caused by temporary processor or communication medium malfunctions. A transient fault may change the information stored in processors and/or the information contained in the communication medium of the system, but not the algorithm itself. Algorithms that eventually make the system regain normal behavior starting from an arbitrary state are called *self-stabilizing* [6].

We are interested in transformations of self-stabilizing algorithms between different models of computation that preserve self-stabilization. A transformation of self-stabilizing algorithms from model $\mathcal{M}$ to model $\mathcal{M}'$ takes, as input, a self-stabilizing algorithm A that works in model $\mathcal{M}$ and produces as output a self-stabilizing algorithm A' that works in model $\mathcal{M}'$. Intuitively, if the transformation is correct, algorithm A' eventually simulates algorithm A (and then, eventually stabilizes to the task of A). More formally, a *transformation of self-stabilizing algorithms from model $\mathcal{M}$ to model $\mathcal{M}'$ is correct* if, for any algorithm A of model $\mathcal{M}$, the resulting algorithm A' of model $\mathcal{M}'$ has a subset Q'' of its set of states Q', such that:

1. In every execution of the transformed algorithm A', eventually all the states will be in Q''.
2. For any execution a' of the transformed algorithm A', starting from a state in Q'', there is an execution a of the original algorithm A, such that a' and a have the same trace.

In the literature, there exist transformations of self-stabilizing algorithms between different models [8,9] proved to be correct based on specific characteristics of the models. We present a technique of proving correctness of a class of transformations of self-stabilizing algorithms. This is a general technique (i.e. it applies to any models). We give a new transformation of self-stabilizing algorithms from the message passing model to the shared memory model with finite number of registers of bounded size and processors with bounded memory. We prove correctness of this transformation using our technique. To define our transformation, we use the composition technique presented by Dolev [8], but with some technical details modified. This transformation uses a self-stabilizing token-passing algorithm (which is common in transformations of both general and self-stabilizing algorithms [8,4]) with the extra property that the processors get the token in a round-robin order. Such an algorithm is given by Dijkstra [6] for the composite atomicity model and Dolev [8] simulates it for the shared memory model. This algorithm stabilizes in $O(n^2)$ rounds. We present a new self-stabilizing round-robin token-passing algorithm, for the shared memory model, that stabilizes in $O(n \log^2 n)$ rounds.

Lynch [13] has a transformation of non-self-stabilizing algorithms from the message passing model to the shared memory model. We show that it can be easily modified to preserve stabilization and then compare our transformation with it. Lynch's transformation is wait-free but requires unbounded register size and unbounded memory processors. Our transformation works for bounded size registers and bounded memory processors but it is not wait-free. We prove that there is no wait-free transformation of self-stabilizing algorithms from the message-passing model to the shared memory model with a finite number of registers of bounded size and processors of bounded memory.

1.1 Models

We consider two different types of operations: the *interface operations* which are used for interaction between each processor and the user interface [13], and the *system operations*, which happen within the distributed system. An interface operation from the user interface to the distributed system is called an *input operation*, and an interface operation from the distributed system to the user interface is called an *output operation.* An interface operation only changes the local memory of a single processor. A system operation performed by a processor can be either *internal* if the processor uses and changes variables only in its local memory, or *external.*

We consider asynchronous models for which an execution is an alternating sequence of states (of the system) and operations. We consider *fair executions* [7] where each processor performs operations infinitely often. The *trace of an execution a* is the sequence of interface operations performed in a and is denoted trace(a) [13]. The *task* of a problem is a set of traces that satisfy the problem specifications. We consider transformations of self-stabilizing algorithms from model $\mathcal{M}$ to model $\mathcal{M}'$ for which there is one-to-one correspondence of the processors of the two models; i.e. processor P_i' of model $\mathcal{M}'$ simulates the behavior of processor P_i of model $\mathcal{M}$.

In the shared memory model, we consider processors that communicate using read/write registers. One processor at a time will be scheduled to atomically execute a read or write operation accompanied by some internal computation.

In the message passing model, channels are used for communication between neighbouring processors. Processor P_i communicates with its neighbour P_j by sending messages to a queue $L_{i,j}$ and by receiving messages from a queue $L_{j,i}$. Note that, for asynchronous systems, there is no fixed upper bound on the delivery time of messages. To model this, we could use two queues, $L'_{i,j}$, $L''_{i,j}$, instead of queue $L_{i,j}$, such that P_i sends messages to $L'_{i,j}$ and P_j receives messages from $L''_{i,j}$. The system delivers messages from $L'_{i,j}$ to $L''_{i,j}$ [4]. The results in this paper hold for both of these ways of describing communication in the message passing model. For simplicity, we use the first.

A commonly used message-passing model is the *message-driven model* where a receive step can only be performed by a processor when there is a message in its receiving queue. In this model, any atomic operation begins with a single receive step, followed by one or more internal steps and 0 or more send steps (during which no two messages are sent to the same queue). Consider an initial system state where the queues are empty and all the processors are waiting to receive a message. Starting from such a state the system is deadlocked. This kind of deadlock is called *communication deadlock.*

The majority of authors assume the existence of time-outs to deal with communication deadlock. Dolev [8] defines time-outs as environmental operations that will be executed when such a deadlock occurs. Dolev, Israeli, and Moran [9] assume a time-out device, which detects communication deadlocks and initializes the system to a state after which no communication deadlock ever occurs. Time-outs are not a desirable assumption for self-stabilizing asynchronous systems. First, it is unclear how they are implemented. Second, a time-out initializes a system to a known state, which is what self-stabilization is trying to accomplish. Katz and Perry [11] provide another solution to the problem of communication deadlock. They assume that, in any state, there is at least one processor whose next operation is to send a message. Therefore, they only consider algorithms that stabilize starting from a subset of states and not from any state.

We use another message passing model, defined by Attiya and Welch [4], called point-to point asynchronous message passing. A processor can perform a receive step on an empty queue. In this case, it returns a special symbol $\perp$. We define an atomic operation performed by some processor to be a sequence of 0 or more receive steps (at most one for each of its neighbours), followed by 1 or more internal steps and a sequence of 0 or more send steps (during which no two messages are sent to the same queue).

1.2 Known Transformations between Message Passing and Shared Memory

In the literature, there are many transformations of general algorithms between different versions of asynchronous message passing model and the asynchronous shared memory model. Most were not designed to preserve stabilization: their correctness depends on the system being initialized in certain states.

Bar-Noy and Dolev [5] present a way to get transformations between the shared memory model and the message passing model for algorithms that only use snapshots for communication. Attiya, Bar-Noy and Dolev [3] give a transformation of algorithms from the shared memory model with multi-reader, single-writer registers to the complete point-to-point message passing model with reliable channels, but processor failures. Attiya [1] improved the space complexity and message complexity of this transformation. Attiya and Welch [4,2] present a transformation of algorithms from the shared memory model to the totally ordered reliable broadcast model and two transformations of algorithms from the totally ordered reliable broadcast model to the message passing model.

A transformation from the message passing model to the shared memory model with registers of unbounded size and processors with unbounded local memory is presented by Lynch [13]. She uses a single-reader, single-writer register for every queue of the message-passing model. The sequences of messages enqueued are stored in the corresponding registers. In its local memory, each processor contains a copy of the sequence of messages stored in each register to which it can write, and a prefix of the sequence of messages stored in each register that it can read. A send operation is simulated by appending the new message to the local copy of the appropriate sequence of messages and then writing the new sequence to the corresponding register. A receive operation is simulated by reading the sequence in the appropriate register and comparing it with the corresponding local copy. If the sequence read in the register is the same as the local copy, then there is no message in the simulated queue to be received. Since a message is never removed from a register, each register needs unbounded size. Because the processors keep copies of the registers, they also need unbounded memory.

Note that Lynch's transformation can be easily modified to preserve stabilization. Specifically, the simulation of the receive operation is modified so that, if the corresponding sequence in the local memory of the processor is not a prefix of the sequence read in the register, then the processor updates its local memory with the sequence read from the register.

Dolev, Israeli, and Moran [8,9] present a transformation of self-stabilizing algorithms from the shared memory model to the message-driven model. Dolev [8] presents another transformation from Dijkstra's composite atomicity model to a read-write shared memory model using Dijkstra's self-stabilizing mutual exclusion algorithm and a self-stabilizing leader election algorithm. Only sketches of the proofs of correctness of these transformations are provided. Formal proofs can be obtained using our mapping technique presented in Section 2.

1.3 Overview

In Section 2, we present a technique for proving correctness of transformations of self-stabilizing algorithms between different models. In Section 3, we present a new transformation from the message passing model to the shared memory model and prove it correct using our technique. In Section 4, we present a token-passing algorithm that stabilizes in $O(n \log^2 n)$ rounds and is used by our transformation. In Section 5, we discuss the advantages and disadvantages of our transformation and prove that there is no wait-free transformation of self-stabilizing algorithms from the message passing model to the shared memory model with finite number of registers of bounded size and processors of bounded memory.

2 The Mapping Technique

In this section, we present a technique that can be used for proving correctness of a transformation of self-stabilizing algorithms from model $\mathcal{M}$ to model $\mathcal{M}'$; more specifically, for proving the second condition. In most transformations, every system operation in $\mathcal{M}$ is simulated by a procedure consisting of one or more operations in $\mathcal{M}'$. Note that, since the original and the transformed algorithm solve the same task, the interface operations are the same for both models (i.e. they simulate themselves). The idea is to define a mapping from the executions of a transformed algorithm in $\mathcal{M}'$ to executions of the original algorithm in $\mathcal{M}$ so that the simulations of different operations do not interfere with one another. To do this, we begin by defining a *mapping of states.* This is a function χ from Q'' to Q, where Q is the set of states of the original algorithm and Q'' is the subset of states of the transformed algorithm specified in the first correctness condition. For each i, let $env_i(s)$ denote the environment of P_i in state s, namely the contents of its local memory and the part of the communication medium it can access directly. Similarly, let $env'_i(s')$ denote the environment of P'_i in state s'. Then, χ induces a mapping ξ_i of the states of the environment of P'_i: if $\chi(s') = s$ then $\xi_i(env'_i(s')) = env_i(s)$. Conversely, if $\xi_i(env'_i(s')) = env_i(s)$ for all $i \in \{1, \ldots, n\}$, then $\chi(s') = s$.

Next, we define a *mapping of operations.* This is a function, σ, that maps every operation, τ of $\mathcal{M}$ to some finite, nonempty sequence of operations $\tau'_1, \ldots, \tau'_k$ of model $\mathcal{M}'$ such that, for every sequence of transitions (s'_1, τ'_1, s'_2), (s'_2, τ'_2, s'_3),

$\ldots$,$(s'_k, \tau'_k, s'_{k+1})$ in $\mathcal{M}'$ performed by some processor P'_i starting from a state $s'_1 \in Q''$, there is a transition (s_1, τ, s_2) performed by P_i in $\mathcal{M}$ such that $\chi(s'_1) = s_1$ and $\chi(s'_{k+1}) = s_2$. We say that $\sigma(\tau) = \tau'_1, \ldots, \tau'_k$ *simulates* τ. Note that, if $\sigma(\tau)$ can be applied by some processor P'_i in state $s' \in Q''$, then $\xi_i(env'_i(s')) = env_i(\chi(s'))$ so, in state $\chi(s')$, τ can be applied by P_i. Any state between the operations in $\sigma(\tau)$ executed by P'_i (for some operation τ in $\mathcal{M}$) is called an *intermediate state* for P'_i. In $\mathcal{M}'$, there may be operations that are not part of a sequence of operations simulating some operation of $\mathcal{M}$. We call these *auxiliary* operations. If τ' is an auxiliary operation, and (s',τ', s'') is a transition performed by P'_i, then $\xi_i(env'_i(s')) = \xi_i(env'_i(s''))$.

Execution $a' = s'_1, t'_1, s'_2, t'_2, \ldots$ *simulates execution* $a = s_1, t_1, s_2, t_2, \ldots$ if for all $1 \leq i \leq n$, the following four conditions hold:

1. The first state of a' maps to the first state of a; i.e. $\chi(s'_1) = s_1$.
2. Every operation performed by P_i in a is simulated by a disjoint subsequence of operations in a' performed by P'_i. Furthermore, these subsequences are performed by P'_i in a' in the same order as the corresponding operations by P_i in a. Formally, if $\tau_{i,1}, \tau_{i,2}, \ldots$ is the sequence of operations performed by P_i in a, and $\tau'_{i,1}$, $\tau'_{i,2}, \ldots$ is the sequence of operations performed by P'_i in a', then there exist $1 = k_0 < j_1 \leq k_1 < j_2 \leq k_2 < \cdots$ such that, for all $l \geq 1$, the sequence of operations $\tau'_{i,j_l}, \tau'_{i,j_l+1}, \ldots, \tau'_{i,k_l}$ simulates τ_l and if τ_l is the ϱ^{th} operation of a, (i.e. $t_\varrho = \tau_{i,l}$), and τ'_{i,j_l} is the ϱ'^{th} operation of a', (i.e. $t'_{\varrho'} = \tau'_{j_l}$), then the state immediately before $t'_{\varrho'}$ simulates the state immediately before t_ϱ; i.e. $\xi_i(env'_i(s'_{\varrho'})) = env_i(s_\varrho)$.
3. Processor P'_i does not simulate any operation in a' that is not executed by P_i in a. In particular, in between any two sequences of operations by P'_i that simulate two consecutive operations by P_i, processor P'_i may perform only auxiliary operations.
4. *Atomicity Property*: Atomicity of the operations in $\mathcal{M}$ is preserved in a': If s'_r is an intermediate state for P'_i and (s'_r, t'_r, s'_{r+1}) is a transition performed by another processor P'_m, then:
   a) operation t'_r does not change the environment of processor P'_i, and
   b) if operation t'_r belongs to a sequence of operations $\sigma(\lambda)$ for some operation λ of model $\mathcal{M}$, then the environment of P'_i and the environment of P'_m do not have any parts in common.

Part (b) of the atomicity property prevents any processor P'_i from performing a simulation of an operation using information in some intermediate state. This is necessary because an intermediate state does not necessarily map to a state in an execution of the original algorithm and, hence, may have information that should not be available to other processors.

If execution a' simulates execution a, then the sequence of interface operations performed by some processor in a' is the same as the sequence of interface operation performed by the corresponding processor in a. However, a and a' do not have necessarily the same trace, because two interface operations by different processors may be in reversed order in a and a'. We will show how to create a valid execution b of the original algorithm with the same trace as a' by reordering operations in a. Then, to prove the second correctness condition for a transformation, it suffices to prove that for any execution a' of the transformed algorithm, starting from a state in Q'', there is an execution a of the original algorithm, such that a' simulates a.

Theorem 1. *If there is an execution a' of A' in $\mathcal{M}'$ that simulates an execution a of an algorithm A in $\mathcal{M}$, then there is an execution b of algorithm A such that trace(a') = trace(b).*

Proof. Let $a' = s'_1, t'_1, s'_2, t'_2, \ldots$ be an execution in model $\mathcal{M}'$ and let $a = s_1$, t_1, s_2, $t_2, \ldots$ be an execution of an algorithm A in model $\mathcal{M}$ such that a' simulates a. Suppose that each operation t_i of a is performed by processor P_{h_i}. We define the permutation $\tau = \tau_1, \tau_2, ..$ of the operations in a as follows: if the first operation of $\sigma(t_i)$ occurs before the first operation of $\sigma(t_j)$ in a', then t_i occurs before t_j in τ. Recall that an interface operation simulates itself, therefore, the sequence of interface operations in τ is equal to trace(a'). We denote by s'_{y_i} the state in a' immediately before the first operation of $\sigma(\tau_i)$ is performed. Let $b = q_1, \tau_1, q_2, \tau_2, \ldots$ be the execution obtained by applying the operations in τ to $q_1 = s_1$. We prove (using induction) that, for all $i \geq 1$, $\xi_{h_i}(env'_{h_i}(s'_{y_i})) = env_{h_i}(q_i)$. Consequently, since $\sigma(\tau_i)$ is applicable to s'_{y_i} (by the definition of a'), it follows that τ_i is applicable to q_i. Therefore, b is an execution of algorithm A. We present the technical details of this proof in [10]. □

Lynch and Vaandrager [12] present a number of different techniques for proving correctness of transformations of non self-stabilizing algorithms. These techniques and ours both use mapping of states in a similar way. Additionally, we define simulation of operations because this provides a natural way of describing mappings of execution fragments that simulate an operation of the original algorithm. Another difference between our technique and theirs is that they deal with initial states. Instead, we deal with initial execution fragments during which the transformed self-stabilizing algorithm does not necessarily simulate the original self-stabilizing algorithm. These fragments happen before we reach a state in Q'' and are eliminated by the first condition of correctness of transformation. Because of the atomicity property, our technique is more restrictive than theirs. However, when the atomicity property holds, our technique is simpler to use. Specifically, we only have to examine each processor's computation separately.

3 A New Transformation of Self-Stabilizing Algorithms from Message Passing to Shared Memory

In this section, we propose a transformation of self-stabilizing algorithms from the message passing model to the shared memory model with a finite number of registers of bounded size and processors with bounded memory. A processor that simulates sending messages will store them either in its local memory or in the registers to which it can write. Since both registers and local memory have bounded size, processors cannot store an unbounded number of simulated messages and, as a result, some of the messages might be lost. In our transformation, we ensure that no message will be lost by forcing every value written to a single-reader, single-writer register to be read exactly once.

For every queue $L_{i,j}$ in the message passing model, we use a single-reader, single-writer register $M_{i,j}$ in the shared memory model that will be written by P'_i and read by P'_j. For every transformed algorithm, each processor repeatedly executes the following sequence of operations, which we call a *simulation cycle.* A processor P'_i copies the value of register $M_{j,i}$ into its local array IN_i, for each neighbour P_j of P_i. The last read operation ends with an internal computation that, given IN_i, produces the values OUT_i to be sent. Then, P'_i copies the corresponding value from its local array OUT_i into register $M_{i,j}$, for each neighbour P_j of P_i. When the algorithm stabilizes, the simulation cycle of different processors will not overlap and will be performed by processors in a round-robin order.

To implement this algorithm, we compose it with a self-stabilizing token-passing algorithm with the additional property that the token circulates in a round-robin order among the processors. (When a processor gets the token, it executes a simulation cycle.) We use Dolev's fair composition technique [8], but present a different formal description of this technique that fixes some technical details. Let S_1 be the set of states of the communication medium and the processors (excluding the values of their program counters) when they execute some algorithm A_1 (eg. token-passing). Let A_2 be a program (eg. the transformed algorithm) that works correctly assuming that A_1 has stabilized. Specifically, processors in A_2 can use but not modify the part of the state in S_1. This allows both A_1 and A_2 to change the program counter of the processor that executes them. This is different (and more natural) than in [8], where a processor executing A_2 cannot modify any part of the processor state used by A_1. In that case, the fair composition technique does not apply when A_1 is a token-passing algorithm, since a processor gives away its token in response to specific change in the program counter by A_2.

To apply the fair composition technique, it suffices to prove that the algorithms produced by our transformation are correct assuming that the token-passing algorithm stabilizes. We do this using the mapping technique presented in Section 2. We choose the set of states Q'' of the transformed algorithm to be the states after which the token-passing algorithm has stabilized. Since the token-passing algorithm is self-stabilizing, the first condition for the correctness of the transformation holds. It remains to prove the second condition.

Consider an execution a' and a state q'' in a'. Let q' be the state immediately after the most recent write to $M_{k,j}$ (by P'_k) that occurs before q'' of a'. (If no write to $M_{k,j}$ is performed before q'' in a', then q' is the initial system state in a'.) Note that no write to $M_{k,j}$ occurs between q' and q''. We say that P'_j *can read a new value* from $M_{k,j}$ at q'' if P'_j does not read from $M_{k,j}$ between q' and q''. Now, we define the mapping of states χ. First, consider a state $s' \in Q''$, in which processor P'_i has just received the token. Recall that the simulation cycle is executed by the processors in a round-robin order and, during each simulation cycle, a processor reads all the registers that it can read and writes to all the registers that it can write to. It follows that, in s', P'_i can read new values from all the registers it can read and no processor can read new values from the registers where P'_i can write to. Every other processor P'_j can read a new value from $M_{k,j}$ in s', if and only if P_k is a neighbour of P_j and k comes between j and i in the round-robin ordering, i.e. either $j < k < i$, or $k < i < j$, or $i < j < k$. If, in $s' \in Q''$, some processor P'_r can read a new value from register $M_{l,r}$ and $M_{l,r} \not= \perp$, then in $\chi(s') \in Q$, the queue $L_{l,r}$ contains exactly one message with the value of $M_{l,r}$. Otherwise $L_{l,r}$ is empty. If P'_i has already started executing the simulation cycle, then by its program counter we know which operations are finished. Therefore, we know the state s'' before the simulation cycle began and we set $\chi(s') = \chi(s'')$.

Lemma 1. *The simulation cycle simulates an operation in the message passing model.*

To prove Lemma 1, the main idea is that a simulation cycle performed by P'_i simulates an operation where P_i receives one message or $\perp$ from each of its neighbours and sends a message to each of its neighbours. The messages received have the same contents as the corresponding values read by P'_i during the simulation cycle, and the messages sent have the same contents as the corresponding values written by P'_i during the simulation cycle.

Theorem 2. *For each execution a' of the transformed algorithm in the shared memory model that starts from a state in Q'', there is an execution a of the original algorithm of the message passing model such that a' simulates a.*

Proof. (Sketch) Let a' be an arbitrary execution of the transformed algorithm in the shared memory model that starts from a state in Q''. First, we show that the atomicity property holds for a'. Part (a) follows from the fact that every processor that does not have the token does not change its environment. Part (b) holds because only one processor at a time has the token, and simulation of operations is performed by a processor only when it has the token.

We create an execution a to satisfy the first, second, and third conditions in the definition of simulation of execution. Execution a' can be viewed as a sequence of execution fragments, each of which contains one simulation cycle and some auxiliary operations. Auxiliary operations affect only information related to the token-passing algorithm and do not affect whether a processor can read a new value from a register. Consequently, from the definition of χ, if (s'_1, t', s'_2) is a transition in a', where t' is an auxiliary operation, then s'_1 and s'_2 simulate the same state in a (i.e: $\chi(s'_1) = \chi(s'_2)$). Therefore, we construct an execution a as follows: If s'_0 is the first state of execution a', then $\chi(s'_0)$ is the first state of a. Whenever P'_i performs a simulation cycle in a', P_i performs the operation in a that is simulated by the simulation cycle. □

The stabilization time of the transformed algorithms depend on the stabilization time of the token-passing algorithm. The maximum number of operations performed in a simulation cycle is $s(n) = 2\max_i\{d_i\}$, where d_i is the degree of processor P_i. The token-passing algorithm presented in the next section stabilizes in $O(s(n)n\log^2 n)$ rounds. If the original algorithm stabilizes in $f(n)$ rounds, then the transformed algorithm stabilizes in $O(n(\log^2 n + f(n))\max_i\{d_i\})$ rounds [10].

4 A New Self-Stabilizing Token-Passing Algorithm

In this section, we give a self-stabilizing token-passing algorithm for the shared memory model with the additional property that the processors get the token in a round-robin order. We define a strict binary tree, T of height $\lceil \log_2 n \rceil$, whose leaves are labeled by the processors in increasing order of their identifiers. Each internal node, n_0, is labeled by a pair of processors, $< P', P'' >$. Processor P' has the minimum identifier among the processors that label the leaves of the left subtree rooted at n_0. Processor P'' has the minimum identifier among the processors that label the leaves of the right subtree rooted at n_0. Note that the identifier of P' is smaller than the identifier of P''.

For each non-leaf node with label $< P'_i, P'_j >$, our algorithm uses a 1-bit multi-reader, multi-writer register, $C_{i,j}$, readable and writable by P'_i and P'_j. The processors P'_i and P'_j take turns owning this node. We say that *processor P'_i owns the node* when $C_{i,j} = 0$ and *gives the node to P'_j* by writing 1 to $C_{i,j}$. Similarly, *P'_j owns the node* when $C_{i,j} = 1$ and it *gives the node to P'_i* by writing 0 to $C_{i,j}$. Consequently, in any state, no two processors can own the same node. In our algorithm, processors P'_i and P'_j never write concurrently to $C_{i,j}$, so it is easy to simulate $C_{i,j}$ using two single-reader, single-writer 1-bit registers.

Every processor labels exactly one leaf, and it occurs as part of the label of the parent of that leaf. Moreover, each processor occurs as part of the label of at most one node at any given depth and these nodes occur at consecutive levels of the tree starting from the leaf it labels. A processor that executes our algorithm performs the following for every node it partially labels, in decreasing order of depth: it gives away the node and then waits until it owns this node by repeatedly reading the corresponding register. When it owns the last node it partially labels (i.e. the most distant from the leaf), it gets the token. When it finishes with the token, it gives the token away, by giving away the first (i.e. deepest) non-leaf node it partially labels. Then it repeats the process to get the token again. The formal description of the token-passing algorithm appears in [10].

For $1 < i \leq n$, there is exactly one $k_i \in \{1, \ldots, i-1\}$ such that $< P'_{k_i}, P'_i >$ labels a node. Let N_i denote the set $\{j \mid < P'_i, P'_j > \text{ labels a node}\} \subseteq \{i+1, \ldots, n\}$. Let $N'_i = N_i$ if $i = 1$; otherwise, let $N'_i = N_i \cup \{k_i\}$. Note that, if i is even, then $N'_i = \{i-1\}$ and $N_i = \emptyset$. For $i \not= 1$, the node labeled$< P'_i, P'_j >$ is a descendant of the node labeled $< P'_{k_i}, P'_i >$.

4.1 Correctness and Complexity of the Token-Passing Algorithm

Processor P'_i is *blocked* if it waits for the ownership of a node forever. One way this can happen is when P'_i repeatedly reads $C_{i,j} = 1$, for some fixed $j \in N_i$. The other way this can happen is when P'_i repeatedly reads $C_{k_i,i} = 0$.

Lemma 2. *(Deadlock freedom) There is no state in any execution of the token-passing algorithm where all processors are blocked.*

Proof. Assume there is an execution in which every processor is blocked and therefore the values of all registers remain fixed. If all registers $C_{i,j}$ have value 0 then P'_1 is not blocked. Consider the lowest level, l, of the tree where there is a register $C_{k_j,j}$ with value 1. Since processor P'_j is blocked, there exists $g \in N_j$ such that $C_{j,g} = 1$. By construction of the tree, the node labeled $< P'_j, P'_g >$ belongs to the right subtree of the node labeled $< P'_{k_j}, P'_j >$. Therefore, the register $C_{j,g}$ corresponds to a node at a lower level than l. This contradicts the choice of l. □

Lemma 3. *(Lockout freedom) Every processor will get the token infinitely many times.*

Proof. By the algorithm, if a processor never gets the token, it has to be blocked in some read operation. First, consider label $< P'_i, P'_j >$ of the deepest node n_0 such that processor P'_i is permanently blocked reading $C_{i,j} = 1$. Processor P'_j cannot be blocked reading this node, because this would require $C_{i,j} = 0$ forever. But P'_j has to be blocked in some node because, otherwise, it would write 0 to $C_{i,j}$. As a result, P'_j has to be blocked in a node with label $< P'_j, P'_r >$ reading $C_{j,r} = 1$. This node is a descendant of n_0, contradicting the choice of n_0. Therefore, no processor P'_i can be blocked reading $C_{i,j} = 1$.

Now consider the least deep node, n_2, labeled by $< P'_{k_i}, P'_i >$, such that P'_i is blocked reading $C_{k_i,i} = 0$. Processor P'_{k_i} has to be blocked at some node; otherwise, it would eventually set $C_{k_i,i}$ to 1. From the previous part of this proof, P'_{k_i} has to be blocked at another node, n_3, with label $< P'_l, P'_{k_i} >$ reading $C_{l,k_i} = 0$. Node n_3 is an ancestor of n_2, which contradicts the choice of n_2.

We conclude that every processor executes all the operations of the token-passing algorithm infinitely often. As a result, every processor gets the token infinitely often. □

We say that a processor P'_i has executed a *cycle of the algorithm* starting from a state s', if it reaches a state s'' in which the program counter of P'_i has the same value as in s' and, during the execution fragment that starts with s' and ends with s'', P'_i got the token at least once. We say that the property *AllCompleteCycle* holds in a state of an execution that is reached after every processor has executed at least one cycle of the algorithm. Note that if AllCompleteCycle holds in some state of an execution, it holds in all subsequent states of that execution. Since the token-passing algorithm is deadlock free and lockout free, every processor will execute a cycle of this algorithm infinitely many times. So, starting from any state, a state where AllCompleteCycle holds is reached eventually.

Let $PATH_i$ denote the simple path in the tree T that starts from the leaf labeled by P'_i and goes up to the root. For all $l \neq i$ such that P'_l partially labels a node in PATH$_i$, let FirstInPath$_{i,l}$ denote the first node in PATH$_i$ partially labeled by P'_l. The proof of Lemma 4 appears in [10].

Lemma 4. *After AllCompleteCycle holds, if P'_i has the token, then processor P'_l does not own FirstInPath$_{i,l}$, for all $l \neq i$ such that P'_l partially labels a node in PATH$_i$.*

Lemma 5. *At most one processor has the token in any state of an execution where AllCompleteCycle holds.*

Proof. By contradiction. Assume that both P'_i and P'_j have the token, where $i < j$. Since P'_i and P'_j own all nodes that they partially label, it follows that P'_i and P'_j cannot label the same node. Let n_0 be the first node in the intersection of PATH$_i$ and PATH$_j$. Suppose that n_0 is labeled by $< P'_h, P'_k >$. Since $i < j$, processor P'_i labels a leaf in the left subtree of n_0, and PATH$_i$ contains the left child of n_0. But the left child of n_0 is partially labeled by P'_h. Therefore, $n_0 \neq$ FirstInPath$_{i,h}$. By Lemma 4, P'_h does not own FirstInPath$_{i,h}$. Since P'_h has executed a cycle, it follows from the algorithm that there is at most one node partially labeled by P'_h that it does not own. Thus P'_h owns n_0. Similarly P'_k owns n_0. This is impossible. □

Finally, we show that the processors get the token in a round-robin order. This property is used by our transformation in Section 3. The proof of Lemma 6 appears in [10].

Lemma 6. *If AllCompleteCycle holds, after P'_i releases the token, $P'_{(i \bmod n)+1}$ will be the next processor to get the token.*

We proved that a state where AllCompleteCycle holds is safe for the token-passing task. We want to see how many rounds are needed until AllCompleteCycle holds. Let $s(n)$ denote the maximum number of operations performed by a processor each time it has the token. We call these operations *token operations.* The time complexity of the token-passing algorithm is a function of $s(n)$.

Lemma 7. *If P'_i and P'_j label the same node and P'_i performs c cycles, then P'_j performs at least $c - 2$ cycles.*

Lemma 8. *In any execution of the Token-Passing algorithm, at least one write operation is performed during $s(n) + 2$ consecutive rounds.*

Proof. We assume the opposite and get a contradiction. If some processor executes a token operation during the first or the second round, then, it releases the token and performs a write by round $s(n)+2$. Therefore, all the processors must execute read operations during the first two rounds. But then the processors could read these same registers forever, which contradicts deadlock freedom. □

Next, we show that the token-passing algorithm stabilizes in $O(n \log^2 n)$ rounds, if a processor performs only one token operation when it gets the token.

Theorem 3. *The Token-Passing algorithm stabilizes within* $O(s(n)n(\log n)^2)$ *rounds.*

Proof. If we look at any path in the tree T, every two consecutive nodes are partially labeled by one common processor. Since the height of the tree is $h = \lceil \log_2 n \rceil$, the length of the simple path between any two leaves is at most $2h \in O(\log n)$. Thus, if any processor performs at least $k = 4h + 1$ cycles, then, by Lemma 7, every processor performs at least $k - 4h = 1$ cycles and, hence, AllCompleteCycle holds. Any processor partially labels at most $h - 1$ nodes in the tree (excluding its leaf). So, it performs at most $h - 1$ write operations per cycle. By Lemma 8, during the first $(s(n)+2)n(h-1)k \in O(s(n)n \log^2 n)$ rounds, at least $n(h-1)k$ writes have occurred. Hence, there is some processor P that has performed at least $(h-1)k$ write operations and, thus, at least k cycles. $\square$

5 Discussion

Our transformation presented in Section 3 works for the shared memory model with bounded register size and processor memory. Lynch's transformation (presented in Section 1.2) requires unbounded register size and unbounded processor memory. However, it has another good property that ours does not have: wait-freedom.

Each operation applied to a shared object has an invocation to the object and a response from the object. Attiya and Welch [4] define a transformation from a shared memory model to any model to be *wait-free* if the following holds: Let a' be a prefix of any execution produced by the transformation that includes an invocation by processor P_i', but not a matching response. Then, there is an extension of a' that only contains operations performed by P_i' in which the matching response is performed.

This definition does not directly apply to transformations from the message passing model because the operations of a message passing model do not have invocations and responses. To use this definition of wait-freedom for the message passing model, we assign to each operation of the message passing model an input operation that plays the role of an invocation and an output operation that plays the role of the response. In particular, let SEND(i,j,m) be the input operation to processor P_i that makes it send message m to its neighbour P_j, and let *ack* be the output operation that P_i performs when it finishes performing the send. Similarly, let RECEIVE(i,j) be the input operation to processor P_i that makes it receive a message (or $\perp$) from queue $L_{j,i}$ and let RECEIVED(i,j,m) be the response when P_i received message m from P_j (where $m = \perp$ if queue $L_{j,i}$ is empty). The restriction for the user interface is that P_i is a neighbour of P_j and that RECEIVE(i,j) operations are scheduled infinitely often for every i and j.

Our transformation is not wait-free because every processor has to wait for the token before it simulates sending and receiving operations. Lynch's transformation is wait-free because every processor can simulate send and receive operations in a constant number of its own steps, starting from any state. Next, we prove that there is no wait-free transformation from the message passing model $\mathcal{M}_{mp}$, to $\mathcal{M}_{bsm}$ the shared memory model with a finite number of registers of bounded size and processors with bounded memory.

Theorem 4. *There is no wait-free transformation of self-stabilizing algorithms from $\mathcal{M}_{mp}$ to $\mathcal{M}_{bsm}$.*

Proof. (Sketch) Assume that such a transformation exists. We construct an algorithm and an execution of its transformed version that has no suffix with the same trace as any execution of the original version. This contradicts the correctness of the transformation. We consider the input operations SEND(i,j,m), RECEIVE(i,j) and the output operations *ack*, RECEIVED(i,j,d), where d is either a message or $\perp$, as described above. The task of the problem is defined by traces with the property that there is a one-to-one correspondence between each RECEIVED(i,j,m) with $m \not\models\perp$ and a preceding SEND(j,i,m).

There is an easy self-stabilizing algorithm that solves this task in $\mathcal{M}_{mp}$. When processor P_i gets the input operation SEND(i, j, m), it sends message m to P_j and then performs the output operation *ack*. When processor P_i gets the input operation RECEIVE(i, j), it performs a receive from the queue $L_{j,i}$ and then performs RECEIVED(i,j,d), where d is either the message received or $\perp$, if $L_{j,i}$ is empty.

Now, consider the following execution a' of the transformed algorithm produced by the wait-free transformation in $\mathcal{M}_{bsm}$, starting from a state where all the simulated queues are empty. The adversary performs send and receive phases alternately, infinitely many times. During the send phase, it schedules SEND operations until an execution fragment with the following properties is created. The fragment starts and ends with the same state and it contains at least one SEND operation and its corresponding *ack*. Since $\mathcal{M}_{bsm}$ only has a finite number of shared registers of bounded size and bounded memory for each processor, there are only a finite number of different states for this model. Therefore, in any infinite sequence of states, at least one state has to be repeated infinitely often. During the receive phase, the adversary continuously schedules RECEIVE input operations until a corresponding RECEIVED(i,j,m) output operation has been performed for every message $m \not\models\perp$ sent by P'_j to P'_i so far in the execution. Each receive phase is completed in a finite number of steps. This follows by wait-freedom: every processor that gets a RECEIVE will perform the corresponding RECEIVED within a finite number of its own steps.

We denote the k^{th} send phase by F_k, s_k, X_k, s_k, where F_k is some execution fragment that contains SEND operations and their corresponding *ack* operations, s_k is a state, and X_k is an execution fragment that contains at least one SEND operations and its corresponding *ack*. We denote the k^{th} receive phase (as described above) by R_k. Then, execution $a' = F_1, s_1, X_1, s_1, R_1, F_2, s_2, X_2, s_2, R_2, ...$ We create another execution b' of the same (transformed) algorithm by eliminating the X_k execution fragments, for $k \geq 1$:

$b' = F_1, s_1, R_1, F_2, s_2, R_2, ...$ Note that, by construction of a', there is one-to-one correspondence between the RECEIVED(i,j,m) operations (with $m \not= \perp$) that happen in R_k and the preceding SEND(j,i,m) operations that happen in F_k and X_k, for $k \geq 1$. Since the X_k fragments are eliminated in b', in any suffix, there is a prefix with more RECEIVED(i,j,m) operations than SEND(j,i,m) operations for some i,j and $m \not= \perp$. Therefore, there is no suffix of b' with the same trace as any execution of the original whose traces belong in the task. □

Acknowledgments. A huge thank you to my supervisor Prof. Faith Fich for proposing the topic, directing my research, and spending uncountable hours of discussion with me.

References

1. Attiya. Efficient and robust sharing of memory in message-passing systems. In *WDAG: International Workshop on Distributed Algorithms*. LNCS, Springer-Verlag, 1996.
2. Attiya and Welch. Sequential consistency versus linearizability (extended abstract). In *SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991.
3. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. Technical Memo MIT/LCS/TM-423, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1992.
4. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill Publishing Company, May 1998. 6.
5. Amotz Bar-Noy and Danny Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In Piotr Rudnicki, editor, *Proceedings of the 8th Annual Symposium on Principles of Distributed Computing*, pages 301–318, Edmonton, AB, Canada, August 1989. ACM Press.
6. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the Association for Computing Machinery*, 17(11):643–644, November 1974.
7. Dolev, Israeli, and Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *DISTCOMP: Distributed Computing*, 7, 1994.
8. Shlomi Dolev. *Self-Stabilization.* MIT Press, Cambridge, MA, 2000. Ben-Gurion University of the Negev, Israel.
9. Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing*, 26(1):273–290, February 1997.
10. Kleoni Ioannidou. Self-Stabilizing Transformations Between Message Passing and Shared Memory Models. Master Thesis, Department of Computer Science of University of Toronto, 2001.
11. Katz and Perry. Self-stabilizing extensioins for message-passing systems. *DISTCOMP: Distributed Computing*, 7, 1994.
12. Lynch and Vaandrager. Forward and backward simulations for timing-based systems. In *REX: Real-Time: Theory in Practice, REX Workshop*, 1991.
13. Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CS, 1996. MIT.

Simple Wait-Free Multireader Registers[⋆]
(Extended Abstract)

Paul Vitányi

Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. paulv@cwi.nl.

Abstract. Multireader shared registers are basic objects used as communication medium in asynchronous concurrent computation. We propose a surprisingly simple and natural scheme to obtain several wait-free constructions of bounded 1-writer multireader registers from atomic 1-writer 1-reader registers, that is easier to prove correct than any previous construction. Our main construction is the first symmetric pure timestamp one that is optimal with respect to the worst-case local use of control bits; the other one is optimal with respect to global use of control bits; both are optimal in time.

1 Introduction

Interprocess communication in distributed systems happens by either message-passing or shared variables (also known as shared *registers*). Lamport [19] argues that every message-passing system must hide an underlying shared variable. This makes the latter an indispensable ingredient. The multireader wait-free shared variable is (even more so than the multiwriter construction) the building block of virtually all bounded wait-free shared-variable concurrent object constructions, for example, [28,15,5,8,6,7,13,2,16,24]. Hence, understanding, simplicity, and optimality of bounded wait-free atomic multireader constructions is a basic concern. Our constructions are really simple and structured extensions of the basic unbounded timestamp algorithm in [28], based on a natural recycling scheme of obsolete timestamps.

Asynchronous communication: Consider a system of asynchronous processes that communicate among themselves by executing read and write operations on a set of shared variables only. The system has no global clock or other synchronization primitives. Every shared variable is associated with a process (called *owner*) which writes it and the other processes may read it. An execution of a write (read) operation on a shared variable will be referred to as a *Write* (*Read*) on that variable. A Write on a shared variable puts a value from a pre determined finite domain into the variable, and a Read reports a value from the

⋆ This work was supported in part by the EU fifth framework project QAIP, IST–1999–11234, the NoE QUIPROCONE IST–1999–29064, the ESF QiT Programmme, and the EU Fourth Framework BRA NeuroCOLT II Working Group EP 27150.

domain. A process that writes (reads) a variable is called a *writer* (*reader*) of the variable.

Wait-free shared variable: We want to construct shared variables in which the following two properties hold. (1) Operation executions are not necessarily atomic, that is, they are not indivisible, and (2) every operation finishes its execution within a bounded number of its own steps, irrespective of the presence of other operation executions and their relative speeds. That is, operation executions are *wait-free*. These two properties give rise to a classification of shared variables: (i) A *safe* variable is one in which a Read not overlapping any Write returns the most recently written value. A Read that overlaps a Write may return any value from the domain of the variable; (ii) A *regular* variable is a safe variable in which a Read that overlaps one or more Writes returns either the value of the most recent Write preceding the Read or of one of the overlapping Writes; and (iii) An *atomic* variable is a regular variable in which the Reads and Writes behave as if they occur in some total order which is an extension of the precedence relation.

Multireader shared variable: A multireader shared variable is one that can be written by one process and read (concurrently) by many processes. Lamport [19] constructed an atomic wait-free shared variable that could be written by one process and read by one other process, but he did not consider constructions of wait-free shared variables with more than one writer or reader.

Previous Work: In 1987 there appeared at least five purported solutions for the wait-free implementation of 1-writer n-reader atomic shared variable from atomic 1-writer 1-reader shared variables: [18,27,4,26] and the conference version of [15], of which [4] was shown to be incorrect in [9] and only [26] appeared in journal version. The only other 1-writer n-reader atomic shared variable constructions appearing in journal version are [10] and the "projection" of the main construction in [21]. A selection of related work is [28,21,3,4,10,11,12,17,18,19, 20,21,23,24,25,26,27]. A brief history of atomic wait-free shared variable constructions and timestamp systems is given in [13].

Israeli and Li [15] introduced and analyzed the notion of *timestamp system* as an abstraction of a higher typed communication medium than shared variables. (Other constructions are [5,16,8,6,7,13].) As an example of its application, [15] presented a non-optimal multireader construction, partially based on [28], using order n^3 control bits overall, and order n control bits locally for each of $O(n^2)$ shared 1-reader 1-writer variables. Our constructions below are inspired by this timestamp method and exploit the limited nature of the multireader problem to obtain both simplification and optimality.

This work: We give a surprisingly simple and elegant wait-free construction of an atomic multireader shared variable from atomic 1-reader 1-writer shared variables. Other constructions (and our second construction) don't use pure timestamps [15] but divide the timestamp information between the writer's control bits and the reader's control bits. Those algorithms have an asymmetric burden on the writer-owned subvariables. Our construction shows that using pure timestamps we can balance the size of the subvariables evenly among all owners, obtaining a symmetric construction. The intuition behind our approach is given in Section 2. Technically:

(i) Our construction is a bounded version of the unbounded construction in [28] restricted to the multireader case (but different from the "projection" of the bounded version in [21]), and its correctness proof follows simply and directly from the correctness of the unbounded version and a proof of proper recycling of obsolete timestamps.

(ii) The main version of our construction is the first construction that uses a sublinear number of control bits, $O(\log n)$, locally *for everyone* of the n^2 constituent 1-reader 1-writer subvariables. It is easy to see that this is optimal locally, leading to a slightly non-optimal global number of control bits of order $n^2 \log n$. Implementations of wait-free shared variables assume atomic shared (control) subvariables. Technically it may be much simpler to manufacture in hardware—or implement in software—small $\Theta(\log n)$-bit atomic subvariables that are robust and reliable, than the exponentially larger $\Theta(n)$-bit atomic subvariables required in all previous constructions. A wait-free implementation of a shared variable is fault-tolerant against crash failures of the subvariables, but not against more malicious, say Byzantine, errors. Slight suboptimality is a small price to pay for ultimate reliability.

(iii) Another version of our construction uses n 1-writer 1-reader variables of n control bits and n^2 1-writer 1-reader variables of 2 control bits each, yielding a global $O(n^2)$ number of control bits. There need to be n^2 1-reader 1-writer subvariables for pairwise communication, each of them using some control bits. Hence the global number of control bits in any construction is $\Omega(n^2)$. We also reduce the number of copies of the value written to $O(n)$ rather than $O(n^2)$. All these variations of our construction use the minimum number $O(n)$ of accesses to the shared 1-reader 1-writer subvariables, per Read and Write operation.

Table 1. Comparison of Results.

Construction	Global control bits	Max local control bits
[18]	$\Theta(n^3)$	$\Theta(n)$
[27]	$\Theta(n^2)$	$\Theta(n)$
[4]-incorrect: [9]	$\Theta(n^2)$	$\Theta(n)$
[26]	$\Theta(n^2)$	$\Theta(n)$
[15]	$\Theta(n^3)$	$\Theta(n)$
[21]	$\Theta(n^2)$	$\Theta(n)$
[10]	$\Theta(n^2)$	$\Theta(n)$
This paper (ii)	$\Theta(n^2 \log n)$	$\Theta(\log n)$
This paper (iii)	$\Theta(n^2)$	$\Theta(n)$

1.1 Model, Problem Definition, and Some Notations

Throughout the paper, the n readers and the single writer are indexed with the set $I = \{0, \dots, n\}$—the writer has index n. The multireader variable constructed will be called ABS (for abstract).

A construction consists of a collection of atomic 1-reader 1-writer shared variables $R_{i,j}, i, j \in I$ (providing a communication path from user i to user j), and two procedures, *Read* and *Write*, to execute a read opeartion or write operation on ABS respectively. Both procedures have an input parameter i, which is the index of the executing user, and in addition, Write takes a value to be written to ABS as input. A return statement must end both procedures, in the case of Read having an argument which is taken to be the value read from ABS.

A procedure contains a declaration of local variables and a body. A local variable appearing in both procedures can be declared *static*, which means it retains its value between procedure invocations. The body is a program fragment comprised of atomic statements. Access to shared variables is naturally restricted to assignments from $R_{j,i}$ to local variables and assignments from local variables to $R_{i,j}$, for any j (recall that i is the index of the executing user). No other means of inter-process communication is allowed. In particular, no synchronization primitives can be used. Assignments to and from shared variables are called writes and reads respectively, always in lower case. The *space complexity* of a construction is the maximum size, in bits, of a shared variable. The *time complexity* of the Read or Write procedure is the maximum number of shared variable accesses in a single execution.

1.2 Correctness

A wait-free construction must satisfy the following constraint: **Wait-Freedom:** Each procedure must be free from unbounded loops. Given a construction, we are interested in properties of its executions, which the following notions help formulate. A *state* is a configuration of the construction, comprising values of all shared and local variables, as well as program counters. In between invocations of the Read and Write procedure, a user is said to be *idle*, and its program counter has the value 'idle'. One state is designated as *initial state*. All users must be idle in this state.

A state t is an *immediate successor* of a state s if t can be reached from s through the execution of a procedure statement by some user in accordance with its program counter. Recall that n denotes the number of readers of the constructed variable ABS. A state has precisely $n+1$ immediate successors: there is at precisely one atomic statement per process to be executed next (each process is deterministic).

A *history* of the construction is a finite or infinite sequence of states $t_0, t_1, \ldots$ such that t_0 is the initial state and t_{i+1} is an immediate successor of t_i. Transitions between successive states are called the *events* of a history. With each event is associated the index of the executing user, the relevant procedure statement, and the values manipulated by the execution of the statement. Each particular access to a shared variable is an event, and all such events are totally ordered.

The *(sequential) time complexity* of the Read or Write procedure is the maximum number of shared variable accesses in some such operation in some history.

An event a *precedes* an event b in history h, $a \prec_h b$, if a occurs before b in h. Call a finite set of events of a history an event-set. Then we similarly say that an event-set A precedes an event-set B in a history, $A \prec_h B$, when each event in A precedes all those in B. We use $a \preceq_h b$ to denote that either $a =_h b$ or $a \prec_h b$. The relation $\prec_h$ on event-sets constitutes what is known as an *interval order*. That is, a partial order $\prec$ satisfying the interval axiom $a \prec b \wedge c \prec d \wedge c \not\prec b \Rightarrow a \prec d$. This implication can be seen to hold by considering the last event of c and the earliest event of b. See [19] for an extensive discussion on models of time.

Of particular interest are the sets consisting of all events of a single procedure invocation, which we call an *operation*. An operation is either a Read operation or a Write operation. It is *complete* if it includes the execution of the final **return** statement of the procedure. Otherwise it is said to be *pending*. A history is complete if all its operations are complete. Note that in the final state of a complete finite history, all users are idle. The *value* of an operation is the value written to ABS in the case of a Write, or the value read from ABS in the case of a Read.

The following crucial definition expresses the idea that the operations in a history appear to take place instantaneously somewhere during their execution interval. A more general version of this is presented and motivated in [14]. To avoid special cases, we introduce the notion of a *proper* history as one that starts with an initializing Write operation that precedes all other operations. **Linearizability:** A complete proper history h is *linearizable* if the partial order $\prec_h$ on the set of operations can be extended to a total order which obeys the semantics of a variable. That is, each Read operation returns the value written by that Write operation which last precedes it in the total order. We use the following definition and lemma from [21]:

Definition 1. *A construction is* correct *if it satisfies Wait-Freedom and all its complete proper histories are linearizable.*

Lemma 1. *A complete proper history h is linearizable iff there exists a function mapping each operation in h to a rational number, called its timestamp, such that the following 3 conditions are satisfied:*

Uniqueness: *different Write operations have different timestamps.*

Integrity: *for each Read operation there exists a Write operation with the same timestamp and value, that it doesn't precede.*

Precedence: *if one operation precedes another, then the timestamp of the latter is at least that of the former.*

2 Intuition

We use the following intuition based on timestamp systems: The concurrent reading and writing of the shared variable by one writer and n readers (collectively, the *players*) is viewed as a pebble game on a finite directed graph. The nodes of the graph can be viewed as the timestamps used by the system, and a pebble on a node as a subvariable containing this timestamp. If the graph contains an arc pointing from node a to node b then a is *dominated* by b ($a < b$).

The graph has no cycles of length 1 or 2. First, suppose that every player has a single pebble which initially is placed at a distinguished node that is dominated by all other nodes. A Read or Write by a player consists in observing where the pebbles of the other players are, and determining a node to move its own pebble to. In a Write, the writer puts its pebble on a node that satisfies all of the following: (i) it dominates the previous position; (ii) it is not occupied by a pebble of a reader; and (iii) it is not dominated by a node occupied by a pebble of a reader. In a Read, the reader concerned puts its pebble at a node containing the writer's pebble. In the pebble game a player can only observe the pebble positions of the other players in sequence, one at a time. By the time the observation sequence is finished, pebbles observed in the beginning may have moved again. Thus, in an observation sequence by a reader, a node can contain another reader's pebble that dominates the node containing the writer's pebble. Then, the second reader has already observed the node pebbled by a later Write of the writer and moved its pebble there. In the unbounded timestamp solution below, where the timestamps are simply the nonnegative integers and a higher integer dominates a lower integer, this presents no problem: a reader simply moves its pebble to the observed pebbled node corresponding to the greatest timestamp. But in the bounded timestamp solution we must distinguish obsolete timestamps from active recycled timestamps. Clearly, the above scenario cannot happen if in a Read we observe the position of the writer's pebble last in the obervation sequence. Then, the linearization of the complete proper history of the system consists of ordering every Read after the Write it joined its pebble with. This is the basic idea. But there is a complication that makes life less easy.

In the implementation of the 1-writer n-reader variable in 1-writer 1-reader subvariables, every subvariable is a one-way communication medium between two processes. Since two-way communication between every pair of processes is required, there are at least $(n+1)n$ such subvariables: Every process owns at least n subvariables it can write, each of which is read by one of the other n processes. Thus, a pebble move of a player actually consists in moving (at least) n pebble copies, in sequence one at a time—a move of a pebble consists in atomically writing a subvariable. Every pebble copy can only be observed by a single fixed other player—the reader of that 1-writer 1-reader subvariable. This way player 0 may move the pebble copy associated with player 1 to another node (being observed by player 1 at time t_1) while the pebble copy associated with player 2 is still at the originating node (being observed later by player 2 at time $t_2 > t_1$). This once more opens the possibility that a reader sees a writer's pebble at a node pebbled by a Write, while another reader earlier on sees a writer's pebble at a node pebbled by a later Write. The following argument shows that this 'later Write' must be in fact the 'next Write': A reader always joins its pebble to a node that is already pebbled, and only the writer can move its pebbles to an unoccupied node. Therefore, the fact that a reader observes the writer's pebble last guaranties that no reader, of which the pebble was observed before, can have observed a Write's pebble position later than the next Write. Namely, at the time the writer's pebble was observed by the former reader at the node pebbled by

Write, the next Write wasn't finished, but the latter reader has already observed the writer's pebble. This fact is important in the bounded timestamp solution we present below, since it allows us to use a very small timestamp graph that contains cycles of length just 3: $G = (V, E)$ with $V = \{\bot, 1, \ldots, 4n-3\}^2$ and $(v_1, v_2) \in E$ $(v_1 < v_2)$ iff either $v_1 = (i, j), v_2 = (k, h)$ and $j = k$ and $i \neq h \neq \bot$, or $(i, j) = (\bot, \bot)$ and $h \neq \bot$.

This approach suffices to linearize the complete proper history of the system if we can prevent a Write to pebble the same node as a Read in the process of chasing an older obsolete Write. This we accomplish by a Read by player i announcing its intended destination node several times to the writer, with an auxiliary pebble for the purpose (special auxiliary subvariable) intended for this purpose, and in between checking whether the writer has moved to the same node. In this process either the Read discovers a Write that is intermediate between two Writes it observed, and hence covered by the Read, in which case it can safely report that Write and choose the bottom timestamp $(\bot, \bot)$. Alternatively, the Read ascertains that future Writes know about the timestamp it intends to use and those Writes will avoid that timestamp.

3 Unbounded

Figure 1 shows Construction 0, which is a restriction to the multireader case of the unbounded solution multiwriter construction of [28]. Line 2 of the Write procedure has the same effect as "*free* := *free* + 1" with *free* initialized at 0 (because the writer always writes *free* to $R_{n,n}.tag$ in line 4). The processes indexed $0, \ldots, n-1$ are the readers and the process indexed n is the writer. We present it here as an aid in understanding Construction 1. Detailed proofs of correctness (of the unrestricted version, where every process can write and not just process n) are given in [28] and essentially simple proofs in [21] and the textbook [22].

The timestamp function called for in lemma 1 is built right into this construction. Each operation starts by collecting value-timestamp pairs from all users. In line 3 of either procedure, the operation picks a value and timestamp for itself. It finishes after distributing this pair to all users. It is not hard to see that the three conditions of lemma 1 are satisfied for each complete proper history. Integrity and Precedence are straightforward to check. Uniqueness follows since timestamps of Write operations of the single writer strictly increase (based on the observation that each $R_{i,i}.tag$ is nondecreasing).

Restricting our unbounded timestamp multiwriter algorithm of [28], in the version of [21], to the single writer case enabled us to tweak it to have a new very useful property that is unique to the multireader case: The greatest timestamp scanned by a reader is either the writer's timestamp *from*$[n].tag$ or another reader's timestamp that is at most 1 larger. The tweaking consists in the fact that there is a definite order in the scanning of the shared variables in line 2 of the Read procedure: the writer's shared variable is scanned last (compare Section 2).

```
type $I : 0..n$
        $shared$ : record
                $value$ : ABS$type$
                $tag$ : integer
            end
```

```
procedure $Write(n, v)$
var $j : I$
    $free$ : integer
    $from$ : array$[0..n]$ of $shared$
begin
1   for $j := 0..n$ do $from[j] := R_{j,n}$
2   $free := \max_{j \in I} from[j].tag + 1$
3   $from[n] := (v, free)$
4   for $j := 0..n$ do $R_{n,j} := from[n]$
end
```

```
procedure $Read(i)$
var $j, max : I$
    $from$ : array$[0..n]$ of $shared$
begin
1   for $j := 0..n$ do $from[j] := R_{j,i}$
2   select $max$ such that $\forall j : from[max].tag \geq from[j].tag$
3   $from[i] := from[max]$
4   for $j := 0..n$ do $R_{i,j} := from[i]$
5   return $from[i].value$
end
```

Fig. 1. Construction 0

Lemma 2. *The* max *selected in line 2 of the Read procedure satisfies* $from[n].tag \leq from[max].tag \leq from[n].tag + 1$.

Proof. At the time the writer's shared variable $R_{n,i}$ is scanned last in line 2 of the $Read(i)$ procedure, yielding $from[n].tag$, the writer can have started its next write, the $(from[n].tag + 1)$th Write, but no write after that—otherwise the $from[n].tag$ would already have been overwritten in $R_{n,i}$ by the $(from[n].tag + 1)$th Write. Hence, a timestamp scanned from another reader's shared variables $R_{j,i}$ $(j \neq n, i)$ can exceed the writer's timestamp by at most 1. □

4 Bounded

The only problem with Construction 0 is that the number of timestamps is infinite. With a finite number of timestamps comes the necessity to re-use timestamps and hence to distinguish old timestamps from new ones.

Our strategy will be as follows. We will stick very close to construction-0 and only modify or expand certain lines of code. The new bounded timestamps will consist of two fields, like dominoes, the *tail* field and *head* field.

Definition 2. *A* bounded timestamp *is a pair (p, c) where p is the value of the* tail *field and c is the value of the* head *field. Every field can contain a value t with $0 \leq t \leq 4n + 2$ or t is the distinguished initial, or bottom, value $\bot$. ($\bot$ is not a number and is lower than any number.) We define the domination relation "$<$" on the bounded timestamps as follows: $(p_1, c_1) < (p_0, c_0)$ if either $c_1 = p_0$ and $p_1 \neq c_0 \neq \bot$, or $p_1, c_1 = \bot$ and $c_0 \neq \bot$.*

The matrix of shared variables stays the same but for added shared variables $R_{i,n+1}$ for communication between readers i ($0 \leq i \leq n - 1$) and the writer n. These shared variables are used by the readers to inform the writer of all timestamp values that are not obsolete, and hence cannot be recycled yet. The interesting part of the later proof is to show that this statement is true. The scan executed in line 1 of the Write protocol gathers all timestamps written in the $R_{i,n}$'s ($0 \leq i \leq n$) and $R_{i,n+1}$'s ($0 \leq i \leq n-1$), the timestamps that are not obsolete, and hence the process can determine the $\leq 4n + 2$ values occurring in the fields (two per timestamp), and select a value that doesn't occur (there are $4n+3$ values available exclusive of the bottom value $\bot$). The initial state of the construction has all $R_{i,j}$ containing $(0, \bot, \bot)$.

The lines of Construction-1 are numbered maintaining—or subnumbering—the corresponding line numbers of Construction-0. The only difference in the Write protocols is line 2 (select new timestamp). In the Read protocols the differences are lines 2.x (determine latest—or appropriate—timestamp) and lines 3.x (assign selected timestamp to local variable *from*[*i*]), and lines 0.x (start Read by reading writer's timestamp and writing it back to writer). According to this scheme we obtain the out-of-order line-sequence 2.4, 3.1, 2.5, 3.2 because the instructions in lines 2 and 3 of Construction-0 are split and interleaved in Construction-1.

Lemma 3. *Line 2 of a Write always selects a free integer c_1 such that the timestamp (p_1, c_1) with new head c_1 and new tail p_1 (head of the timestamp of the directly preceding Write), assigned in line 3 and written in line 4 of that Write, satisfies $(p_1, c_1) \not< (p_0, c_0)$ and $(p_1, c_1) \neq (p_0, c_0)$ for every timestamp (p_0, c_0) either scanned in line 1 of the Write, or presently occurring in a local variable of a concurrent Read that will eventually be written to a shared variable in line 4 of that Read.*

Proof. In line 1 of a Write the shared variables with each reader, and a redundant shared variable with itself, are scanned in turn. In assigning a value to *free* in line 2 the writer avoids all values that occur in the *head* and *tail* fields of the shared variables. All local variables of a reader are re-assigned from shared variables in executing the Read procedure, before they are written to shared variables. So the only problem can be that *free* occurs in a local variable of a concurrently active Read that has not yet written it to a variable shared with the writer at the time that variable was scanned by the Write. If *free* exists in a local variable *temp* this doesn't matter since the timestamp concerned will not be used as a Read timestamp. Hence, the only way in which *free* can be assigned a value that concurrently exists in a local variable of a Read which already has

```
        $shared$ : record
                $value$ : ABS$type$
                $tail$ : $\perp, 0..4n+2$
                $head$ : $\perp, 0..4n+2$
            end
```

```
procedure $Write(n, v)$
var $j : 0..n$
    $free : 0..4n+2$
    $from$ : array$[0..n]$ of $shared$
    $notfree$ : array$[0..n-1]$ of $shared$
begin
1   for $j := 0..n$ do $from[j] := R_{j,n}$; for $j := 0..n-1$ do $notfree[j] := R_{j,n+1}$
2   $free :=$ least positive integer not in $tail$ or $head$ fields of $from$ or $notfree$
            records
3   $from[n] := (v, from[n].head, free)$
4   for $j := 0..n$ do $R_{n,j} := from[n]$
end
```

```
procedure $Read(i)$
var $j, max$
    $temp$ : $shared$
    $from$ : array$[0..n]$ of $shared$
begin
0.1 $temp := R_{n,i}$
0.2 $R_{i,n+1} := temp$
1   for $j = 0..n$ do $from[j] := R_{j,i}$
2.1 if $from[n] \not\models temp$ then
2.2         $temp, R_{i,n+1} := from[n]$
2.3         for $j = 0..n$ do $from[j] := R_{j,i}$
2.4         if $from[n] \not\models temp$ then
3.1             $from[i] := (temp.value, \perp, \perp)$; goto 4
2.5 if $\exists max : from[max].(tail, head) > from[n].(tail, head)$ then
3.2         $from[i] := from[max]$
3.3 else $from[i] := from[n]$
4   for $j := 0..n$ do $R_{i,j} := from[i]$
5   return $from[i].value$
end
```

Fig. 2. Construction 1

been used or eventually can be used in selection line 2.5 of a Read, and hence eventually can be part of an offending timestamp written to a shared variable, is according to the following scenario: A $Read(i)$, say R, is active at the time a Write, say W_1, reads either one of their shared variables. This R will select or has already selected the offending timestamp (p_0, c_0), originally written by a Write W_0 preceding W_1, but R has not yet written it to $R_{i,n+1}$ or $R_{i,n}$ by the times W_1 scans those variables. Moreover, W_1 will in fact select a timestamp

(p_1, c_1) with either $(p_1, c_1) < (p_0, c_0)$ or $(p_1, c_1) = (p_0, c_0)$. This can only happen if *free* in line 2 is p_0, or c_0 and the $p_0 = \mathit{head}$ of the timestamp of the Write immediately preceding W_1, respectively. Then, a later Read R_0 using in its line 2.5 the (p_1, c_1) timestamp written by W_1 and the (p_0, c_0) timestamp written by R concludes falsely that W_1 precedes W_0 or $W_1 = W_0$. For this to happen, R has to write (p_0, c_0) in line 4, and has to assign it previously in one of lines 3.1, 3.2, or 3.3. In line 3.1 the timestamp assigned is the bottom timestamp $(\bot, \bot)$ which by definition cannot dominate or equal a timestamp assigned by the writer. This leaves assignement of the offending timestamp (p_0, c_0) in line 3.2 or line 3.3. Without loss of generality, assume that the domination and equality of the timestamps used by R in line 2.5 is still correct (so R is a “first” Read writing an offending timestamp).

Case 1: Timestamp (p_0, c_0) is assigned in line 3.2 of R and hence won the competition in line 2.5 by $\mathit{from}[n].(\mathit{tail}, \mathit{head}) < (p_0, c_0)$. Let W be the Write that wrote $\mathit{from}[n].(\mathit{tail}, \mathit{head})$. We first show that W directly precedes W_0. Since $\mathit{from}[n].(\mathit{tail}, \mathit{head}) < (p_0, c_0)$ we have $W \neq W_0$. If W follows W_0, then the conclusion from $\mathit{from}[n].(\mathit{tail}, \mathit{head}) < (p_0, c_0)$ that W_0 is the next Write after W is false (and (p_0, c_0) has already been written by a $\mathit{Read}(j)$, $j \neq i$, to the shared variable $R_{j,i}$ from which R scanned it). This contradicts the assumption above that domination and equality of timestamps used by R in line 2.5 is still correct. The only remaining possibility is that W precedes W_0. Of the timestamps used in the competition in line 2.5, the timestamp written by W was read last. Therefore, with W preceding W_0, W directly precedes W_0 and W wrote a timestamp $(\cdot, p_0)$, as is in fact evidenced by the relation $\mathit{from}[n].(\mathit{tail}, \mathit{head}) < (p_0, c_0)$.

Case 1.a: R executes line 2.5, and the test in line 2.1 was ‘true’—a Write wrote shared variable $R_{n,i}$ in between line 0.1 and line 2.1—and the test in line 2.4 was ‘false’—no Write wrote $R_{n,i}$ in between line 1 and line 2.4—and $\mathit{from}[n].(\mathit{tail}, \mathit{head})$ used in line 2.5 is the one read in line 1 and written by Write W in between line 0.1 and line 2.1. Therefore, Write W_0 started after line 0.1 but before line 2.4 (since (p_0, c_0) came by way of a $\mathit{Read}(j)$ with $j \neq i$ in line 2.3), and W_0 had not yet written $R_{n,i}$ before line 2.4. Then, every Write following W_0 started after line 2.3. As a consequence, such a Write both reads the timestamp written to $R_{i,n+1}$ in line 2.2 of R, and avoids choosing $\mathit{free} := p_0$ $(= \mathit{from}[n].\mathit{head}$ originating from $W)$ in its own line 2 (until $R_{i,n+1}$ is overwritten again). Hence the *tail* value in (p_0, c_0) is not re-used and the timestamp is actually not offensive. (Since $c_0 \neq p_0$ by the selection of *free* in line 2 of the Write protocol, all Writes following W_0 cannot create a timestamp $(\cdot, p_0)$ and hence also not (p_0, c_0).)

Case 1.b: R executes line 2.5, and the test in line 2.1 was ‘false’—no Write wrote shared variable $R_{n,i}$ in between line 0.1 and line 2.1—and $\mathit{from}[n].(\mathit{tail}, \mathit{head})$ used in line 2.5 is the one read in line 1 and written by Write W in between line 0.1 and line 2.1. The remainder of the argument is the same as in Case 1.a.

Case 2: Timestamp (p_0, c_0) is assigned in line 3.3 of R.

Case 2.a: Timestamp (p_0, c_0) was already scanned in line 0.1 of R, and again in line 1, and written by a Write W_0 writing the shared variable $R_{n,i}$ before the scan of line 0.1. If W_1 is the next Write after W_0, then every Write following W_1 will read (p_0, c_0) and avoid both its values in selecting *free* in line 2 and assigning the *head* of its new timestamp in line 3, because (p_0, c_0) was written in line 0.2 to $R_{i,n+1}$ before W_1 finished (and hence before a later Write started)—otherwise the timestamp of W_1 would have been observed in the scan of line 1. But W_1 will do this as well, since (p_0, c_0) is the timestamp written by W_0, and hence observed by W_1. Consequently, no value in (p_0, c_0) is re-used as the *head* of a new timestamp by a Write (until $R_{i,n+1}$ is overwritten again and the timestamp disappears) and (p_0, c_0) is actually not offensive.

Case 2.b: Timestamp (p_0, c_0) was first scanned in line 1 of *Read*(i)—the contents of $R_{n,i}$ scanned in line 0.1 is different from that scanned in line 1. Therefore, (p_0, c_0) was written by Write W_0 writing the shared variable $R_{n,i}$ after the scan at line 0.1. The timestamp scanned in line 2.3 was still the one written by W_0 (otherwise the test in line 2.4 would be positive and result in assignment 3.1 that doesn't assign the offending timestamp at all). The remainder of this case is identical with last part of Case 2.a starting from "If W_1 is the next Write after W_0". □

The crucial feature that makes the bounded algorithm work is the equivalent of lemma 2:

Lemma 4. *The timestamp selected in lines 2.x and written in line 4 of a Read*(i) *is one of the following:*

(i) The timestamp $(\perp, \perp)$ *for a Write that is completely overlapped by the Read concerned;*

(ii) Another reader's timestamp scanned in line 1 (respectively, line 2.3) that is written by the next Write after the Write that wrote the timestamp from$[n]$*.*$(tail, head)$ *scanned in line 1.*

(iii) Otherwise, the writer's timestamp from$[n]$*.*$(tail, head)$ *scanned in line 1.*

Proof. Only assignments in lines 3.1, 3.2, 3.3 can result in a write in line 4.

(i) If line 3.1 is executed in a Read, then previously we scanned the writer's timestamp in lines 0.1, 1, 2.3, and obtained three successive timestamps without two successive ones being the same. Hence the Write corresponding to the middle scan, of line 1, is overlapped completely by the Read, and the Read can be ordered directly after this Write, and this has no consequences for the remaining ordering. This is reflected by using the timestamp $(\perp, \perp)$ to be written by such a Read in line 4.

(ii) If line 3.2 is executed in a Read then the timestamp assigned is a reader's timestamp scanned in line 1 or line 2.3. The writer's timestamp used in the comparison line 2.5 is, say, (t, h). According to the semantics of the "$<$" sign in definition 2, only a reader's timestamp of the form $(h, free)$ satisfies the condition in line 2.5. The only timestamps in the system are created by the writer. By Lemma 3, existence of the $(h, free)$ timestamp somewhere in the system at the time of writing the current instance of timestamp (t, h) would have prevented the writer from writing (t, h). Thus the writer must have written the $(h, free)$ timestamp after it wrote (t, h). Using Lemma 3 a second time, the writer can write a timestamp with h in the *tail* field only at the very next Write after the Write that wrote a timestamp with h in the *head* field, since the (t, h) timestamp is still somewhere in a shared variable or to be written to a shared variable.

(iii) If line 3.3 is executed in a Read, then items (i) and (ii) were not applicable and the timestamp assigned is the writer's timestamp scanned in line 1. □

Theorem 1. *Construction-1 is a wait-free implementation of an atomic 1-writer n-reader register. It uses* $(n + 1)(n + 2) - 1$ *atomic 1-writer 1-reader* $2 \log(4n + 4)$ *bits control shared variables. The Write scans* $2n + 1$ *of these variables and writes* $n + 1$ *of them; the Read scans* $\leq 2n + 3$ *of these variables and writes* $\leq n + 3$ *of them.*

Proof. **Complexity:** Since the program of Construction-1 contains no loops, it is straightforward to verify that every Read and Write executes the number of accesses to shared variables, and the size of the shared variables, as in the statement of the theorem.

Wait-Freedom: This follows directly from the upper bounds on the complexity (scans and writes) of the shared variables in the construction, and the fact that the program of Construction-1 is loop-free.

Linearizability: Consider a complete proper history (as defined before) h of all Writes and only those Read's that don't write the bottom timestamp $(\perp, \perp)$ in line 4. Let $\prec_h$ be the partial order induced by the timing of the Reads and Writes of h. Lemma 4, items (ii) and (iii), asserts that on this history h, Construction-0 and Construction-1 behave identically in that the same Read reports the values of the same Write in both constructions. Therefore, with respect to h, linearizability of Construction-1 follows from the linearizability of Construction-0. Let $\prec_h^l$ be the linear order thus resulting from $\prec_h$. The remaining Read's, those that write the bottom timestamp $(\perp, \perp)$ in line 4, completely overlap a Write, lemma 4 item (i), and report the value of that overlapped Write. Hence they can, without violating linearizability, be inserted in the $\prec_h^l$-order directly following the Write concerned. This shows that Construction-1 is linearizable. □

Minimum Number of Global Control Bits: The same algorithm with only $O(n^2)$ control bits overall can be constructed as follows. Each register owned by the writer contains $2n$ control bits, and each register owned by a reader contains only 2 control bits. The control bits are used to determine the domination relation between readers and the writer. The Protocol stays the same, only the decisions in the protocol are made according to different format data. Since the decisions are isomorphic with that of Protocol 1, the correctness of the new Protocol follows by induction on the total atomic order of the operation executions in each run by the correctness of Construction-1.

Minimum Number of Replicas of Stored Values: In the algorithm, each subregister ostensibly contains a copy of the value to be written. This sums up to $O(n^2 \log V)$ bits, for the value ranging from 1 to V. With the following scheme only the registers owned by the writer contain the values. Each register owned by the writer can contain two values. The two fields concerned are used alternatingly. The writer starts its tth write with an extra write (line 0) to all registers it owns, writing the new value in field $t \pmod 2$. In line 4 it writes to field $t \pmod 2$ (it marks this field as the last one written), and finishes by setting $t := (t + 1) \pmod 2$. The readers, on the other hand, now write no values, only the timestamps. If the reader chooses the writer, it takes the value from the marked field; if it chooses a reader, it takes the value from the unmarked field. Since no observed reader can be more than one write ahead of the actually observed write, this is feasible while maintaining correctness. This results in $O(n)$ replications of the value written resulting in a total of $O(n \log V)$ value bits. This is clearly the optimal order since the writer needs to communicate the value to everyone of the readers through a separate subregister. (Consider a schedule where every reader but one has fallen asleep indefinitely. Wait-freeness requires that the writer writes the value to a subregister being read by the active reader.)

Acknowledgment. I thank Ming Li and Amos Israeli for initial interactions about these ideas in the late eighties. Other interests prevented me from earlier publication. I also thank Grisha Chockler for pointing out a problem with the previous formulation of Lemma 3, and meticulously reading earlier drafts and providing useful comments.

References

1. J. Anderson, Multi-Writer Composite Registers, *Distributed Computing*, 7:4(1994), 175–195.
2. K. Abrahamson, On achieving consensus using a shared memory. In *Proc. 7th ACM Symp. Principles Distribut. Comput.*, 1988, 291–302.
3. B. Bloom, Constructing two-writer atomic registers. *IEEE Trans. on Computers*, 37(1988), 1506–1514.
4. J.E. Burns and G.L. Peterson, Constructing multi-reader atomic values from non-atomic values. In *Proc. 6th ACM Symp. Principles of Distributed Computing.* 1987, 222–231.
5. D. Dolev and N. Shavit, Bounded concurrent time-stamp systems are constructible. *Siam J. Computing*, 26:2(1997), 418–455.
6. C. Dwork and O. Waarts, Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *J. Assoc. Comput. Mach.*, 46:5(1999), 633–666.
7. C. Dwork, M. Herlihy, S. Plotkin and O. Waarts, Time-Lapse snapshots. *SIAM J. Computing*, 28:5(1999), 1848–1874.
8. R. Gawlick, N.A. Lynch, and N. Shavit, Concurrent timestamping made simple. In *Proc. Israeli Symp. Theory Comput. and Systems*, LNCS 601, Springer-Verlag, 1992, 171–183.
9. S. Haldar and K. Vidyasankar, Counterexamples to a one writer multireader atomic shared variable construction of Burns and Peterson. *ACM Oper. Syst. Rev.*, 26:1(1992), 87–88.
10. S. Haldar and K. Vidyasankar, Constructing 1-writer multireader multivalued atomic variables from regular variables. *J. Assoc. Comput. Mach.*, 42:1(1995), 186–203.

11. S. Haldar and K. Vidyasankar, Buffer-optimal constructions of 1-writer multireader multivalued atomic shared variables. *J. Parallel Distr. Comput.*, 31:2(1995), 174–180.
12. S. Haldar and K. Vidyasankar, Simple extensions of 1-writer atomic variable constructions to multiwriter ones. *Acta Informatica*, 33:2(1996), 177–202.
13. S. Haldar and P. Vitanyi, Bounded concurrent timestamp systemss using vector clocks, *J. Assoc. Comp. Mach.*, 49:1(2002), 101–126.
14. M. Herlihy and J. Wing, Linearizability: A correctness condition for concurrent objects. *ACM Trans. Progr. Lang. Syst.*, 12:3(1990), 463–492.
15. A. Israeli and M. Li, Bounded time-stamps. *Distributed Computing*, 6(1993), 205–209.
16. A. Israeli and M. Pinhasov, A concurrent time-stamp scheme which is linear in time and space. In *Proc. 6th Intn'l Workshop Distribut. Alg.*, LNCS 647, Springer-Verlag, Berlin, 1992, 95–109.
17. A. Israeli and A. Shaham, Optimal multi-writer multireader atomic register. In *Proc. 11th ACM Symp. Principles. Distr. Comput.*, 1992, 71–82.
18. L.M. Kirousis, E. Kranakis, and P.M.B. Vitányi, Atomic multireader register. In *Proc. 2nd Intn'l Workshop Distr. Alg.*. LNCS 312, Springer-Velag, Berlin, 1987, 278–296.
19. L. Lamport, On interprocess communication — Part I: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1:2(1986), 77–101.
20. M. Li and P.M.B. Vitányi, Optimality of wait-free atomic multiwriter variables, *Inform. Process. Lett.*, 43:2(1992), 107–112.
21. M. Li, J.T. Tromp, and P.M.B. Vitányi, How to share concurrent wait-free variables. *J. Assoc. Comput. Mach.*, 43:4(1996), 723–746.
22. N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1997.
23. G.L. Peterson, Concurrent reading while writing. *ACM Trans. Progr. Lang. Systems*, 5:1(1983), 56–65.
24. G.L. Peterson and J.E. Burns, Concurrent reading while writing II: The multiwriter case. In *Proc. 28th IEEE Symp. Found. Comput. Sci.*, 1987, 383–392.
25. R. Schaffer, On the correctness of atomic multiwriter registers. Report MIT/LCS/TM-364, 1988.
26. A.K. Singh, J.H. Anderson and M.G. Gouda, The elusive atomic register. *J. Assoc. Comput. Mach.*, 41:2(1994), 311–339.
27. R. Newman-Wolfe, A protocol for wait-free, atomic, multi-reader shared variables. In *Proc. 6th ACM Symp. Princples Distribut. Comput.*, 1987, 232–248.
28. P.M.B. Vitányi and B. Awerbuch, Atomic shared register access by asynchronous hardware. In *Proc. 27th IEEE Symp. Found. Comput. Sci.*, 1986, 233–243. Errata: *Ibid.*, 1987, 487.

An Efficient Universal Construction for Message-Passing Systems
(Extended Abstract)

Partha Dutta[1], Svend Frølund[2], Rachid Guerraoui[1], and Bastian Pochon[1]

[1] Distributed Programming Laboratory, Swiss Federal Institute of Technology, Lausanne, CH 1015
[2] Hewlett-Packard Laboratories, Palo Alto, CA 94304

Abstract. A universal construction is an algorithm that transforms any object with a sequential specification into a wait-free linearizable implementation of that object. This paper presents a novel universal construction algorithm for a message-passing system with process crash failures. Our algorithm relies on two fine-grained underlying abstractions: a weak form of leader election, and a one-shot form of register.
Our algorithm is *indulgent*, *efficient* and *generic*. Being indulgent intuitively means that the algorithm preserves consistency even if the underlying system is asynchronous for arbitrary periods of time. Compared to other indulgent universal constructions, our algorithm uses fewer messages and gives rise to less work in steady-state. Our algorithm is generic in two senses: (1) although it is devised for a crash-stop model, it can be easily ported to various crash-recovery models, and (2) although it is optimized for steady-state periods, it can easily be extended to trade-off between steady-state performance and fail-over time.

1 Introduction

A universal construction is an algorithm that provides a wait-free and linearizable implementation of any object that has a sequential specification [12]. In short, being wait-free requires the implemented object to be highly available—any invocation of the object must complete in a finite number of steps, even in the presence of failures. Being linearizable intuitively means that the implemented object must remain consistent—the object must appear to be accessed in a sequential manner [13]. It is very appealing to use the notion of universal construction as the theoretical underpinning of highly-available distributed systems. The notion of universal construction clearly and precisely defines the contractual obligations and guarantees of the various players, such as the objects, the algorithm, and the clients. The object can be implemented in any way that complies with its sequential specification. In particular, objects can be non-deterministic. Clients are given precise safety and liveness guarantees. The universal construction algorithm can be based on any implementation that provides clients with wait-free, linearizable access to the object. From a practical point of view, the object represents an online service. A universal construction algorithm can be viewed as middleware that implements highly-available access to the service from a number of clients.

To be practical as the foundation for high-availability middleware, a universal construction algorithm should have a number of desirable properties:

– It should tolerate arbitrary asynchrony periods of the underlying system (we refer to this property as indulgence [11]). Indulgence is important because the service may be subject to unpredictable workloads, and it may share resources, such as network bandwidth, with other online services.
– The steady-state behavior should be efficient. Steady-state is a period where no process fails or is suspected to have failed. In most systems, this is the common case, and thus the case for which we want to optimize.
– The algorithm should minimize the communication between clients and the service. It is indeed common for online services to be accessed via the Internet, and such access typically involves communication over wide-area network links.

Traditionally, universal construction algorithms were devised in a shared-memory model, where processes communicate through shared registers and consensus-like objects [12]. One can indeed emulate a register abstraction using message-passing (e.g., assuming a majority of correct processes [2]). However, such a solution does not take full advantage of the message-passing model, and therefore, is not efficient in practice. Emulating atomic registers leads to a universal construction that requires $\Omega(n^2)$ messages, instead of $O(n)$ in our algorithm.

Primary-backup algorithms, such as [1,3,5], are indeed universal constructions devised with a message-passing model in mind. However, these algorithms rely on a strong form of leader election that make them non-indulgent. More precisely, they rely on the assumption of a single primary, and asynchrony in the underlying system may violate this assumption. The semi-passive replication algorithm [8] can be viewed as an indulgent primary-backup universal construction. Nevertheless, because semi-passive replication relies on an underlying consensus-like abstraction, it increases the number of messages exchanged between clients and a replicated service, as compared to traditional primary-backup algorithms. With primary-backup algorithms, a client sends its request to the primary, whereas with semi-passive replication, a client must send its request to all replicas. As we pointed out, the number of messages exchanged between a service and its clients is an important metric for a service that is accessed via the Internet.

This paper presents an indulgent universal construction algorithm for a message-passing model with process crash failures,[1] using two fine-grained underlying abstractions: a weak form of leader election, $\Diamond$Leader (denoted Ω in [6]),

[1] In our context of message-passing, we say that an object is *wait-free* if any client always returns from the invocation of an operation on this object within a finite number of its own steps independently of the crash of other clients, and despite the failure of a minority of replicas.

and a new one-shot form of register, $\triangle$Register. Neither of these can, in isolation, implement consensus, but their combined power is equivalent to consensus.[2]

Our algorithm follows the primary-backup replication pattern, but is more efficient than other indulgent primary-backup algorithms: our algorithm uses fewer messages and gives rise to less work by the backups. The latency of our algorithm matches the lower bound established in [5] for a non-indulgent solution. The message complexity (number of messages exchanged to process a request) is $2n + 2$ in our case, just like traditional primary-backup. In contrast, the message complexity of semi-passive replication is $5n$.

In our algorithm, only primaries update their state, backups only witness the updates performed by primaries. Thus, only a primary has the current state of the replicated object. A backup "constructs" the current state only if and when it becomes primary. Because backups do not update their state, they play an even more passive role in our algorithm than in traditional primary-backup algorithms that seek to keep backups up-to-date. Because a client only needs to send its requests to the primary, we simultaneously achieve the low message-complexity of the primary-backup approach and the low time-complexity of a lazy replication algorithm. Furthermore, our abstractions lead to a simple, comprehensive, and extensible universal construction algorithm. Indeed, the main idea behind our algorithm is to move work from steady-state periods to transition periods. Thus, the trade-off of our algorithm is to optimize the performance in steady state at the expense of making fail-over more costly. If failures are rare and the fail-over time not that critical, this is likely to be a good trade-off. However, in certain environments, it is very important to react quickly to failures. The good news is that our algorithm allows us to trade-off between steady-state performance and fail-over time in a modular manner, i.e., by *extending* the algorithm, not *changing* it. We build our $\triangle$Register abstraction on top of a more basic notion of register, a *round-based* register, denoted $\mathcal{R}$register, for which we give a precise specification. Interestingly, only the $\mathcal{R}$register needs to be changed for the $\triangle$Register implementation (and the universal construction) to operate in various distributed system models [9].

The rest of the paper is organized as follows. Section 2 defines the underlying system model. Section 3 introduces our abstractions. Section 4 gives our universal construction algorithm based on these abstractions. Section 5 analyzes some execution scenarios. Section 6 considers the genericity of our universal construction. Section 7 discusses our abstractions and our algorithm. Due to space limitation, details on our register implementation, including optimized algorithms and correctness proofs, have been omitted from this extended abstract but are included in the full version of the paper [9].

[2] Our weak leader election encapsulates the synchrony assumption needed to implement consensus, whereas our register encapsulates the assumption of a majority of correct processes needed for indulgent consensus.

2 Model

2.1 Processes, Communication, and Time

We represent a distributed system as a finite set of processes $\Pi = \{p_1, \ldots, p_n\}$. Processes fail by crashing—we do not deal with Byzantine failures, nor do we assume that processes recover after a crash. A process is correct if it does not fail.

Processes communicate by message passing. A message can be sent by the primitive send and received by the primitive receive. Message passing is reliable in the following sense: (*validity*) if a correct process sends a message to a correct process, the message is eventually received, (*no duplication*) each message is received at most once, and (*integrity*) the network does not create nor corrupt messages.

We assume an asynchronous distributed system. We indirectly introduce synchrony assumptions through the properties of our $\Diamond$Leader abstraction (Sect. 3), and we use a notion of time to specify the properties of our abstractions. To this end, we assume the presence of a discrete global clock with the set of natural numbers as tick range. The purpose of the clock is to simplify the presentation: processes cannot access this global clock.

We sub-divide the set Π of processes into two disjoint subsets: Client and Server. Processes in Server (*servers*) collectively implement a wait-free linearizable object that processes in Client (*clients*) can access.

2.2 Objects

An object has an internal state and a number of actions to manipulate this state. An action takes an input value and produces an output value. As a side-effect of producing the output value, the action may also update the internal state. Actions may be non-deterministic. That is, the side-effect and output value of a specific action may not be the same each time we execute it, even if we execute it in the same initial state and with the same input value.

The goal of our algorithm is to implement wait-free and linearizable access to any given object. Each server has its own copy of the given object. The algorithm uses two primitive and generic actions in dealing with objects:

– The execute primitive action takes a request (action name and input value) and returns an output value and an update value. The execute primitive executes the action on the given input. The returned update value captures the state update performed by executing the action. The primitive does not change the internal state of the object.
– The update primitive takes an update value, and performs the state update captured by the update value.

These two primitives separate the purely functional aspect of an object (mapping input to output) and the state-update aspect (using an input value to update the internal state).

3 Abstractions

Our universal construction is based on two fundamental abstractions: an eventual leader election, denoted $\Diamond$Leader and a one-shot form of register, denoted $\triangle$Register. $\Diamond$Leader encapsulates the synchrony assumptions needed to ensure wait-freedom. It is in this sense a *liveness* abstraction. $\triangle$Register encapsulates a convenient form of storage to ensure linearizability. It is in this sense a *safety* abstraction.

3.1 $\Diamond$Leader

$\Diamond$Leader eventually elects a unique and correct leader. The abstraction has one operation, called elect(). This operation does not take any input parameters. It returns an output parameter, which is the identity of a process. When p_i invokes elect() and gets p_j as an output at some time t, we say that p_i *elects* p_j at t. (We also say that p_j is *leader* (for p_i) at time t.) We define the specification of $\Diamond$Leader through the following properties.

Agreement: There is a time after which no two correct processes elect two different leaders.
Validity: There is a time after which every leader is correct.
Termination: After a process invokes elect(), either the process crashes or it eventually returns from the invocation (wait-free).

Note that our specification does not preclude the existence of concurrent leaders for arbitrary periods of time: hence the notion of *eventual* leader. In the following, we assume the existence of the $\Diamond$Leader abstraction[3] and refer to [6] for its implementation.

3.2 $\triangle$Register

Roughly speaking, our $\triangle$Register is a one-shot register, in the sense that (1) once a value is successfully written, it remains forever in the register, and (2) a write operation is guaranteed to succeed, however, if only a single process is writing, and an infinite number of times. Our $\triangle$Register is different from Lamport's notion of registers, because (1), in face of concurrency, any invocation on a $\triangle$Register may abort without returning any value and (2), at most one value can be successfully written in a $\triangle$Register.

Formally, $\triangle$Register has a single primitive, propose(). When a process p invokes propose() with a single argument $v \in \mathsf{Values}$ such that $\mathsf{abort} \notin \mathsf{Values}$, we say that *p proposes v*. The propose() primitive returns a value in $\mathsf{Values} \cup \{\mathsf{abort}\}$. If p returns from propose() with $v' \neq \mathsf{abort}$, we say that *p decides v′* and that the value v' returned is a *decision*. Otherwise, if propose() returns abort, we say that *p aborts*. If p proposes v and decides v, we say that *p commits v*. We now give the properties of $\triangle$Register:

[3] As already pointed out, $\Diamond$Leader corresponds to the failure detector Ω of [6]. In the terminology of [6], this is the weakest failure detector to solve consensus.

Validity: If a process decides a value v, then v was proposed by some process.
Agreement: No two processes decide differently.
Termination: If a process proposes, it either crashes or returns (wait-free). If only a single process proposes an infinite number of times, and if this process is correct, then it eventually decides.[4]

In the case where two or more processes concurrently invoke the propose() primitive an infinite number of times (i.e. the invocation of propose() at a process happens before the invocation of propose() at another process returns), the result returned by $\triangle$Register is only restricted by validity and agreement.

Note that the validity and agreement properties of $\triangle$Register are similar to traditional consensus [12]. Our termination property is strictly weaker than traditional consensus (i.e., *if a process proposes, it either crashes or decides*).

3.3 Implementing $\triangle$Register

Our implementation of $\triangle$Register uses the $\mathcal{R}$register abstraction. We first give the specification of $\mathcal{R}$register and then describe our implementation of $\triangle$Register.

$\mathcal{R}$register. Roughly speaking, our $\mathcal{R}$register is a round-based register, where processes read and write the content using a single operation. As long as a process does not receive a consistent value from the register, it may keep trying to update the content. If a process returns a value from the register, then no process can ever return a different value from the register.

More formally, the interface of our $\mathcal{R}$register consists of a single primitive, readWrite(). When a process p invokes readWrite() with two arguments, respectively $k \in \mathbb{N}^*$ and $v \in$ Values, such that abort $\notin$ Values, we say that *p writes v*. The readWrite() primitive returns a value in Values $\cup$ {abort}. If p returns from readWrite() with $v' \neq$ abort, we say that *p reads v'*. We now give the specification of our $\mathcal{R}$register:

Decide-validity: If a readWrite(k,v) returns $v' \notin \{v, \text{abort}\}$, then there is at least one readWrite(k',v') invocation such that $k' < k$.
Abort-validity: If a readWrite(k,$*$) invocation returns abort, then there is a distinct readWrite(k',$*$) invocation such that (i) $k' \geq k$, and (ii) readWrite(k',$*$) is invoked before readWrite(k,$*$) returns.
Agreement: If a readWrite(k,$*$) returns $v \neq$ abort, then if any readWrite(k',$*$) with $k' \geq k$ returns, the invocation returns either v or abort.
Termination: If a process invokes readWrite($*$,$*$), then it either crashes or the invocation returns (wait-free).

Note that we do not preclude that two processes use the same round number at the same time.[5] As we will see for $\triangle$Register, agreement will be reached faster though, if processes use different round number.

[4] We insist that the single process be correct to also cover the crash-recovery model [9].
[5] In [9], we consider the crash-recovery model, where a process that crashes and recovers might invoke readWrite() with an old round number. Hence, we do not assume round number uniqueness in $\mathcal{R}$register.

```
object △Register
  method △Register                          {Constructor at process $p_i$}
    register ← new $\mathcal{R}$register                 {Instance of register}
    $k \leftarrow i$                                        {Round number}

  method propose($v$)                       {When $p_i$ proposes a value $v$}
    $k \leftarrow k + n$
    return register.readWrite($k$, $v$)
```

Fig. 1. △Register implementation: code for process p_i

Implementing △Register. Figure 1 gives our implementation of △Register. △Register encapsulates the round number at which the $\mathcal{R}$register is accessed. A process p_i owns rounds $i + n$, $i + 2n$, $\ldots$, and only uses these round numbers within the propose() primitive. Process p_i first invokes the readWrite() primitive using round $i + n$, and increments the round number by n on each subsequent call to propose(). Round number uniqueness is not necessary and is made here only to accelerate the convergence towards agreement.

Implementing $\mathcal{R}$register. Figure 2 gives an implementation of our $\mathcal{R}$register, assuming a majority of correct processes. The key idea behind our implementation is that a process can "lock" a value if it successfully stores it among a majority of processes (we call them *witnesses* hereafter). Once a value is locked, no other value can be agreed upon. The idea follows that of [2]: each witness keeps its own copy of the current value, and the readWrite() invocation accesses the value at witnesses by emulating round-based read-like or write-like operations out of message passing.

The readWrite() primitive essentially consists in reading the value from a majority of witnesses, and writing ("locking") this value among a majority. If the value read by a process corresponds to $\perp$, then the process writes its own value.

A round number is associated with every message and permits a witness that receives a message to abort if the round number carried by the message is below its own round number. The readWrite() primitive aborts as soon as it aborts at a witness. In order to lock a value, the round number maintained at a witness is updated to the highest round number ever seen by the witness.

We give a correctness proof of our implementation of the $\mathcal{R}$register in [9]. Several optimizations are possible. In particular, following an idea in [14], it is possible to eliminate the read phase in steady-state. In [9], we discuss this optimized version of the $\mathcal{R}$register that allows a process to skip the read phase if it is safe to do so. One round-trip communication is enough for a process to propose and decide in this case (steady-state).

```
type Decision is Values ∪ {abort}                              {abort ∉ Values}

object $\mathcal{R}$register
  method $\mathcal{R}$register                                 {Constructor at process $p_i$}
    decision ← abort                                           {decision ∈ Decision}
    $v^* \leftarrow \bot$                                      {value read from witnesses}
    $read_i \leftarrow 0, write_i \leftarrow 0$                {Highest read/write rounds handled by $p_i$}
    $value_i \leftarrow \bot$                                  {$p_i$'s copy of register's value}

  method readWrite(Integer $k$, Values $v$)
    send [READ,$k$] to all processes
    wait until received [ackREAD, $v_j, ts_j, k$] or [nackREAD, $k$] from $\lceil \frac{n+1}{2} \rceil$ processes
    if received at least one [nackREAD, $k$] then
      decision ← abort
    else
      select the [ackREAD, $v_j, ts_j, k$] with the highest $ts_j$
      $v^* \leftarrow v_j$
      if $v^* = \bot$ then
        $v^* \leftarrow v$
      send [WRITE, $v^*, k$] to all processes
      wait until received [ackWRITE, $k$] or [nackWRITE, $k$] from $\lceil \frac{n+1}{2} \rceil$ processes
      if received at least one [nackWRITE, $k$] then
        decision ← abort
      else
        decision ← $v^*$
    return decision

  upon receive [READ, $k$] from $p_j$ do
    if $write_i \geq k$ or $read_i \geq k$ then
      send [nackREAD, $k$] to $p_j$
    else
      $read_i \leftarrow k$
      send [ackREAD, $value_i, write_i, k$] to $p_j$

  upon receive [WRITE, $v_j, k$] from $p_j$ do
    if $write_i > k$ or $read_i > k$ then
      send [nackWRITE, $k$] to $p_j$
    else
      $write_i \leftarrow k$
      $value_i \leftarrow v_j$
      send [ackWRITE, $k$] to $p_j$
```

Fig. 2. $\mathcal{R}$register implementation: code for process p_i

4 Universal Construction

We present here our universal construction, i.e. an algorithm that transforms any local and sequential implementation of an object into a wait-free linearizable shared object. Like any replication algorithm that guarantees some form of strong consistency, a key principle behind our algorithm is to implement a total order among the requests, and to ensure that a request only appears once in the total order. Because objects may be non-deterministic, the replicas have to agree not only on the total order of requests but also on the state update and reply associated with a given request. In our algorithm, they do that at the same time. We describe the algorithm here and give a correctness proof in [9].

```
leader ← new ◇Leader
reply ← nil

procedure submit(Request req)          {To submit a request to the replicated service}
  while true do                          {Loop as long as reply is not received}
    set timer
    send [Request, req] to leader.elect()            {Send request to the leader}
    wait until received [Reply, res] or timer expires
    if received [Reply, res] then
      return res
```

Fig. 3. Client behavior

4.1 Description of the Algorithm

We start by describing the client-side of the algorithm, given in Fig. 3. A client accesses the replicated object using the *submit* procedure. A client sends its request to the current leader and then waits for a reply. If the client does not receive a reply within a certain time, it re-transmits its request (maybe to a different leader). Note that we use a local timer to make this retransmission possible. The choice of time-out period only affects performance; safety is ensured by the server-side abstractions regardless of the chosen time-out period.[6]

On the server-side, each replica p_i executes the algorithm presented in Fig. 4. Each replica has its own copy of the shared object O, and each copy of the object is initialized in the same initial state Λ. A replica also maintains a variable *num* reflecting its opinion about the first free position in the total order. This variable is hence initialized to 1 for each replica.

When a replica receives a request, it verifies that it did not already execute the same request up to its current position *num*. This verification is done locally, without involving any communication step. It is legal because, by the algorithm, the local state of each replica is guaranteed to be coherent with the total order (a proof is given in [9]). If the replica detects that it already decided this request, it simply returns the reply that was committed together with the request, otherwise it introduces the request in the total order.

To insert a new request in the total order, a replica optimistically assumes that its variable *num* reflects the first free position in the total order. It executes the request on its local object, but without performing any update on its internal state. It then constructs an outcome based on the request, the reply and update values returned from the execution of the request. The constructed outcome is proposed at position *num* using a new instance of △Register.

Roughly speaking, an instance of △Register corresponds to a single position in the total order of requests in our universal construction. Distinct instances are indexed with their respective position in the total order.[7] The messages sent

[6] As mentioned earlier, we ignore possible optimizations, such as having an adaptive timer, for the sake of simplicity.

[7] For the sake of clarity, the algorithm in Fig. 4 uses a single instance of △Register and the index appears as a subscript of the propose() primitive. This is equivalent to using several instances of △Register.

```
1:  O ← new Object
2:  leader ← new ◇Leader
3:  register ← new △Register
4:  store ← new OutcomeStore
5:  num ← 1
6:  decision ← abort
7:  prop, res, upd ← nil

8:  upon receive [Request,req] from $c$ do                {Upon receiving a request from a client}
9:    while $true$ do
10:     if store.isCommitted(req,num) = [$true$, decision] then   {Verify uniqueness of request}
11:       send [Reply,decision.res] to $c$
12:       break
13:     [res, upd] ← O.execute(req)                                {Execute request}
14:     prop ← [req, res, upd]
15:     decision ← abort
16:     while decision=abort and leader.elect()=$p_i$ do
17:       decision ← register.propose$_{\text{num}}$(prop)            {Propose request in total order}
18:     if decision=abort then break
19:     store.setCommitted(num,decision)                          {Store request locally}
20:     num ← num + 1
21:     O.update(decision.upd)                                    {Update local state machine}
22:     if decision=prop then
23:       send [Reply,decision.res] to $c$
24:       break
```

Fig. 4. Replica behavior: code for process p_i

```
1:  object OutcomeStore
2:    method OutcomeStore                                  {Constructor at process $p_i$}
3:      $i$ ← 0
4:      outcomes[ ] ← {nil, . . . , nil}                   {Array of requests in total order}

5:    method isCommitted(Request req, Integer index)       {Load request from local state}
6:      for $i$ from 1 to index − 1 do
7:        if outcomes[$i$].req = req then
8:          return [$true$, outcomes[$i$]]
9:      return [$false$, nil]

10:   method setCommitted(Outcome out, Integer index)      {Store request locally}
11:     outcomes[index] ← out
```

Fig. 5. Uniqueness verification: code for process p_i

on behalf of an instance are labeled with its index, to differentiate them from the messages of another instance.

There are three possibilities when a replica executes the **while** loop in lines 16–17: △Register either returns (1) abort or (2) some decision, or (3), the replica stops electing itself.

In case (1), the replica proposes its request again. By the properties of △Register, a replica might need to propose several times before deciding. If the replica is eventually single to propose and does not crash, it will eventually decide. By the properties of ◇Leader, there will eventually be a single process which proposes. In case (2), the replica stores the decided outcome in its local state, increments its variable *num* to the next position, and updates its object using the update value returned with the decision. If the replica committed its outcome, the associated reply is sent to the client, the replica exits and waits for the next request. Otherwise, the replica restarts the main loop of the algorithm with the same request. In case (3), the replica exits the algorithm, because only leaders are allowed to propose.

To accelerate the verification of whether a request is new when it comes in, each replica stores the decided outcomes within its local state, as shown in Fig. 5. It is possible (though very costly) not to verify whether a received request is new, and to systematically propose this request starting from position 1 in the total order.

Following the algorithm, a replica that decides an outcome that it does not propose restarts the algorithm, and again verifies whether the request is a new one. It would be sufficient to verify that the request is new among the outcomes committed since the last verification. We ignore this optimization in Fig. 5 to make the algorithm simpler.

5 Performance Analysis

We analyze the performance of our universal construction in term of message complexity (number of messages it takes to process a request end-to-end) and response time (end-to-end latency observed by clients). To quantify latency, we assume that message transmission times are bounded by some known δ.

5.1 Steady-State

To analyze the steady-state behavior of our algorithm, we consider a nice run in which no process crashes and in which $\Diamond$Leader returns the same leader to all processes at every invocation.[8] Moreover, we assume that this leader uses the optimized implementation of the $\mathcal{R}$register given in [9].

In a nice run, our algorithm exhibits a message complexity of $2n + 2$ and a response time bounded by 4δ, whenever a client submits a new request. Following the algorithm, the leader proposes the request with *num* set to one more than the previous request. Because we consider a nice run, the leader is the perpetual leader, and no other replica has been leader in the meantime. This means that the leader commits the request the first time it proposes it. Committing a request in the optimized version of $\triangle$Register has message complexity $2n$ and latency 2δ (in the optimized implementation, the leader skips the read phase). The communication between client and leader involves a single round-trip message, and has complexity 2 and latency 2δ. In total, we get a message complexity of $2n + 2$, and a total latency of 4δ.

[8] Steady-state performance can be achieved in runs that are not so nice, but where the leader does not crash and does not change (no matter what happens to the other replicas).

```
upon $p_i$ commits decision do
    send [Eager, decision, num] to all processes except $p_i$

upon receive [Eager, decision, num'] where num'=num do
    O.update(decision.upd)
    store.setCommitted(num,decision)
    num $\leftarrow$ num + 1
```

Fig. 6. Shifting work from transition periods to steady-state: code for process p_i

5.2 Transition Periods

The message complexity and response time in a fail-over scenario depends on many factors, such as the number of concurrent leaders that try to take over from the failed leader. If multiple leaders try to take over, it may require multiple rounds inside of $\triangle$Register to commit requests. Furthermore, a leader that takes over has to both read and write inside of $\triangle$Register. In general, this means that each round initiated by a new leader inside of $\triangle$Register requires $4n$ messages, and involves a latency of 4δ. If a new leader has to commit k requests to construct its state during take-over, the leader has to at least initiate k rounds—one for each request. In the next section, we show how to extend our algorithm to reduce the communication during fail-over at the expense of increased communication during steady-state.

6 Genericity

6.1 Trade-Offs

In our algorithm, backups play a very passive role: a backup postpones all work until the last possible moment, and "catches up" when it becomes a leader. This scheme is efficient during steady-state periods, but not during transition periods (i.e., fail-over time). Here, we show how one could easily extend our algorithm (without modifying it) to move work from transition periods to steady-state periods, and thereby achieve a faster fail-over time.

Figure 6 gives an extension in which a leader sends each committed update value to all other replicas. The two **upon** clauses extend the behavior of Fig. 4. The other replicas then process the received update values. Note that the dissemination of update values does not have to be reliable: the basic algorithm ensures consistency and progress, even if a leader fails while sending an update value. We show the different steady-state interaction patterns in Fig. 7. These patterns capture two extremes of a spectrum. We can think of various trade-offs in between these two extremes.

Our scheme also allows for work compression. For example, the leader may gather a number of update messages and send them as a single message. Moreover, some of the "old" updates may become obsolete after "new" updates have been computed. For example if an update message assigns a value to a variable, and if a subsequent update message also assigns a value to the same variable,

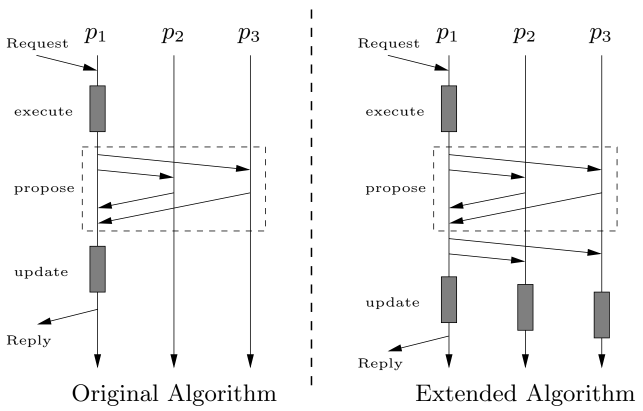


Fig. 7. Steady-state behavior for the original and extended scheme

we only have to apply the most recent update message. Detecting obsolete update values requires knowledge about the semantics of the object, and cannot be performed in a generic manner in the replication algorithm.

6.2 Dealing with Recovery

One of the complications of dealing with process recovery is the use of stable storage. To recover consistently, a process has to store parts of its state in stable storage. When recovering, a process can then access its own stable storage to determine its pre-crash state. Stable storage access is however expensive and should be minimized.

It turns out that we can encapsulate the manipulation of stable storage within our $\mathcal{R}$register. That is, neither our universal construction nor our $\triangle$Register have to manipulate stable storage. A recovering process would start out as a backup, and would construct its state from scratch when it becomes leader, just as if it were a backup that had never failed and never been leader. In [9], we give implementations for $\mathcal{R}$register in different crash-recovery models.

7 Discussion

The key to the indulgence, efficiency, and genericity of our algorithm is the use of two fine-grained underlying abstractions: $\Diamond$Leader and $\triangle$Register. Separately, each of these abstractions is strictly weaker than consensus. Together, they can implement consensus, as we show in Fig. 8.[9]

Consensus defines a single primitive, Propose(). We say that a process *proposes* a value v when it invokes Propose() with v. A process *decides* a value v if it returns from Propose() with v. Only the termination property of consensus (*every correct process eventually decides*) differs from $\triangle$Register.

[9] Throughout this section, we consider the *uniform* variant of the consensus problem (i.e., we do not restrict agreement to correct processes only).

```
object Consensus
  method Consensus                                   {Constructor at process $p_i$}
    register ← new △Register
    leader ← new ◇Leader
    decision ← abort                                 {decision ∈ Decision}

  method Propose(Values $init_i$)
    while decision=abort do
      if leader.elect()=$p_i$ then
        decision ← register.propose($init_i$)
    send [Decision, decision] to all processes
    return decision

  upon receive [Decision, value] from $p_j$ for the first time do
    decision ← value
```

Fig. 8. A simple consensus algorithm: code for process p_i

Our consensus algorithm can be viewed as a modular variant of the Synod consensus scheme underlying the Paxos replication algorithm [14]. In fact, our ◇Leader abstraction is identical to the notion of "sloppy" leader in [15], and △Register precisely crystallizes Lampson's intuition [15] about the safety of the agreement part of the Synod consensus scheme [14]: we capture both safety and liveness aspects of that intuition through our △Register specification.[10] To our knowledge, Paxos pioneered the idea of combining a notion of leader election and some form of register, although not explicitly identified (more recent variants of the algorithm, e.g. [7,10], more explicitly use register-like abstractions).

There are two major differences between the use of ◇Leader and △Register in our universal construction and in Paxos. First, Paxos is not a universal construction since it relies on objects to be deterministic. To implement the agreement on total order and on state in a single instance of eventual consensus, our algorithm invokes △Register *a posteriori*, after processing a request. This is in contrast to Paxos because, being deterministic, the replicas only need to agree on the total order of requests, and this agreement can be reached before a request is processed. Second, our two fine-grained abstractions are first class citizens in our universal construction. As we observe in [4], two variants of Paxos are actually given in [14]: a modular algorithm based on a consensus primitive, built using a sloppy leader and a register-like abstraction, and then an ad-hoc algorithm that opens these abstractions for the sake of performance. Promoting our ◇Leader and △Register as first class citizens of the algorithm is exactly what makes our universal construction more efficient than [8]. Indeed, where a consensus object requires all replicas to propose and decide, △Register allows a single replica to propose and decide by itself. With consensus, it is not possible for backups to play a passive role where they just witness the actions taken by a primary. Moreover, consensus encapsulates ◇Leader whereas △Register does not. If we used consensus and had another notion of leader in the replication algorithm (e.g., the primary), these leaders may not coincide (the same process may not be leader at both levels). Having different leaders at different levels would likely result in

[10] △Register is also close to the *k-converge* primitive, (with $k = 1$) of [16].

an additional leader-to-leader communication that is absent in our single-leader scheme. This saves messages between client and replicas.

Indirectly, we argue that $\diamond$Leader and $\triangle$Register are natural abstractions to build universal constructions in a message-passing model, pretty much like consensus and atomic register do in a shared-memory model.

References

1. P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second IEEE International Conference on Software Engineering (ICSE)*, 1976.
2. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.
3. J. F. Bartlett. A nonstop kernel. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP)*, 1981.
4. R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. Technical Report EPFL-2001, Swiss Federal Institute of Technology, January 2001.
5. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems.* Addison-Wesley, 1993.
6. T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector to solve consensus. *Journal of the ACM*, 43(4):685–722, 1996.
7. G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC) (to appear)*, July 2002.
8. X. Défago and A. Schiper. Semi-passive replication and lazy consensus. Technical Report DSC/2000/012, Swiss Federal Institute of Technology, February 2000.
9. P. Dutta, F. Frølund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. Technical Report EPFL/IC/2002/28, Swiss Federal Institute of Technoology, Lausanne, May 2002.
10. E. Gafni and L. Lamport. Disk paxos. In *International Symposium on Distributed Computing*, pages 330–344, 2000.
11. R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.
12. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
13. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
14. L. Lamport. The part-time parliament. Technical Report 49, DEC Systems Research Center, 1989. Also published in ACM Transactions on Computer Systems (TOCS), Vol. 16, No. 2, 1998.
15. B. Lampson. How to build a highly available system using consensus. In *Proceedings of the International Workshop on Distributed Algorithms (WDAG), Springer-Verlag, LNCS*, September 1996.
16. J. Yang, G. Neiger, and E. Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 297–306, 1998.

Ruminations on Domain-Based Reliable Broadcast

Svend Frølund[1] and Fernando Pedone[2]

[1] Hewlett-Packard Laboratories (HP Labs), Palo Alto, CA 94304, USA
svend_frolund@hp.com
[2] Hewlett-Packard Laboratories (HP Labs), Palo Alto, CA 94304, USA
Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
fernando_pedone@hp.com

Abstract. A distributed system is no longer confined to a single administrative domain. Peer-to-peer applications and business-to-business e-commerce systems, for example, typically span multiple local-area and wide-area networks, raising issues of trust, security, and anonymity. This paper introduces a distributed systems model with an explicit notion of *domain* that defines the scope of trust and local communication within a system. We introduce leader-election oracles that distinguish between common and distinct domains, encapsulating failure-detection information and leading to modular solutions and proofs. We show how Reliable Broadcast can be implemented in our domain-based model, we analyze the cost of communicating across groups, and we establish lower-bounds on the number of cross-domain messages necessary to implement Reliable Broadcast.

1 Introduction

1.1 Motivation

Distributed systems are no longer confined to a single administrative domain. For example, peer-to-peer applications and business-to-business e-commerce systems typically span multiple local-area and wide-area networks. In addition, these systems commonly span multiple organizations, which means that issues of trust, security, and anonymity have to be addressed. Such global environments present new challenges to the way we program and organize distributed systems.

At the programming level, several researchers have recently proposed constructs to decompose a global system into smaller units that define boundaries of trust and capture locality. An example of such a construct is the notion of ambient introduced by Cardelli [5]. The trend reflects a growing realization that one should not treat a global system as a very large local-area network.

1.2 The Domain-Based Model

We introduce a distributed systems model with an explicit notion of *domain* that defines the scope of trust and local communication within a system. A domain is a set of processes, and a system is a set of domains. A domain-based model allows us to employ novel complexity measures, such as the number of times a given algorithm communicates across domain boundaries. Such complexity measures reflect the real-world costs of crossing firewalls and communicating along wide-area links. In contrast, a conventional "flat" model, with a single set of processes, attributes the same cost to any point-to-point communication. Besides complexity measures, a domain-based model also allows us to capture the realistic notion that failure information within a local-area network is more accurate than failure information obtained across wide-area networks. We develop a notion of leader-election oracle that provides one level of information to processes within the same domain, and another, with weaker properties, to processes in other domains. Finally, domains enable us to reflect the common security policy that a process trusts other processes in the same organization (domain), but not necessarily processes in other domains. We introduce Byzantine failures into our model, and having domains allows us to attribute Byzantine behavior to entire domains (as seen from the outside).

1.3 Domain-Based Algorithms

In building global systems, a fundamental concern is the reliable dissemination of information. We want the information dissemination to be reliable, but also efficient and scalable. In practice, an important aspect of efficiency is the amount of wide-area bandwidth consumed by information dissemination algorithms. In terms of scalability, an important quality is to make the system decentralized. To address the issue of information dissemination in large-scale systems, we describe a number of Reliable Broadcast algorithms that work in our domain-based model.

The first set of algorithms tolerate crash failures only. In this context, we start by examining how to implement Reliable Broadcast in a purely asynchronous, domain-based model. We then incrementally add synchrony assumptions to this model, and show how one can exploit these assumptions, in the form of leader-election oracles, to reduce the number of messages that cross domain boundaries. We analyze the cost of these algorithms in terms of cross-domain messages and present lower bounds that quantify the inherent cost of reliability in a domain-based model.

The second set of algorithms implement Reliable Broadcast in a system with Byzantine failures. We consider a domain to be Byzantine if it contains a Byzantine process, and we develop algorithms that can tolerate Byzantine domains. Considering Byzantine behavior at the domain level reflects the notion that a domain is the unit of trust and security: once a domain has been compromised, it is likely that an adversary can take over as many processes as it wishes within that domain. We first provide a protocol that ensures agreement: all correct processes in all non-Byzantine domains deliver the same set of messages. We then give an algorithm that also ensures *consistency:* messages are tagged with unique identifiers, and no two correct processes in non-Byzantine domains deliver different messages with the same identifier. Consistency is an important property. It prevents spurious messages from being delivered, which may happen, for example, if an erroneous sender keeps using the same message identifier with different message contents.

1.4 Related Work

Compared with the ambient calculus [4], and other wide-area models based on process algebras, our system model explicitly captures failures and the availability of failure information.

A number of papers have addressed the issue of information dissemination with Reliable Broadcast. The algorithms in [12] use message diffusion, and tolerate crash and link failures. In [7], the authors present time and message efficient Reliable Broadcast algorithms that tolerate crash and omission failures. The protocol in [10] exploits failure detection to more efficiently diffuse messages. All these protocols assume a flat universe of processes. If we were to employ them on top of a networking infrastructure with wide-area networks, the resulting wide-area message complexity would be proportional to the total number of processes. With our protocols, the wide-area message complexity is proportional to the number of domains (assuming that there are no wide-area links within a domain).

The Reliable Broadcast algorithm in [16] assumes a flat model with Byzantine processes. In our terminology, the algorithm implements agreement but not consistency—there is no notion of message identifiers to ensure that processes deliver the same message content for the same identifier. The notion of Byzantine Agreement [14,15] captures a variation of Reliable Broadcast in the Byzantine model. With Byzantine Agreement, a single process broadcasts a value, and all correct processes must decide on the same value. The original formulation of the problem in [15] requires an explicit notion of time. The definition of Asynchronous Byzantine Agreement [3] does not use time and only requires honest processes to decide if the broadcasting process is honest. Asynchronous Byzantine Agreement ensures consistency relative to a single message, but the definition of the problem is for a single message. Thus, besides implementing Byzantine-tolerant Reliable Broadcast in a domain-based model, our algorithms also bridge the gap between Reliable Broadcast and Byzantine Agreement.

Furthermore, numerous works on group communication have considered a hierarchical approach to large-scale distributed systems (e.g., [1,2,11,13]). These works differ from ours with respect to either the underlying system model, the failure model, or the problems solved.

1.5 Summary of Contributions

The paper makes the following contributions:

– We define a domain-based system model, which corresponds to current wide-area distributed systems, and give a specification of Reliable Broadcast in this model.

– We introduce leader-election oracles that distinguish between common and distinct domains, encapsulate failure-detection information, and lead to modular solutions and proofs.
– We show how Reliable Broadcast can be implemented in our domain-based model. We start with the simple case of only two groups and then build up to more complex cases.
– We analyze the cost of communicating across groups, evaluate the performance of our protocols in terms of cross-domain messages, and provide lower-bounds on the number of cross-domain messages necessary to implement Reliable Broadcast.

2 System Model and Definitions

2.1 Processes, Failures, and Communication

We assume that the system is composed of groups of processes, that is, $\Pi = \{\Pi_1, \Pi_2, ..., \Pi_n\}$, where $\Pi_x = \{p_1, p_2, ..., p_{n_x}\}$. When we need to distinguish processes from different groups, we will use superscripts: $p_i^x \in \Pi_x$. Processes may be *honest* (i.e., they execute according to their protocols) or *malicious* (i.e., Byzantine). Honest processes can crash, but before they crash, they follow their protocols.

A process that is honest and does not crash is *correct*; if the process is honest but crashes it is *faulty.* If a group has at least one malicious process, then it is *bad*; otherwise the group is *good.* Therefore, good groups can contain only correct and faulty processes, while bad groups can contain any kind of processes. In a good group Π_x, we assume that at most $f_x < n_x$ processes crash.

Processes communicate by message passing. Each message m has three fields: *sender*(m), the process where m originated, *id*(m), a unique identifier associated with m, and *val*(m), the actual contents of m. We assume that the network is fully connected, and each link is reliable. A reliable link guarantees that (a) if p_i sends a message m to p_j, and both p_i are p_j are correct, then p_j eventually receives m; (b) each message is received at most once by honest processes; and (c) if an honest process receives a message m, and if sender(m) is honest, then m was sent.

The system is asynchronous: message-delivery times are un-bounded, as is the time it takes for a process to execute steps of its local algorithm. We assume the existence of a discrete global clock, although processes do not have access to it—the global clock is used only to simplify some definitions. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

2.2 Domain-Based Reliable Broadcast

The domain-based Reliable Broadcast abstraction is defined by the primitives byz-broadcast(m) and byz-deliver(m), and has the following properties:

– *(Validity.)* If a correct process in a good group byz-broadcasts m, then it byz-delivers m.
– *(Agreement.)* If a correct process in a good group byz-delivers m, then each correct process in every good group also byz-delivers m.
– *(Integrity.)* Each honest process byz-delivers every message at most once. Moreover, if sender(m) is honest, then sender(m) byz-broadcast m.
– *(Consistency.)* Let p_i and p_j be two processes in good groups. If p_i byz-delivers m, p_j byz-delivers m', and $id(m) = id(m')$, then $val(m) = val(m')$.

Domain-based Reliable Broadcast without consistency is a generalization of Reliable Broadcast in a flat model. Throughout the paper, we use Reliable Broadcast locally in groups as a building block to implement domain-based Reliable Broadcast. We refer to such an abstraction as local Reliable Broadcast. Local Reliable Broadcast is defined by the primitives r-broadcast(m) and r-deliver(m). The properties of local Reliable Broadcast can be obtained from the properties of domain-based Reliable Broadcast by replacing byz-broadcast(m) and byz-deliver(m) by r-broadcast(m) and r-deliver(m), respectively, and considering a system composed of one good group only. Without the consistency property, local Reliable Broadcast can be implemented with a conventional "flat" algorithm [8]. When the consistency property is needed, such as in our most general algorithm in Section 5.2, the implementation is different from Reliable Broadcast implementations in the flat model. We discuss such an implementation further in Section 5.2.

2.3 Leader-Election Oracles

In some of our algorithms we use oracles that give hints about process crashes—they do not provide any information about which processes are malicious. Our oracles are quite similar to the Ω failure detector in [6]. Where Ω is defined for a "flat" system, our oracles are defined for a distributed system with groups.

We introduce a notion of group oracle—an oracle that gives information about the processes in a particular group. For example, the group oracle Ω_x gives information about the processes in Π_x. Thus, our system contains a set of oracles, $\{\Omega_1, \Omega_2, \ldots, \Omega_n\}$, one per group. Each process has access to all oracles. Having an oracle per group, rather than a single "global" oracle, allows us to distinguish between local information and remote information. A process p_i^x gets local information from the oracle Ω_x (information about other processes in Π_x). In contrast, a process p_i^x obtains remote information from an oracle Ω_y, where $y \neq x$ (information about processes in other groups). We use the notion of group oracle to model a system where local information is stronger than remote information.

We use the set G to denote the set of all processes in the system ($G = \Pi_1 \cup \Pi_2 \cup \ldots \cup \Pi_n$). Moreover, we use the set good to denote the set of good groups ($\mathsf{good} \subseteq \Pi$).

In the following, we adapt the model in [6] to define a notion of group oracle. A failure pattern represents a run of the system. A failure pattern F captures which processes in G crash, and when they crash. Formally speaking, a failure pattern is a map from time to a subset of G. Based on a failure pattern F, we can define the set of processes that crash in F as $\mathsf{crash}(F)$:

$$F \in \mathcal{F} = \mathcal{T} \to 2^G \tag{1}$$

$$\mathsf{crash}(F) = \cup_{t \in \mathcal{T}} F(t) \tag{2}$$

$$\mathsf{correct}_x(F) = \Pi_x \setminus \mathsf{crash}(F), \quad \text{if } \Pi_x \in \mathsf{good} \tag{3}$$

where $\mathcal{F}$ is the set of all failure patterns, and F is an element of this set. For a good group Π_x, the set $\mathsf{correct}_x(F)$ is the set of correct processes in Π_x.

A group-oracle history H_x is a map from process-time pairs to a process in Π_x.[1] A pair (q, t) maps to the process p_i^x if the process q at time t believes that p_i^x has not crashed. We also say that q *trusts* p_i^x at time t. Intuitively, a failure pattern is what actually happens in a run, and a group-oracle history represents the output from a group oracle. We can now define a group oracle Ω_x as a map from a failure pattern to a set of group-oracle histories:

$$H_x \in \mathcal{H}_x = (G \times \mathcal{T}) \to \Pi_x \tag{4}$$

$$\Omega_x \in \mathcal{D}_x = \mathcal{F} \to 2^{\mathcal{H}_x} \tag{5}$$

The set $\mathcal{H}_x$ is the set of all group-oracle histories relative to a group Π_x, and the history H_x is an element in this set. Furthermore, the set $\mathcal{D}_x$ is the set of all oracles for Π_x, and Ω_x is an element in this set (in other words, Ω_x is an oracle).

We can use the above definitions to establish constraints on the information that an oracle Ω_x gives a process p. First of all, if Π_x is a bad group, there are no constraints—Ω_x may return arbitrary information. If Π_x is a good group, there are two sets of constraints: a set of constraints for local information ($p \in \Pi_x$) and remote information ($p \notin \Pi_x$):

- *Local Trust:* For any good group Π_x, eventually all correct processes in Π_x trust the same correct process in Π_x. Formally:

$$\Pi_x \in \mathsf{good} \Rightarrow \big\langle \forall F, \forall H_x \in \Omega_x(F), \exists t \in \mathcal{T}, \\ \exists q \in \mathsf{correct}_x(F), \forall p \in \mathsf{correct}_x(F), \forall t' \geq t : H_x(p, t') = q \big\rangle \tag{6}$$

- *Remote Trust:* For any good group Π_x, eventually, all correct processes in all good groups trust a correct process in Π_x. Formally:

$$\Pi_x \in \mathsf{good} \Rightarrow \big\langle \forall F, \forall H_x \in \Omega_x(F), \exists t \in \mathcal{T}, \\ \forall \Pi_y \in \mathsf{good}, \forall p \in \mathsf{correct}_y(F), \forall t' \geq t : H_x(p, t') \in \mathsf{correct}_x(F) \big\rangle \tag{7}$$

[1] The concept of a group-oracle history is similar to the notion of failure-detector history in [6]. Where a failure-detector history is global, a group-oracle history is local to a particular group. Furthermore, where a failure-detector history maps to a set of processes (the processes that have failed at a given time), a group-oracle history maps to a single process (a process that is believed not to have crashed).

– *Stability:* For any good group Π_x, eventually, any correct process in a good group trusts the same process in Π_x forever. Formally:

$$\Pi_x \in \mathsf{good} \Rightarrow \big\langle \forall F, \forall H_x \in \Omega_x(F), \exists t \in \mathcal{T}, \\ \forall\, \Pi_y \in \mathsf{good}, \forall p \in \mathsf{correct}_y(F), \exists q \in \mathsf{correct}_x(F), \forall t' \geq t : H_x(p,t') = q \big\rangle \quad (8)$$

The *Local Trust* property defines the information that processes in Π_x can obtain about processes in Π_x (i.e., local information). As seen by processes in Π_x, a group oracle Ω_x is similar to the Ω oracle in [6]. We use Ω_x for leader election within Π_x. We elect a leader to decrease the number of inter-group messages that originate from Π_x: only a leader is allowed to send inter-group messages.

Processes in groups other than Π_x use the *Remote Trust* and *Stability* properties of Ω_x to select processes in Π_x that can serve as destinations for inter-group messages. With *Remote Trust* and *Stability* , different processes in remote groups may forever select different destinations in Π_x. An alternative way to define group oracles would be to have a single level of trust, and elect a global leader for each group to handle all inter-group communication. However, insisting that only a single process in Π_x serves as destination does not decrease the number of messages sent to Π_x, it only increases the load on this single process.

3 Abstractions for Solving Reliable Broadcast

It is possible to implement domain-based Reliable Broadcast in a purely asynchronous system. However, one can come up with more efficient algorithms in a model with oracles. We want to provide insights about both types of algorithms: algorithms that assume a purely asynchronous model and algorithms that use the oracles introduced in Section 2.3. It turns out that both types of algorithms share a common principle: for each broadcast message, at least one correct process in the sending group communicates the message to a correct process in the receiving group. The choice of underlying model does not change this principle, only how it is achieved. Rather than describe a number of algorithms that are identical except for their dealing with the underlying model, we encapsulate model-related concerns in well-defined abstractions. The use of these abstractions allows us to simplify the description of our algorithms and to modularize the proof of their correctness.

Our abstractions are specified relative to a given group. Rather than explicitly pass a group as parameter to every invocation, we supply the group as a subscript of the abstraction. For example, the abstraction $\mathsf{senders}_x()$ means "the senders abstraction relative to a group Π_x."

The $\mathsf{senders}_x()$ *abstraction.* This abstraction is used within a group Π_x to select the processes in Π_x that send messages to processes in other groups. The $\mathsf{senders}_x()$ abstraction returns a set of processes in Π_x. If p_i^x in Π_x invokes $\mathsf{senders}_x()$ and the returned set includes p_j^x, we say that p_i^x *selects* p_j^x. The $\mathsf{senders}_x()$ abstraction has the following properties:

- *Termination:* The $\mathsf{senders}_x()$ abstraction is non-blocking.
- *Validity:* Eventually, $\mathsf{senders}_x()$ selects a correct process in Π_x.
- *Agreement:* Eventually, any invocation of $\mathsf{senders}_x()$ selects the same processes.

Notice that the $\mathsf{senders}_x()$ abstraction is only available to processes in the group Π_x. Notice also that the set of processes returned by $\mathsf{senders}_x()$ may change over time.

The $\mathsf{destinations}_x()$ *abstraction.* Processes in group Π_y can use the $\mathsf{destinations}_x()$ abstraction to select the recipients in Π_x of inter-group messages. That is, processes in Π_y use $\mathsf{destinations}_x()$ to determine which processes in Π_x to send messages to. If a process in Π_y, p_i^y, invokes $\mathsf{destinations}_x()$, and if the returned set includes a process p_j^x, we say that p_i^y *selects* p_j^x. The $\mathsf{destinations}_x()$ abstraction has the following properties:

- *Termination:* The $\mathsf{destinations}_x()$ abstraction is non-blocking.
- *Validity:* Eventually, $\mathsf{destinations}_x()$ selects a correct process in Π_x.
- *Agreement:* For any process p, eventually any invocation of $\mathsf{destinations}_x()$ by p selects the same processes.

Notice that $\mathsf{destinations}_x()$ is a global abstraction—any process in any group can invoke this abstraction under the above guarantees. Although the properties of $\mathsf{destinations}_x()$ and $\mathsf{senders}_x()$ are quite similar, the Agreement properties are different. For $\mathsf{senders}_x()$, the Agreement property ensures that all processes in Π_x eventually select the same set of senders. For $\mathsf{destinations}_x()$, the Agreement property only ensures that any individual process "stabilizes" on the same set of destinations, different processes may stabilize on different sets of destinations. The weaker Agreement property for $\mathsf{destinations}_x()$ implies that processes within Π_x can share the load: the abstractions do not insist that only a single process receives all messages in Π_x.

Implementing the $\mathsf{senders}_x()$ *and* $\mathsf{destinations}_x()$ *abstractions.* In an asynchronous model, we can implement the abstractions by returning a subset of Π_x that contains at least $f_x + 1$ processes (such a set will contain at least one correct process). In a model with oracles, we can implement the abstractions by simply returning the single process output from the oracle. We show these implementations in Table 1.

Table 1. Implementation of the abstractions with or without oracles

	asynchronous implementation	oracle-based implementation
$\mathsf{senders}_x()$, $\mathsf{destinations}_x()$	**return** $\{p_j^x \mid j \leq f_x + 1\}$	**return** Ω_x

In the asynchronous implementation, we return the same set of processes in both abstractions. However, there is no requirement to do so: the implementation of $\mathsf{senders}_x()$ could return one subset of size $f_x + 1$ and the implementation of $\mathsf{destinations}_x()$ could return another. Moreover, although the oracle-based implementation of $\mathsf{senders}_x()$ is identical to the implementation of $\mathsf{destinations}_x()$, the abstractions still provide different agreement properties: $\mathsf{senders}_x()$ is only called by processes in Π_x, and the oracle gives stronger guarantees to processes within Π_x.

The $\mathsf{send}_x()$ *and* $\mathsf{receive}_x()$ *abstractions.* The $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ abstractions capture reliable communication between groups. The $\mathsf{send}_x()$ abstraction takes a message as argument, and the $\mathsf{receive}_x()$ abstraction returns a message. The two abstractions have the following properties:

– *Termination:* The $\mathsf{send}_x()$ abstraction is non-blocking.
– *Validity:* If a correct process in Π_y invokes $\mathsf{send}_x()$ with a message m then eventually a correct process in Π_x that invokes $\mathsf{receive}_x()$ receives m.
– *Integrity:* If $\mathsf{receive}_x()$ returns a message m to an honest process p, and if sender(m) is honest, then sender(m) called $\mathsf{send}_x()$ with m.

Unlike a traditional message-sending operation, $\mathsf{send}_x()$ takes a group, not a process, as the message destination—we use a subscript to designate the group. The $\mathsf{send}_x()$ abstraction encapsulates the concern of sending to a correct process. This is in contrast to $\mathsf{destinations}_x()$, which simply ensures that some correct process is selected.

Algorithm 1 $\mathsf{send}_x()$ and $\mathsf{receive}_x()$ based on destination selection

```
 1: procedure send_x(m)
 2:    dest ← destinations_x()
 3:    send [GS,m] to all processes in dest
 4:    fork task watch(m,dest,x)

 5: task watch(m,dest,x)
 6:    while TRUE
 7:       if dest ≠ destinations_x() then
 8:          dest ← destinations_x()
 9:          send [GS,m] to all processes in dest

10: when receive([GS,m])
11:    receive_x(m)
```

We can implement $\mathsf{send}_x()$ with regular point-to-point communication primitives and the $\mathsf{destinations}_x()$ abstraction previously introduced (see Algorithm 1). From the agreement property of the $\mathsf{destinations}_x()$ abstraction, processes executing Algorithm 1 eventually stop sending messages, but since they cannot know when this will happen, the main loop in the watch() task never terminates.

4 A Special Case: Two Good Groups

4.1 The Algorithm

We examine how to implement domain-based Reliable Broadcast. For simplicity, we restrict our scope to systems with only two good groups—our algorithm tolerates crash failures only. We introduce algorithms that work for any number of groups and tolerate malicious processes in Section 5. Throughout this section, we consider two good groups, Π_x and Π_y, and assume that messages originate in Π_x.

We are interested in algorithms that are judicious about the number of inter-group messages used to deliver a broadcast message in both groups. We present a generic algorithm that relies on the abstractions in Section 3. By instantiating the abstractions in different ways (with or without using oracles), we achieve solutions for different models.

Algorithm 2 has four *when* clauses, but only one executes at a time. Whenever one of the *when* conditions evaluates true, the clause is executed until the end. If more than one condition evaluates true at the same time, one clause is arbitrarily chosen.

Algorithm 2 Reliable broadcast for two good groups

```
 1: Initially:
 2:     $rcvMsgs \leftarrow \emptyset$
 3:     $fwdMsgs \leftarrow \emptyset$

 4: procedure byz-broadcast($m$)
 5:     r-broadcast($\{m\}$)

 6: when r-deliver($mset$)
 7:     for all $m \in mset \setminus rcvMsgs$ do byz-deliver($m$)
 8:     $rcvMsgs \leftarrow rcvMsgs \cup mset$

 9: when $p_i \in \mathsf{senders}_x()$ and $rcvMsgs \setminus fwdMsgs \neq \emptyset$
10:     for all $m \in rcvMsgs \setminus fwdMsgs$ do $\mathsf{send}_y([\mathrm{FW}, m])$
11:     $fwdMsgs \leftarrow rcvMsgs$

12: when $\mathsf{receive}_y([\mathrm{FW}, m])$
13:     send [LC,$m$] to all processes in $\Pi_y$

14: when receive [LC,$m$] for the first time
15:     byz-deliver($m$)
```

4.2 Algorithm Assessment

Because Algorithm 2 only relies on the specification of our abstractions, and not on their implementation, it is possible to mix and match abstractions with different implementations. For example, it is possible to combine an oracle-based implementation of $\mathsf{senders}()$ with an asynchronous implementation of $\mathsf{destinations}()$. Such a combination would exploit synchrony assumptions within groups but not across groups.

The various combinations of abstraction implementations give rise to different costs in inter-group messages for the resulting Reliable Broadcast algorithm. If we use the asynchronous implementation of both abstractions in Algorithm 2, we achieve a performance of $(f_x+1)(f_y+1)$ inter-group messages per broadcast.

If we combine the asynchronous implementation of $\mathsf{destinations}_y()$ with an oracle-based implementation of $\mathsf{senders}_x()$, we obtain a domain-based Reliable Broadcast algorithm with a best-case message cost of f_y+1. The algorithm may have a higher message cost for arbitrary periods of time, but the properties of Ω_x ensure that eventually only a single process in Π_x will be selected. Thus, eventually only a single process will send inter-group messages. Moreover, with the asynchronous implementation of $\mathsf{destinations}_y()$, the number of destinations is constant (i.e., $f_y + 1$), and so is the number of messages sent by each selected sender.

If instead we combine the asynchronous implementation of $\mathsf{senders}_x()$ with an oracle-based implementation of $\mathsf{destinations}_y()$, we obtain a domain-based Reliable Broadcast algorithm with best-case message cost of $f_x + 1$. With an asynchronous implementation of $\mathsf{senders}_x()$, there will always be $f_x + 1$ senders. Due to the stability property of the oracle Ω_y, each of the $f_x + 1$ senders will eventually trust a correct process in Π_y forever. Thus, at any of the $f_x + 1$ senders there is a time t after which $\mathsf{destinations}_y()$ returns the same process forever. This means that after t, the watch task in Algorithm 1 does not send any messages. Thus, after t, each broadcast message results in each of the f_x+1 senders sending exactly one inter-group message. Notice that the senders do not necessarily send to the same process in Π_y.

Finally, if we combine the oracle-based implementation of $\mathsf{senders}_x()$ with the oracle-based implementation of $\mathsf{destinations}_y()$, the resulting Reliable Broadcast algorithm will have best case of 1 inter-group message per broadcast. As we discussed above, there is a time after which the oracle Ω_x results in the selection of the single sender in Π_x. Furthermore, there is a time after which the Ω_y oracle returns the same destination forever to each sender. In combination, the two oracles ensure that, eventually, each broadcast message results in only a single process in Π_x sending a single message to a single process in Π_y.

4.3 Some Lower Bounds

It is easy to see that when both $\mathsf{senders}_x()$ and $\mathsf{destinations}_y()$ use oracle-based implementations, our resulting algorithm has an optimal best-case cost in terms of inter-group messages: no algorithm can solve Reliable Broadcast without exchanging at least one message between groups.

We informally argue now that if either $\mathsf{senders}_x()$ or $\mathsf{destinations}_y()$ has an oracle-based implementation, our resulting algorithms also have optimal best-case costs. The argument is similar for both situations, and so, assume that

$\mathsf{senders}_x()$ has an implementation that uses oracles. What we want to show is that the inter-group message cost of our algorithm, namely $f_y + 1$, is a lower bound for algorithms where the sending group does not have information about failures in the receiving group. In the best case, a correct process p_i in Π_x is selected to send messages to processes in Π_y. Assume for a contradiction that p_i sends only f_y messages. Consider a run where each process in Π_y that receives a message fails right after receiving the message; the remaining processes in Π_y will then never receive the message, and therefore, cannot deliver it.

In a purely asynchronous system (i.e., neither the implementation of $\mathsf{senders}_x()$ nor the implementation of $\mathsf{destinations}_y()$ use oracles), our algorithm has inter-group message cost of $(f_x+1)(f_y+1)$, which is not optimal. Consider, for example, the special case when both n_x and n_y are greater than $f_x + f_y$. In this case, the following algorithm solves domain-based Reliable Broadcast: the first f_x+f_y+1 processes in Π_x send one message to the first f_x+f_y+1 processes in Π_y. The resulting message cost is $f_x + f_y + 1$. Moreover, from Proposition 1, it turns out that this algorithm is optimal.

Proposition 1. *Let $\mathcal{A}$ be a reliable broadcast algorithm in which processes do not query any oracle. For every run of $\mathcal{A}$ in which a correct process in Π_x byz-broadcasts some message, at least $f_x + f_y + 1$ messages are sent from Π_x to Π_y.*

PROOF: The proof is by contradiction. Assume that processes in Π_x send only $f_x + f_y$ messages to Π_y. Let R be a failure-free run in which p_i in Π_x byz-broadcasts message m and S the set of processes that send messages to Π_y in R.

We claim that there exists a run R' in which p_i also byz-broadcasts m, each process in $S_x \subseteq S$ crashes, and no process sends more messages in R' than it sends in R. Let p_j be a process in S_x that crashes at time t_j and p_k some process that sends one more message in R' at time $t > t_j$. Then, we can construct a run R'_j such that from time t_j until time t, p_j is very slow and does not execute any steps, and so, does not send any message—this can be done since processes can be arbitrarily delayed. Process p_k cannot distinguish between R_j and R'_j, and will also send a message to Π_y in R'_j. After t, p_j is back to normal and sends a message to Π_y. Thus, $f_x + f_y + 1$ messages are sent to Π_y, a contradiction.

We now show that $|S| > f_x$. Assume $|S| \leq f_x$. Then, we can construct a run in which $S_x = S$, every p_j in S_x crashes and from our claim above, no other process in $\Pi_x \setminus S$ sends a message to processes in Π_y. Thus, correct processes in Π_x deliver m, and processes in Π_y never receive m, and so, cannot deliver it.

Without loss of generality assume that each process in S_x sends only one message to Π_y. Thus, processes in $S \setminus S_x$ can send up to $f_x + f_y - |S_x| = f_y$ messages to processes in Π_y. Let set S_y denote such processes in Π_y. Since any f_y processes may crash in Π_y, assume that processes in S_y crash in R'. Therefore, in R' correct processes in Π_x deliver m, but correct processes in Π_y do not, a contradiction. □

Even when n_x or n_y are smaller than $f_x + f_y + 1$, our algorithm, which exchanges $(f_x + 1)(f_y + 1)$ messages is not optimal. Consider the following case in which $n_x = n_y = 4$ and $f_x = f_y = 2$. With our algorithm, processes in Π_x will send 9 messages to processes in Π_y. While this is certainly enough to guarantee correctness, it is possible to do better: instead of sending to three distinct processes in Π_y, each process p_i^x sends a message to processes p_i^y and $p_{(i \,\mathbf{mod}\, 4)+1}^y$. With such an algorithm, only 8 messages are exchanged!

5 The General Case

5.1 A Reliable Broadcast Algorithm without Consistency

Algorithm 3 implements Reliable Broadcast (without the consistency property) for any number of groups and tolerates any number of failures, that is, there is no limit on the number of bad groups and on the number of correct processes in good groups.

Algorithm 3 Reliable broadcast algorithm without the consistency property

```
1:  Initially:
2:      $rcvMsgs \leftarrow \emptyset$
3:      $fwdMsgs \leftarrow \emptyset$

4:  procedure byz-broadcast($m$)
5:      r-broadcast($\{m\}$)

6:  when r-deliver($mset$)
7:      for all $m \in mset \setminus rcvMsgs$ do byz-deliver($m$)
8:      $rcvMsgs \leftarrow rcvMsgs \cup mset$

9:  when $p_i \in$ senders$_x$() and $rcvMsgs \setminus fwdMsgs \neq \emptyset$
10:     for all $\Pi_y \in \Pi$ do send$_y$([FW, $rcvMsgs \setminus fwdMsgs$])
11:     $fwdMsgs \leftarrow rcvMsgs$

12: when receive$_y$([FW, $mset$])
13:     r-broadcast($mset$)
```

Algorithm 3 builds on Algorithm 2. It works as follows. In order for some process in a good group to byz-deliver a message, it has to r-deliver it (i.e., using local Reliable Broadcast). This guarantees that all correct processes in the group will r-deliver the message. Using a mechanism similar to the one used in Algorithm 2, the message will eventually reach some correct process in each good group, which will r-broadcast the message locally, and also propagate it to other groups. In principle, the inter-group communication of Algorithm 3 is similar to the flooding-based Reliable Broadcast algorithm presented in [8] for a "flat" process model.

5.2 A Reliable Broadcast Algorithm with Consistency

We now extend Algorithm 3 to also enforce the consistency property. Algorithm 4 resembles Algorithm 3. The main differences are the first *when* clause, the fact that all messages are signed to guarantee authenticity, and the fact that the local Reliable Broadcast requires the consistency property.

Algorithm 4 Reliable broadcast algorithm with the consistency property

```
1: **Initially:**
2:    $rcvMsgs \leftarrow \emptyset$
3:    $fwdMsgs \leftarrow \emptyset$
4:    $dlvMsgs \leftarrow \emptyset$

5: **procedure** byz-broadcast($m$)
6:    r-broadcast($\{m\} : k_i$)

7: **when** r-deliver($mset : k_j$)
8:    $rcvMsgs \leftarrow rcvMsgs \cup mset$
9:    **for each** $m \in rcvMsgs \setminus dlvMsgs$ **do**
10:       **if** [for $\lceil (2n+1)/3 \rceil$ groups $\Pi_y, \exists p_l \in \Pi_y$ : r-delivered ($mset' : k_l$) **and** $m \in$
          $mset'$] **then**
11:          byz-deliver($m$)
12:          $dlvMsgs \leftarrow dlvMsgs \cup \{m\}$

13: **when** $p_i \in \mathsf{senders}_x()$ **and** $rcvMsgs \setminus fwdMsgs \neq \emptyset$
14:    **for each** $\Pi_y \in \Pi$ **do** $\mathsf{send}_y$([FW, $(rcvMsgs \setminus fwdMsgs) : k_i$])
15:    $fwdMsgs \leftarrow rcvMsgs$

16: **when** $\mathsf{receive}_y$([FW, $mset : k_j$])
17:    r-broadcast($mset : k_j$)
```

Local Reliable Broadcast with the consistency property can be implemented with an algorithm similar, in principle, to Algorithm 4. To r-broadcast some message m, p_i in Π_x signs m and sends it to all processes in Π_x (we use $mset : k_i$ to denote that the message set $mset$ is signed by p_i). When a process p_j receives m for the first time, it also signs m and sends it to all processes in Π_x. If a process receives m from $\lceil (2n+1)/3 \rceil$ processes, it r-delivers m. (A detailed description of this algorithm as well as all proofs of correctness can be found in [9].)

References

1. D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, 1998.

2. Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of IEEE Conference on Distributed Systems and Networks (DSN)*, June 2000.
3. G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), October 1985.
4. L. Cardelli. Abstractions for mobile computation. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*. Springer Verlag, 1999.
5. L. Cardelli. Wide area computation. In *Proceedings of the 26th International Colloqium on Automata, Languages, and Programming (ICALP)*, 1999. LNCS 1644.
6. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
7. T. D. Chandra and S. Toueg. Time and message efficient reliable broadcasts. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, September 1990.
8. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
9. S. Frølund and F. Pedone. Ruminations on domain-based reliable broadcast. Technical Report IC/2002/52, Ecole Polytechnique Fédérale de Lausanne (EPFL), July 2002.
10. R. Guerraoui and A. Schiper. Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, Sendai, Japan, June 1996.
11. K. Guo, W. Vogels, and R. van Renesse. Structured virtual synchrony: Exploring the bounds of virtual synchronous group communication. In *Proceedings of the ACM SIGOPS European Workshop*, 1996.
12. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.
13. I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, April 2000.
14. L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
15. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), April 1980.
16. M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicasts in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994.

Stateless Termination Detection

Gideon Stupp

Sheer Networks, Israel 67021. stupp@sheernetworks.com

Abstract. The switches and routers of large scale networks cannot manage a state per user session. The burden of memory management would overwhelm the network. Therefore, it is important to find distributed network algorithms which hold a state only at the initiating node. Termination detection algorithms are particularly interesting, since they can be used in the implementation of other stateless algorithms.
The importance of stateless termination detection is apparent in mulitcast trees. Multicast trees are commonly used to multicast messages across the network. In many cases the mulitcast message represents a request sent from the root node that must be answered by all the leaves of the tree. In most networks the leaves could send their answer directly to the root. Unfortunately, the root would have no way of knowing when all the leaves answered the request. Broadcast-echo algorithms are often used in this case, but these algorithms require a state in the internal nodes of the mulitcast tree. Nack oriented protocols are also common, particularly in reliable multicast implementations. These algorithms are optimized for continues downstream information from the source to the destinations rather than for transactional request-reply operations.
We present a simple algorithm for termination detection in trees and DAGs which does not require managing a state in the nodes of the graph. The algorithm works even if the graph changes during the execution. For a tree with n nodes, the number of bits added to each message is $O(\log n)$. We also discuss how this algorithm may be used in general graphs.

1 Introduction

Network cores are usually comprised of many switching nodes that are connected to each other by point to point communication links. Users may be connected to some of the switches, and the switching fabric enables communication between any two users by forwarding messages between the switches.

When a user needs to send the same information to many other users, multicast trees are commonly used. A tree is defined over the network where the originating sender is the root and the destinations are the leaves. When the root sends packets they follow the edges of the tree, and duplicate only at the branching points. This way the same packet is never sent twice over the same link. This scheme is used in IP multicast and in ATM point to multipoint connections.

Using mulitcast trees is much more efficient than sending a different message to each of the destinations. However, there is no simple way for the root to know how many nodes received its multicast. This is important if the root must wait for an answer from all the destinations (a distributed query). One solution is to use an echo-broadcast algorithm, where the leaves of the tree send their answer to their parents and each internal node in the tree waits for all the answers from all its children before propagating the result to its parent. While such solutions may be efficient in theory, they are impractical in data communication networks for several reasons:

1. Network cores can not hold a state per user session. The memory management would overwhelm the network switches.
2. The aggregation of the information may be specific to the executing algorithm. It is unlikely that in the near future network cores will support user logic (but see [1]).
3. Backtracking the multicast tree may not be the most efficient way to return data to the root. The structure of the multicast tree may be affected by parameters such as quality of service, or which nodes support multicast. It may be more efficient for the leaves to send a point to point reply directly to the root.

Reliable multicast protocols usually implement some sort of termination detection to verify that all the packets reached their destinations and theoretically could be used for distributed queries. However, these protocols were created to support a large and continuing flow of information from the source to the destinations, where upstream acknowledgments are a major overhead. The information flow in distributed queries is very different since the result of the query must be returned to the source. As acks can be piggybacked on actual information that is flowing upstream, ack implosion is not a problem.

For this paper we model modern data communication networks using a variant of the standard asynchronous message passing model. First, we assume that all nodes can communicate with all other nodes and that the cost of transferring the same message between any two nodes is the same. This property captures the concept of an underlying unsaturated communication infrastructure, where the cost of sending a message through several hops is almost the same as through one hop. Next, we assume that only the initiating node of any algorithm execution can manage a state dedicated for that execution. This property protects the network infrastructure from the affects of any specific execution.

For simplicity we assume that messages are not lost and that nodes do not fail. Since only the root manages a state, a simple timeout mechanism can be used in practice.

Algorithms that do not hold a state in the nodes sometimes move the state to the message. For example, a trivial solution for detecting termination in a multicast tree would be to have each message keep a history of the nodes it visited and the number of children each node had. The leaves return the messages to the root, which can then reconstruct the entire tree. Such solutions are inefficient in the size of the messages and in the amount of memory used in the root. In this paper we suggest an algorithm for termination detection in trees and DAGs which uses $O(\log n)$ bits in each message (n = number of nodes in the tree) and $O(n)$ bits in the root. The algorithm is based on a new counter that is carried by the messages, much like a hop count. The new counter, which we call the *bifurcation count (b-count)*, gives an estimate on the amount of duplications created by a single branch of the multicast tree.

Definition 1. *Let $T = (V, E)$ be a tree. For any node v, denote by v_c the number of children of v and by $A(v)$ the set of ancestors of v (i.e., the set of nodes on the path from the root to v). The* bifurcation count (b-count) *of node v, v_b, is defined to be the logarithm of the multiplication of the number of children of all the ancestors of v with respect to base two. Formally:*

$$v_b = \begin{cases} 0 & A(v) = \emptyset \\ \log(\prod_{u \in A(v)} u_c) & A(v) \neq \emptyset \end{cases} \tag{1}$$

One of the simplest uses of b-count is to limit the amount of messages created by mulitcast using a TTL. However, it can also be used more sophisticatedly. In our algorithms the b-count of messages that reach the leaves is sent back to the root and the root uses this information to estimate the expected number of acknowledge messages.

This work was done as part of the development of a large scale distributed object oriented network management system (Sheer MetroCentral$^{\mathrm{TM}}$). In this system, hundreds of thousands of autonomous objects communicate by sending asynchronous messages to each other. Many queries in the system are implemented by accessing one object, which delegates the request to one or more other objects, and so on. Thus, multiple concurrent sessions of multicast requests naturally occur. The algorithm presented in this paper was designed to replace a standard broadcast-echo implementation. In OO programming it is common to delegate from one object to just one other object without branching. Thus, in many cases there are a lot of internal nodes in the multicast tree and only a few leaves, which makes backtracking extremely inefficient.

Related Work: Computation trees have long been used to detect termination, either for diffusing algorithms [2] or for decentralized computations [3]. These algorithms are usually used for detecting termination in multicast trees, but require bidirectional links and more importantly, a state in the internal nodes of the multicast tree. Wave based solutions [4], both synchronous and asynchronous [5, 6] are sometimes used instead of computational trees, to optimize the average message complexity, yet they too manage a state in each node.

A considerable amount of research was done in reliable multicast algorithms, which use termination detection to ensure the delivery of packets ([7,8]). These algorithms are typically optimized for transferring streams of information from the root of the tree to the destinations, rather than for supporting request-reply sessions.

In Sect. 2 we present an algorithm for termination detection in binary multicast trees. In Sect. 3 we make a small adjustment to make it practical for general trees and directed acyclic graphs (DAGs).

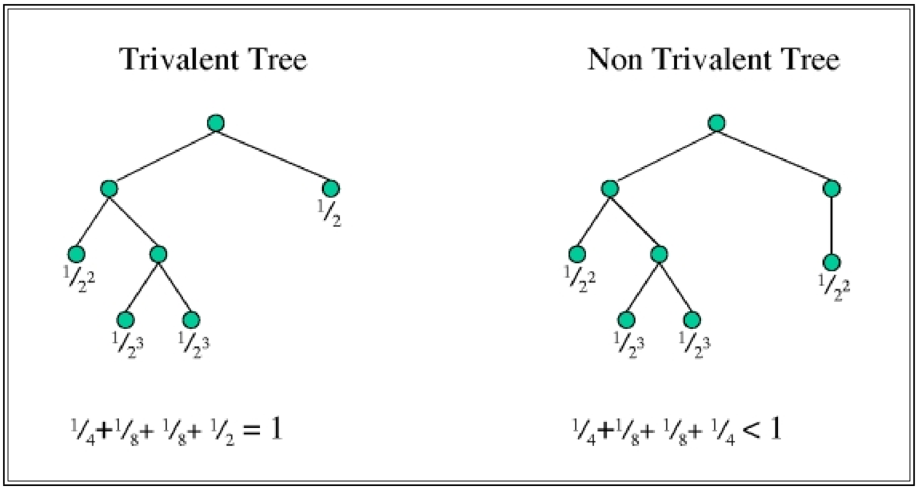


Fig. 1. Sum of weights of the leaves of a trivalent tree.

2 Binary Trees

Algorithm 1 presents the termination detection code in the root and the internal nodes of a binary multicast tree. The Algorithm is based on a well known property of trivalent trees which is described in Theorem 1. The theorem states that if branches are not missing from a binary tree (i.e., it is a trivalent tree) then a weight can be distributed between the leaves of the tree, according to their depth, such that the total weight is always 1 (See Figure 1).

Definition 2. *A* trivalent *tree (also known as a 3-cayley tree or a boron tree [9]) is a binary tree where the number of children of every node is either 2 or 0.*

For trivalent trees, the bifurcation count of node v is simply the depth of v.

Let $T = (V, E)$ be a weighted binary tree. Denote by $w(v)$ the weight of node v and by $S(T) = \sum_{v \in V} w(v)$ the sum of the weights of all the nodes. Denote by $d(v)$ the depth of node v.

Theorem 1. *Let T be a weighted trivalent tree where the weight of any interior node is* 0 *and the weight of any leaf node v is $\frac{1}{2^{d(v)}}$. Then $S(T) = 1$.*

Proof. We prove the theorem by induction on the height, $H(T)$, of tree T with root r.

$H(T) = 1$: Since r has no children it is a leaf with weight $\frac{1}{2^0} = 1$. Thus $S(T) = w(r) = 1$.

$H(T) = k + 1$: Denote by T_1 the subtree rooted at the left child of r, by T_2 the subtree rooted at the right child of r and notice that both T_1 and T_2 must exist since T is trivalent. Adding 1 to the depth of all the leaves in T_1 (T_2) halves $S(T_1)$ ($S(T_2)$). It follows that $S(T) = \frac{S(T_1)}{2} + \frac{S(T_2)}{2}$. Since

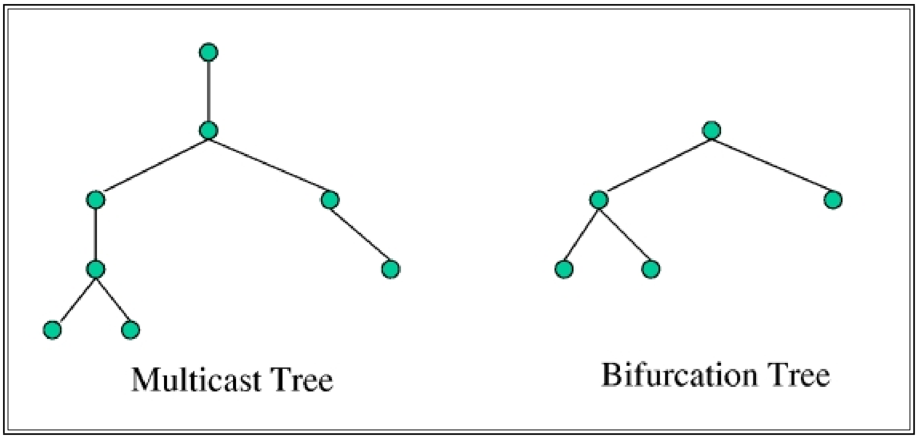


Fig. 2. A binary tree and its trivalent bifurcation tree.

T is trivalent both T_1 and T_2 are trivalent so by the induction hypothesis $S(T) = \frac{1}{2} + \frac{1}{2} = 1$ and the theorem holds.

□

Corollary 1 states that partial sums of $S(T)$ are always smaller than 1. This follows from the fact that all the weights of the leaves of T are positive.

Definition 3. *Tree T' is a* partially spanning subtree *of tree T if T' is a subtree of T, the root of T' is the root of T and the leaves of T' are also leaves in T.*

Corollary 1. *If T is trivalent and T' is a partially spanning tree of T then $S(T') \leq 1$ and $S(T') = 1$ only if $T' = T$.*

Using Corollary 1 it is possible to implement termination detection at the root of a multicast tree. Every message would count the number of times it was duplicated on its way to the leaf and this value would be returned to the root as the depth of the leaf. The root would add up the weights of the leaves and the algorithm would terminate when the total sum reaches 1. While multicast trees are not necessarily trivalent, the tree created by ignoring non-duplicating nodes is trivalent, and this tree has the same number of leaves as the original multicast tree. We call such trees *bifurcation trees* (See Fig. 2).

Definition 4. *The* bifurcation tree *of a multicast tree T is the tree induced by the bifurcation of messages as they traverse T.*

Corollary 2. *Bifurcation trees are trivalent trees.*

The code for the root node and the other nodes of a multicast tree is shown in Alg. 1. Every node in the tree keeps a list of children, C, where $||C|| \leq 2$ and

Algorithm 1 Stateless termination detection in binary trees.

```
   [root node $r$]
   $C$          : const      list of children;                          // $||C||$=0..2.
   $b_{max}$    : integer ;                          // Maximal bifurcation encountered.
   $n$          : integer    init 0;                                   // Numerator.

   Initialization:
1:     if $||C||$=0 then exit;                                // Tree is a single node.
2:     $b_{max}$=log $||C||$;
3:     for every $x$ in $C$ do send $\langle$msg,$r$,$b_{max}\rangle$ to $x$;

   Upon arrival of a $\langle$ack,$b\rangle$ message from node $u$:
4:     if $b <= b_{max}$ then $n := n + 2^{(b_{max}-b)}$;
       else
5:         $n := 2^{(b-b_{max})} * n$+1;
6:         $b_{max} := b$;
7:     if $n = 2^{b_{max}}$ then exit;                          // Algorithm terminated!

   [other nodes]
   $C$          : const      list of children;                          // $||C||$=0..2.

   Upon arrival of a $\langle$msg,$r$,$b\rangle$ message:
8:     if $||C||$=0 then send $\langle$ack,$b\rangle$ to root $r$.                        // Leaf.
9:     for every $x$ in $C$ do send $\langle$msg,$r$,$b + \log||C||\rangle$ to $x$;  // Bifurcate if necessary.
```

$||C|| = 0$ if the node is a leaf. Since handling fractions can be inefficient and inaccurate, the root node keeps track of the sum of weights, $S(T)$, by holding the numerator, n, and the denominator, $2^{b_{max}}$, separately (the denominator is named b_{max} because the least common multiple for the received denominators is always equal to $2^{(\text{maximal received depth})}$). When the algorithm starts (Lines 1-3) the root checks that it is not the only node in the tree, initializes b_{max} to 0 or 1 according to the log of the number of its children and finally sends the multicast message to all its children. In general, the root waits until $S(T)$ equals 1 (Line 7). As messages traverse the tree they are processed by the internal nodes (Lines 8-9). Whenever a message is duplicated by a node, the bifurcation counter, b, in both duplicates is increased by one (Line 9). When the message reaches a leaf, it is redirected to the root (Line 8) and if the message's b-count, b, is equal to the current b_{max} then the root simply increases n by 1 (Line 4). In general, however, there are two possible cases (Lines 4-7):

1. If an acknowledge message arrives from leaf u with b-count $b \leq b_{max}$, then the weight of leaf u, $w(u) = \frac{1}{2^b}$, must be adjusted to the current least common multiple of the denominators, $2^{b_{max}}$, before adding it to the total sum. Thus, the nominator of $w(u)$ (which is equal to 1) is multiplied by $2^{(b_{max}-b)}$ before being added to n (Line 4).

2. If an acknowledge message arrives from leaf u with b-count $b > b_{max}$ then the least common multiple of the denominators is 2^b and the total sum must be adjusted to it before $w(u)$ added. The nominator of the sum, n, is multiplied by $2^{(b-b_{max})}$ and only then the nominator of $w(u)$ (which is equal to 1) is added to it (Line 5). Finally, b_{max} is updated to represent the new least common multiple (Line 6).

To prove the correctness of Alg. 1, we show that during any point in the execution of the algorithm, $\frac{n}{2^{b_{max}}}$ is equal to $S(T')$, where T' is the partially spanning subtree of the bifurcation tree that was fully traversed by messages and that the acks of its leaves returned to the root (Theorem 2).

Let E be an execution of Alg. 1 and let $\alpha = m_1, m_2, \ldots$ be the sequence of messages that return to the root. Let β be some prefix of α. Denote by T_β the partially spanning subtree of the bifurcation tree created by taking all the leaves that sent messages in β along with all their ancestors up to the root.

By the model, messages are not lost and nodes do not fail. Since every internal node forwards the multicast message to each of its sons (Lines 3, 9) every leaf in the tree must eventually return an ack.

Corollary 3. *Tree T_α is equal to the entire bifurcation tree.*

Theorem 2. *For every execution E and for every prefix β in E, $\frac{n}{2^{b_{max}}} = S(T_\beta)$.*

Proof. Assume that the theorem is correct for some prefix $\beta_k = m_1, m_2, \ldots, m_k$. We show that the theorem holds for $\beta_{k+1} = \beta_k, m_{k+1}$.

Assume that message m_{k+1} came from leaf u with bifurcation counter, b (i.e., $w(u) = \frac{1}{2^b}$). By the induction hypothesis, $\frac{n}{2^{b_{max}}} = S(T_{\beta_k})$. Since $T_{\beta_{k+1}} \supset T_{\beta_k}$ and since only leaf nodes have weights it follows that $S(T_{\beta_{k+1}}) = S(T_{\beta_k}) + w(u) = \frac{n}{2^{b_{max}}} + \frac{1}{2^b}$.

If $b \leq b_{max}$ then the least common multiple of the two denominators is $2^{b_{max}}$. Thus b_{max} does not change, n is set to $n + 2^{(b_{max}-b)}$ (Line 4) and the theorem holds. If $b > b_{max}$ then the least common multiple of the two denominators is 2^b. Thus n is set to $n * 2^{(b-b_{max})} + 1$ (Line 5), b_{max} is set to b (Line 6) and the theorem holds. □

3 General Trees and DAGs

It is possible to use Alg. 1 as is to detect termination in multicast trees with nodes of any degree. However, if the number of children of some of the nodes is not a power of 2 then roundoff errors may accumulate when the bifurcation count is calculated.

We suggest constructing the bifurcation tree of a general multicast tree as a binary tree. Each node u in the multicast tree is represented in the bifurcation tree as a binary trivalent subtree, where the leaves of the subtree are the children of u (see Fig. 3). It can be shown that the number of leaves in the total bifurcation tree is equal to the number of leaves in the multicast tree.

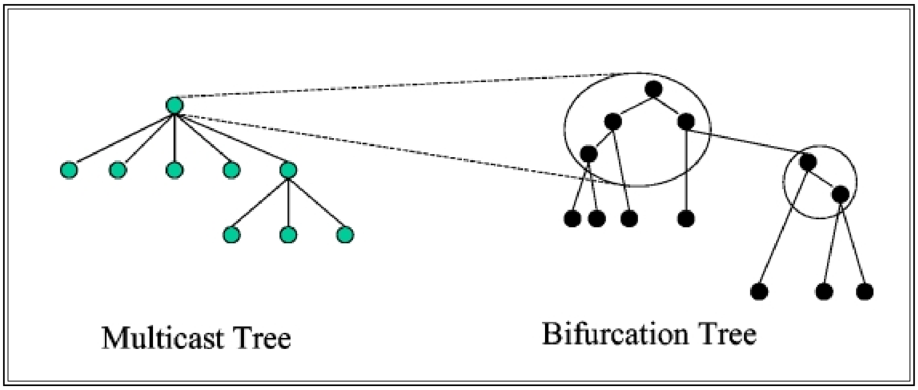


Fig. 3. A general tree and its trivalent bifurcation tree.

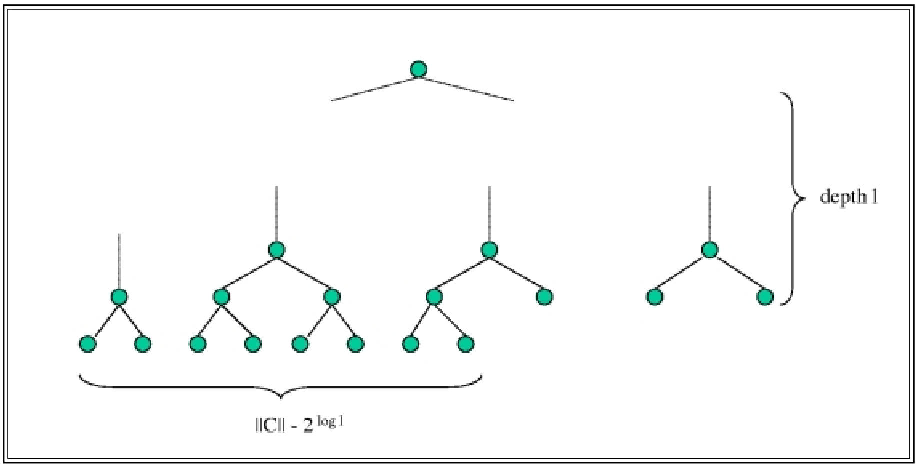


Fig. 4. A trivalent tree with $||C||$ leaves ($l = \lfloor \log ||C|| \rfloor$).

There are many ways to exchange an internal node, u, having $||C|| > 0$ children with a trivalent tree of $||C||$ leaves. One such way is to construct an almost-balanced tree, where the depth of the leaves is either $\lfloor \log ||C|| \rfloor$ or $\lfloor \log ||C|| \rfloor + 1$. The number of leaves at each depth can be calculated, and is presented in Fig. 4. Since the abstraction of the bifurcation tree is created in the internal nodes, changes in the code are not necessary in the root (Alg. 2).

Although the algorithm was presented for trees, it works as is for DAGs. The bifurcation tree would include all the possible paths in the DAG, from the root to the leaves.

Algorithm 2 Updating the bifurcation count in general trees.

```
   [other nodes]
   $C$          : const      init array of children of size $||C||$;
   $l$          : integer    init 0;

   Upon arrival of a $\langle$msg,$r$,$b\rangle$ message:
1:     if $||C|| = 0$ then send $\langle$msg,$b\rangle$ to root $r$;          // Leaf.
2:     $l := \lfloor \log ||C|| \rfloor$;
3:     for $i$:= 1 to $||C||$ do
4:         if $(i \leq 2^{l+1} - ||C||)$ then send $\langle$msg,$r$,$b + l\rangle$ to $C[i]$;
5:         else send $\langle$msg,$r$,$b + l + 1\rangle$ to $C[i]$;
       od;
```

4 Future Work

Graphs that contain loops present a difficult problem for stateless algorithms. Since there is no state in the nodes, there is no way for a message to detect it returned to the same node twice. It is possible to solve this problem if we assume that there is a spanning tree embedded in the graph. Since the same spanning tree can be concurrently used by many sessions of the stateless algorithms, it satisfies our model.

Assuming that termination detection is available on general graphs, it is possible search for practical stateless algorithms for various problems, such as finding the shortest path between any two nodes or finding the minimal spanning tree. But since the result of these algorithms may be the entire graph, the best solution might be to recreate the graph at the root. A more challenging problem may be to find the length of the shortest path between any two nodes for example.

Acknowledgments. I want to thank Yehuda Afek and Dan Touitou for helpful discussions and support. In particular, I want to thank Manor Mendel for bringing to my attention Theorem 1 and how it simplifies the explanations and proofs.

References

1. David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.
2. E. Dijkstra and B. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
3. Nir Shavit and Nissim Francez. A new approach to detection of locally indicative stability. In *Automata, Languages and Programming*, pages 344–358, 1986.

4. Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge Uni Press, Cambridge, 1994.
5. E. Dijkstra, W. Feijen, and A Gasteren. Derivation of a termination detection algorithm for distributed computations. In *Inf. Proc. Lett. 16, 5*, pages 217–219, 1983.
6. E. Dijkstra. Shmuel safra's version of termination detection. Technical Report EWD998, The University of Texas at Austin, 1987.
7. Donald F. Towsley, James F. Kurose, and Sridhar Pingali. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. *IEEE Journal of Selected Areas in Communications*, 15(3):398–406, 1997.
8. Brian Neil Levine and J. J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems*, 6(5):334–348, 1998.
9. Eric W. Weisstein. Binary tree. http://mathworld.wolfram.com/.

RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks[*]

Nancy Lynch[1] and Alex A. Shvartsman[2,1]

[1] Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, NE43-365, Cambridge, MA 02139, USA.
[2] Department of Computer Science and Engineering, University of Connecticut, 191 Auditorium Road, Unit 3155, Storrs, CT 06269, USA.

Abstract. This paper presents an algorithm that emulates atomic read/write shared objects in a dynamic network setting. To ensure availability and fault-tolerance, the objects are replicated. To ensure atomicity, reads and writes are performed using *quorum configurations*, each of which consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. The algorithm is *reconfigurable*: the quorum configurations may change during computation, and such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time. The algorithm tolerates processor stopping failure and message loss. The algorithm performs three major tasks, all concurrently: reading and writing objects, introducing new configurations, and "garbage-collecting" obsolete configurations. The algorithm guarantees atomicity for arbitrary patterns of asynchrony and failure. The algorithm satisfies a variety of conditional performance properties, based on timing and failure assumptions. In the "normal case", the latency of read and write operations is at most $8d$, where d is the maximum message delay.

1 Introduction

This paper presents an algorithm that can be used to implement atomic read/write shared memory in a dynamic network setting, in which participants may join or fail during the course of computation. Examples of such settings are mobile networks and peer-to-peer networks. One use of this service might be to provide long-lived data in a dynamic and volatile setting such as a military operation.

In order to achieve availability in the presence of failures, the objects are replicated. In order to maintain memory consistency in the presence of small and transient changes, the algorithm uses *configurations*, each of which consists of a set of *members* plus sets of *read-quorums* and *write-quorums*. In order to accommodate larger and more permanent changes, the algorithm supports *reconfiguration*, by which the set of members and the sets of quorums are modified. Such changes do not cause violations of atomicity. Any quorum configuration may be installed at any time.

[*] This work was supported by the NSF ITR Grant 0121277. The work of the first author was also supported by AFOSR under contract F49620-00-1-0097 and by NTT under contract MIT9904-12. The work of the second author was also supported by NSF CAREER Award 9984778 and NSF Grant 9988304.

We first provide a formal specification for reconfigurable atomic shared memory as a global service. We call this service RAMBO, which stands for "Reconfigurable Atomic Memory for Basic Objects" ("Basic" means "Read/Write"). The rest of the paper presents our algorithm and its analysis. The algorithm carries out three major activities, all concurrently: reading and writing objects, introducing new configurations, and removing ("garbage-collecting") obsolete configurations. The algorithm is composed of a *main algorithm*, which handles reading, writing, and garbage-collection, and a global reconfiguration service, *Recon*, which provides the main algorithm with a consistent sequence of configurations. Reconfiguration is loosely coupled to the main algorithm, in particular, several configurations may be known to the algorithm at one time, and read and write operations can use them all.

The main algorithm performs read and write operations using a two-phase strategy. The first phase gathers information from read-quorums of active configurations and the second phase propagates information to write-quorums of active configurations. This communication is carried out using background gossiping, which allows the algorithm to maintain only a small amount of protocol state information. Each phase is terminated by a *fixed point* condition that involves a quorum from each active configuration. Different read and write operations may execute concurrently: the restricted semantics of reads and writes permit the effects of this concurrency to be sorted out afterwards.

The main algorithm also includes a facility for *garbage-collecting* old configurations when their use is no longer necessary for maintaining consistency. Garbage-collection also uses a two-phase strategy, where the first phase communicates with an old configuration and the second phase communicates with a new configuration. A garbage-collection operation ensures that both a read-quorum and a write-quorum of the old configuration learn about the new configuration, and that the latest value from the old configuration is conveyed to a write-quorum of the new configuration.

The reconfiguration service is implemented by a distributed algorithm that uses distributed consensus to agree on the successive configurations. Any member of the latest configuration c may propose a new configuration at any time; different proposals are reconciled by an execution of consensus among the members of c. Consensus is, in turn, implemented using a version of the Paxos algorithm [17], as described formally in [8]. Although such consensus executions may be slow—in fact, in some situations, they may not even terminate—they do not delay read and write operations.

We show atomicity for arbitrary patterns of asynchrony and failure. We analyze performance *conditionally*, based on timing and failure assumptions. For example, assuming that gossip and garbage-collection occur periodically, that reconfiguration is requested infrequently enough for garbage-collection to keep up, and that quorums of active configurations do not fail, we show that read and write operations complete within time $8d$, where d is the maximum message latency.

Comparison with other approaches. Consensus algorithms can be used directly to implement an atomic data service by allowing participants to agree on a global total ordering of all operations [17]. In contrast, we use consensus to agree only on the sequence of configurations and not on the individual operations. Also, in our algorithm, the termination of consensus affects the termination of reconfiguration, but not of read and write operations.

Group communication services (GCSs) [1] can also be used to implement an atomic data service, e.g., by implementing a global totally ordered broadcast service on top of a view-synchronous GCS [11] using techniques of [16,2]. In most GCS implementations, forming a new view takes a substantial amount of time, and client-level operations are delayed during the view-formation period. In our algorithm, reads and writes can make progress during reconfiguration. Also, in some standard GCS implementations, e.g., [5], performance is degraded even if only one stopping failure occurs; our algorithm uses quorums to tolerate small numbers of failures.

A dynamic primary configuration GCS was introduced in [7] and used to implement atomic memory, using techniques of [3] within each configuration. That work restricts the set of possible new configurations to those satisfying certain intersection properties with respect to the previous configurations, whereas we impose no such restrictions. Like other solutions based on GCSs, the algorithm of [7] delays reads and writes during reconfiguration.

Single reconfigurer implementations for atomic memory are considered in [19,10]. In these approaches, the failure of the reconfigurer disables future reconfiguration. Also, in [19,10], garbage-collection of an old configuration is tightly coupled to the introduction of a new configuration, whereas in our new algorithm, garbage-collection is carried out in the background, concurrently with other processing.

Other related work. The first general scheme for emulating shared memory in message-passing systems by using replication and accessing majorities of time-stamped replicas was given in [21]. An algorithm for majority-based emulation of atomic read/write memory was presented in [3]. This algorithm introduced a two-phase paradigm in which the first phase gathers information from a majority of participants, and the second phase propagates information to a majority. A *quorum system* [13]—a generalization of majority sets—is a collection of sets such that any two sets, called *quorums*, intersect [12]. Quorum systems have been used to implement data replication protocols, e.g., [4,6,14]. Consensus algorithms have been used as building blocks in other work, e.g, [15].

Note. This paper is an extended abstract of a full report [20]. The full version includes specifications of all components, complete proofs, and additional results.

2 Data Types

We assume distinguished elements $\perp$ and $\pm$, which are not in any of the basic types. For any type A, we define types $A_\perp = A \cup \{\perp\}$. and $A_\pm = A \cup \{\perp, \pm\}$. If A is a poset, we augment its ordering by assuming that $\perp < a < \pm$ for every $a \in A$.

We assume the following data types and distinguished elements: I, the totally-ordered set of *locations*. T, the set of *tags*, defined as $\mathbb{N} \times I$. M, the set of *messages*. X, the set of *object identifiers*, partitioned into subsets X_i, $i \in I$; X_i is the set of identifiers for objects that may be created at location i. For any $x \in X$, $(i_0)_x$ denotes the unique i such that $x \in X_i$. For each $x \in X$, we define V_x, the set of values that object x may take on, and $(v_0)_x \in V_x$, the initial value of x.

We also assume: C, the set of *configuration ids*; we use the trivial partial order on C, in which all elements are incomparable. For each $x \in X$, $(c_0)_x \in C$, the *initial configuration id* for x. For each $c \in C$: $members(c)$, a finite subset of I,

$read\text{-}quorums(c)$ and $write\text{-}quorums(c)$, two sets of finite subsets of $members(c)$. We assume: (1) $members((c_0)_x) = \{(i_0)_x\}$, that is, the initial configuration for x has one member, the creator of x. (2) For every c, every $R \in read\text{-}quorums(c)$, and every $W \in write\text{-}quorums(c)$, $R \cap W \neq \emptyset$.

We define functions and sets for configurations: $update$, a binary function on $C_\pm$, defined by $update(c, c') = \max(c, c')$ if c and c' are comparable, $update(c, c') = c$ otherwise. $extend$, a binary function on $C_\pm$, defined by $extend(c, c') = c'$ if $c = \bot$ and $c' \in C$, and $extend(c, c') = c$ otherwise. $CMap$, the set of *configuration maps*, defined as $\mathbb{N} \to C_\pm$. We extend the $update$ and $extend$ operators elementwise to binary operations on $CMap$. $truncate$, a unary function on $CMap$, defined by $truncate(cm)(k) = \bot$ if there exists $\ell \leq k$ such that $cm(\ell) = \bot$, $truncate(cm)(k) = cm(k)$ otherwise. Truncation removes all the configuration ids that follow a $\bot$. $Truncated$, the subset of $CMap$ such that $cm \in Truncated$ if and only if $truncate(cm) = cm$. $Usable$, the subset of $CMap$ such that $cm \in Usable$ iff the pattern occurring in cm consists of a prefix of finitely many $\pm$s, followed by an element of C, followed by an infinite sequence of elements of $C_\bot$ in which all but finitely many elements are $\bot$.

3 Reconfigurable Atomic Memory Service Specification

Our specification for the RAMBO service consists of an external signature plus a set of traces that embody RAMBO's safety properties. No liveness properties are included; we replace these with conditional latency bounds, which appear in Section 8. The external signature appears in Figure 1. We use I/O automata notation for all our specifications.

Input:
- $\mathsf{join}(\mathsf{rambo}, J)_{x,i}$, J a finite subset of $I - \{i\}$, $x \in X$, $i \in I$, such that if $i = (i_0)_x$ then $J = \emptyset$
- $\mathsf{read}_{x,i}$, $x \in X, i \in I$
- $\mathsf{write}(v)_{x,i}$, $v \in V_x, x \in X, i \in I$
- $\mathsf{recon}(c, c')_{x,i}$, $c, c' \in C, i \in members(c), x \in X, i \in I$
- fail_i, $i \in I$

Output:
- $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$, $x \in X, i \in I$
- $\mathsf{read\text{-}ack}(v)_{x,i}$, $v \in V_x, x \in X, i \in I$
- $\mathsf{write\text{-}ack}_{x,i}$, $x \in X, i \in I$
- $\mathsf{recon\text{-}ack}(b)_{x,i}$, $b \in \{\mathsf{ok}, \mathsf{nok}\}, x \in X, i \in I$
- $\mathsf{report}(c)_{x,i}$, $c \in C, c \in X, i \in I$

Fig. 1. RAMBO(x): External signature

The client at location i requests to join the system for a particular object x by performing a $\mathsf{join}(\mathsf{rambo}, J)_{x,i}$ input action. The set J represents the client's guess at a set of processes that have already joined the system for x. If $i = (i_0)_x$, the set J is empty, because $(i_0)_x$ is supposed to be the first process to join the system for x. If the join attempt is successful, the RAMBO service responds with a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$ output action. The client at i initiates a read (resp., write) operation using a read_i (resp., write_i) input action, which the RAMBO service acknowledges with a $\mathsf{read\text{-}ack}_i$ (resp., $\mathsf{write\text{-}ack}_i$) output. The client initiates a reconfiguration using a recon_i input, which is acknowledged with a $\mathsf{recon\text{-}ack}_i$ output. RAMBO reports a new configuration to the client using a report_i output. Finally, a stopping failure at location i is modelled using a fail_i input action. We model process "leaves" as failures.

The set of traces describing RAMBO's safety properties consists of those that satisfy an implication of the form "environment assumptions imply service guarantees". The environment assumptions are simple "well-formedness" conditions:

Well-formedness: (1) For every x and i: (a) No $\mathsf{join}(\mathsf{rambo}, *)_{x,i}$, $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*, *)_{x,i}$ event is preceded by a fail_i event. (b) At most one $\mathsf{join}(\mathsf{rambo}, *)_{x,i}$ event occurs. (c) Any $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*, *)_{x,i}$ event is preceded by a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$ event. (d) Any $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, or $\mathsf{recon}(*, *)_{x,i}$ event is preceded by an $\mathsf{\text{-}ack}$ event for any preceding event of any of these kinds. (2) For every x and c, at most one $\mathsf{recon}(*, c)_{x,*}$ event occurs. (Configuration ids that are proposed in recon events are unique. This is not a serious restriction, because the same membership and quorum sets may be associated with different configuration ids.) (3) For every c, c', x, and i, if a $\mathsf{recon}(c, c')_{x,i}$ event occurs, then it is preceded by a $\mathsf{report}(c)_{x,i}$ event and by a $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,j}$ event for every $j \in \mathit{members}(c')$.

The safety guarantees provided by the service are as follows:

Well-formedness: For every x and i: (a) No $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$, $\mathsf{read\text{-}ack}(*)_{x,i}$, $\mathsf{write\text{-}ack}_{x,i}$, $\mathsf{recon\text{-}ack}(*)_{x,i}$, or $\mathsf{report}(*)_{x,i}$ event is preceded by a fail_i event. (b) Any $\mathsf{join\text{-}ack}(\mathsf{rambo})_{x,i}$ (resp., $\mathsf{read\text{-}ack}(*)_{x,i}$, $\mathsf{write\text{-}ack}_{x,i}$, $\mathsf{recon\text{-}ack}(*)_{x,i}$) event has a preceding $\mathsf{join}(\mathsf{rambo}, *)_{x,i}$ (resp., $\mathsf{read}_{x,i}$, $\mathsf{write}(*)_{x,i}$, $\mathsf{recon}(*, *)_{x,i}$) event with no intervening invocation or response action for x and i.

Atomicity:[1] If all the read and write operations that are invoked complete, then the read and write operations for object x can be partially ordered by an ordering $\prec$, so that the following conditions are satisfied: (1) No operation has infinitely many other operations ordered before it. (2) The partial order is consistent with the external order of invocations and responses, that is, there do not exist read or write operations π_1 and π_2 such that π_1 completes before π_2 starts, yet $\pi_2 \prec \pi_1$. (3) All write operations are totally ordered and every read operation is ordered with respect to all the writes. (4) Every read operation ordered after any writes returns the value of the last write preceding it in the partial order; any read operation ordered before all writes returns $(v_0)_x$.

The rest of the paper presents our implementation of RAMBO. The implementation is a distributed algorithm in the asynchronous message-passing model. All processes may communicate with each other. Processes may fail by stopping without warning.

Our implementation can be described formally as the composition of a separate implementation for each x, so we describe the implementation for a generic x. We suppress explicit mention of x, writing V, v_0, c_0, and i_0 as shorthand for V_x, $(v_0)_x$, $(c_0)_x$, and $(i_0)_x$, respectively.

[1] Atomicity is often defined in terms of an equivalence with a serial memory. The definition given here implies this equivalence, as shown, for example, in Lemma 13.16 in [18]. Although Lemma 13.16 of [18] is presented for a setting with only finitely many locations, nothing in Lemma 13.16 or its proof depends on the finiteness of the set of locations.

4 Reconfiguration Service Specification

Our RAMBO implementation for object x consists of a main RW algorithm and a reconfiguration service, $Recon$. Here we present the specification for $Recon$. Our implementation of $Recon$ is described in Section 7.

The external signature appears in Figure 2. The client of $Recon$ at location i requests to join the reconfiguration service by performing a $\mathsf{join}(\mathsf{recon})_i$ input action. The service acknowledges this with a corresponding $\mathsf{join\text{-}ack}_i$ output action. The client requests reconfiguration using a recon_i input, which is acknowledged with a $\mathsf{recon\text{-}ack}_i$ output action. The service reports a new configuration to the client using a report_i output action. Outputs of the form $\mathsf{new\text{-}config}(c,k)_i$ announce at location i that c is the k^{th} configuration id. These outputs are used for communication with the portion of the RW algorithm running at location i. $Recon$ announces consistent information, only one configuration id for each index in the configuration id sequence. $Recon$ delivers information about each configuration to members of the new configuration and members of the immediately preceding configuration. Crashes are modeled using fail actions.

Input:	Output:
$\mathsf{join}(\mathsf{recon})_i, i \in I$	$\mathsf{join\text{-}ack}(\mathsf{recon})_i, i \in I$
$\mathsf{recon}(c,c')_i, c, c' \in C, i \in members(c)$	$\mathsf{recon\text{-}ack}(b)_i, b \in \{\mathsf{ok}, \mathsf{nok}\}, i \in I$
$\mathsf{fail}_i, i \in I$	$\mathsf{report}(c)_i, c \in C, i \in I$
	$\mathsf{new\text{-}config}(c,k)_i, c \in C, k \in \mathbb{N}^+, i \in I$

Fig. 2. $Recon$: External signature

The set of traces describing $Recon$'s safety properties is defined by environment assumptions and service guarantees. The environment assumptions are simple well-formedness conditions, consistent with those for RAMBO:

Well-formedness: (1) For every i: (a) No $\mathsf{join}(\mathsf{recon})_i$ or $\mathsf{recon}(*,*)_i$ event is preceded by a fail_i event. (b) At most one $\mathsf{join}(\mathsf{recon})_i$ event occurs. (c) Any $\mathsf{recon}(*,*)_i$ event is preceded by a $\mathsf{join\text{-}ack}(\mathsf{recon})_i$ event. (d) Any $\mathsf{recon}(*,*)_i$ event is preceded by an -ack for any preceding $\mathsf{recon}(*,*)_i$ event. (2) For every c, at most one $\mathsf{recon}(*,c)_*$ event occurs. (3) For every c, c', x, and i, if a $\mathsf{recon}(c,c')_i$ event occurs, then it is preceded by: (a) A $\mathsf{report}(c)_i$ event, and (b) A $\mathsf{join\text{-}ack}(\mathsf{recon})_j$ for every $j \in members(c')$.

The safety guarantees are:

Well-formedness: For every i: (a) No $\mathsf{join\text{-}ack}(\mathsf{recon})_i$, $\mathsf{recon\text{-}ack}(*)_i$, $\mathsf{report}(*)_i$, or $\mathsf{new\text{-}config}(*,*)_i$ event is preceded by a fail_i event. (b) Any $\mathsf{join\text{-}ack}(\mathsf{recon})_i$ (resp., $\mathsf{recon\text{-}ack}(c)_i$) event has a preceding $\mathsf{join}(\mathsf{recon})_i$ (resp., recon_i) event with no intervening invocation or response action for x and i.

Agreement: If $\mathsf{new\text{-}config}(c,k)_i$ and $\mathsf{new\text{-}config}(c',k)_j$ both occur, then $c = c'$.

Validity: If $\mathsf{new\text{-}config}(c,k)_i$ occurs, then it is preceded by a $\mathsf{recon}(*,c)_{i'}$ for some i' for which a matching $\mathsf{recon\text{-}ack}(\mathsf{nok})_{i'}$ does not occur.

No duplication: If $\mathsf{new\text{-}config}(c,k)_i$ and $\mathsf{new\text{-}config}(c,k')_{i'}$ both occur, then $k = k'$.

5 Implementation of RAMBO Using a Reconfiguration Service

Our RAMBO implementation includes, for each i, a $Joiner_i$ automaton, which handles joining, and a RW_i automaton, which handles reading, writing, and "installing" new configurations. These automata use asynchronous communication channels $Channel_{i,j}$. The RW automata also interact with an arbitrary implementation of $Recon$.

5.1 Joiner Automata

When $Joiner_i$ receives a join(rambo, J) request from its environment, it sends join messages to the processes in J (with the hope that they are already participating, and so can help in its attempt to join). Also, it submits join requests to the local RW and $Recon$ components and waits for acknowledgments. The join messages that are sent by $Joiner$ automata are handled by RW automata at other locations.

5.2 Reader-Writer Automata

RW_i processes each read or write operation using one or more configurations, which it learns about from the $Recon$ service. It also handles the garbage-collection of older configurations. The signature and state of RW_i appear in Figure 3. Figure 4 presents the transitions pertaining to joining the protocol and failing. Figure 5 presents those pertaining to reading and writing, and Figure 6 presents those pertaining to garbage-collection.

Signature:

Input:
- read$_i$
- write$(v)_i$, $v \in V$
- new-config$(c, k)_i$, $c \in C$, $k \in \mathbb{N}^+$
- recv(join)$_{j,i}$, $j \in I - \{i\}$
- recv$(m)_{j,i}$, $m \in M$, $j \in I$
- join(rw)$_i$
- fail$_i$

Output:
- join-ack(rw)$_i$
- read-ack$(v)_i$, $v \in V$
- write-ack$_i$
- send$(m)_{i,j}$, $m \in M$, $j \in I$

Internal:
- query-fix$_i$
- prop-fix$_i$
- gc$(k)_i$, $k \in \mathbb{N}$
- gc-query-fix$(k)_i$, $k \in \mathbb{N}$
- gc-prop-fix$(k)_i$, $k \in \mathbb{N}$
- gc-ack$(k)_i$, $k \in \mathbb{N}$

State:

$status \in$ {idle, joining, active}, initially idle
$world$, a finite subset of I, initially $\emptyset$
$value \in V$, initially v_0
$tag \in T$, initially $(0, i_0)$
$cmap \in CMap$, initially
  $cmap(0) = c_0$,
  $cmap(k) = \bot$ for $k \geq 1$
$pnum1 \in \mathbb{N}$, initially 0
$pnum2 \in I \rightarrow \mathbb{N}$,
  initially everywhere 0
$failed$, a Boolean, initially $false$

op, a record with fields:
  $type \in$ {read, write}
  $phase \in$ {idle, query, prop, done}, initially idle
  $pnum \in \mathbb{N}$
  $cmap \in CMap$
  acc, a finite subset of I
  $value \in V$

gc, a record with fields:
  $phase \in$ {idle, query, prop}, initially idle
  $pnum \in \mathbb{N}$
  acc, a finite subset of I
  $index \in \mathbb{N}$

Fig. 3. RW_i: Signature and state

State variables. The $status$ variable keeps track of the progress of the component as it joins the protocol. When $status =$ idle, RW_i does not respond to any inputs (except for join) and does not perform any locally controlled actions. When $status$ = joining, RW_i responds to inputs but still does not perform any locally controlled actions. When $status$ = active, the automaton participates fully in the protocol.

The $world$ variable is used to keep track of all processes that are known to have tried to join the system. The $value$ variable contains the current value of the local replica of x, and tag holds the associated tag.

The $cmap$ variable contains information about configurations. If $cmap(k) = \bot$, it means that RW_i has not yet learned what the k^{th} configuration id is. If $cmap(k) = c \in C$, it means that RW_i has learned that the k^{th} configuration id is c, and has not yet garbage-collected it. If $cmap(k) = \pm$, it means that RW_i has garbage-collected the k^{th} configuration id. RW_i learns about configuration ids either directly from the $Recon$ service, or from other RW processes. The value of $cmap$ is always in $Usable$, that is, $\pm$ for some finite prefix of $\mathbb{N}$, followed by an element of C, followed by elements of $C_\bot$, with only finitely many elements of C. When RW_i processes a read or write operation, it uses all the configurations whose ids appear in its $cmap$, up to the first $\bot$.

The $pnum1$ variable and $pnum2$ array are used to implement a handshake that identifies "recent" messages. RW_i uses $pnum1$ to count the total number of operation phases (either query or propagation phases) that it has initiated overall, including phases occurring in read, write, and garbage-collection operations. For every j, including $j = i$, RW_i uses $pnum2(j)$ to record the largest number of a phase that i has learned that j has started, via a message from j to i. Finally, two records, op and gc, are used to maintain information about locally-initiated read, write, and garbage-collection operations.

Joining and failure transitions. When a join(rw)$_i$ input occurs when $status$ = idle, if i is the object's creator i_0, then $status$ immediately becomes active, which means that RW_i is ready for full participation in the protocol. Otherwise, $status$ becomes joining, which means that RW_i is receptive to inputs but not ready to perform any locally controlled actions. In either case, RW_i records itself as a member of its own $world$. From this point on, RW_i adds to its $world$ any process from which it receives a join message. (Recall that these join messages are sent by $Joiner$ automata.)

If $status$ = joining, then $status$ becomes active when RW_i receives a message from another RW process. (The code for this appears in the recv transition definition in Figure 5.) At this point, process i has acquired enough information to begin participating fully. After $status$ becomes active, process i can perform a join-ack(rw).

Information propagation transitions. Information is propagated between RW processes in the background, via point-to-point channels that are accessed using the send and recv actions. The algorithm uses one kind of message, which contains a tuple consisting of the sender's $world$, its latest known $value$ and tag, its $cmap$, and two phase numbers—the current phase number of the sender, $pnum1$, and the latest known phase number of the receiver, from the $pnum2$ array. These messages may be sent at any time, to processes in the sender's $world$.

When RW_i receives a message, it sets its $status$ to active, if it has not already done so. It adds incoming world information, in W, to its $world$ set. It compares the incoming tag t to its own tag. If t is strictly greater, it represents a more recent version of the object; in this case, RW_i sets its tag to t and its $value$ to the incoming value v. RW_i also updates its $cmap$ with the information in the incoming $CMap$, cm, using the $update$ operator

```
Input join(rw)$_i$
Effect:
   if $\neg failed$ then
     if $status$ = idle then
       if $i = i_0$ then
         $status \leftarrow$ active
       else
         $status \leftarrow$ joining
       $world \leftarrow world \cup \{i\}$

Input recv(join)$_{j,i}$
Effect:
   if $\neg failed$ then
     if $status \neq$ idle then
       $world \leftarrow world \cup \{j\}$

Output join-ack(rw)$_i$
Precondition:
   $\neg failed$
   $status$ = active
Effect:
   none

Input fail$_i$
Effect:
   $failed \leftarrow$ true
```

Fig. 4. RW_i: Join-related and failure transitions

defined in Section 2. That is, for each k, if $cmap(k) = \bot$ and $cm(k) \in C$, process i sets its $cmap(k)$ to $cm(k)$. Also, if $cmap(k) \in C_\bot$ and $cm(k) = \pm$, indicating that the sender knows that configuration k has already been garbage-collected, then RW_i sets its $cmap(k)$ to $\pm$. RW_i also updates its $pnum2(j)$ component for the sender j to reflect new information about j's phase number, which appears in the pns component of the message.

While RW_i is conducting a phase of a read, write, or garbage-collection operation, it verifies that the incoming message is "recent", in the sense that the sender j sent it after j received a message from i that was sent after i began the current phase. RW_i uses the phase numbers to perform this check: it checks that the incoming phase number pnr is at least as large as the current operation phase number ($op.pnum$ or $gc.pnum$). If the message is recent, then RW_i uses it to update the op or gc record.

Read and write operations. A read or write operation is performed in two phases: a query phase and a propagation phase. In each phase, RW_i obtains recent $value$, tag, and $cmap$ information from "enough" processes. This information is obtained by sending and receiving messages, as described above.

When RW_i starts a phase of a read or write, it sets $op.cmap$ to a $CMap$ whose configurations are to be used to conduct the phase. Specifically, RW_i uses $truncate(cmap)$, which is defined to include all the configuration ids in $cmap$ up to the first $\bot$. When a new $CMap$ cm is received during the phase, $op.cmap$ is "extended" by adding all newly-discovered configuration ids, up to the first $\bot$ in cm. If adding these new configuration ids does not create a "gap", that is, if the extended $op.cmap$ is in $Truncated$, then the phase continues using the extended $op.cmap$. On the other hand, if adding these new configuration ids creates a gap, then RW_i can infer that it has been using out-of-date configuration ids. In this case, it restarts the phase using the best currently known $CMap$, which is obtained by computing $truncate(cmap)$ for the latest local $cmap$.

In between restarts, while RW_i is engaged in a single attempt to complete a phase, it never removes a configuration id from $op.cmap$. In particular, if process i learns during a phase that a configuration id in $op.cmap(k)$ has been garbage-collected, it does not remove it from $op.cmap$, but continues to include it in conducting the phase.

```
Output $\mathsf{send}(\langle W, v, t, cm, pns, pnr\rangle)_{i,j}$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $j \in world$
  $\langle W, v, t, cm, pns, pnr\rangle =$
    $\langle world, value, tag, cmap, pnum1, pnum2(j)\rangle$
Effect:
  none

Input $\mathsf{recv}(\langle W, v, t, cm, pns, pnr\rangle)_{j,i}$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $status \leftarrow \mathsf{active}$
      $world \leftarrow world \cup W$
      if $t > tag$ then $(value, tag) \leftarrow (v, t)$
      $cmap \leftarrow update(cmap, cm)$
      $pnum2(j) \leftarrow \max(pnum2(j), pns)$
      if $op.phase \in \{\mathsf{query}, \mathsf{prop}\}$ and $pnr \geq op.pnum$ then
        $op.cmap \leftarrow extend(op.cmap, truncate(cm))$
        if $op.cmap \in Truncated$ then
          $op.acc \leftarrow op.acc \cup \{j\}$
        else
          $op.acc \leftarrow \emptyset$
          $op.cmap \leftarrow truncate(cmap)$
      if $gc.phase \in \{\mathsf{query}, \mathsf{prop}\}$ and $pnr \geq gc.pnum$ then
        $gc.acc \leftarrow gc.acc \cup \{j\}$

Input $\mathsf{new\text{-}config}(c, k)_i$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $cmap(k) \leftarrow update(cmap(k), c)$

Input $\mathsf{read}_i$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $pnum1 \leftarrow pnum1 + 1$
      $\langle op.pnum, op.type, op.phase, op.cmap, op.acc\rangle$
        $\leftarrow \langle pnum1, \mathsf{read}, \mathsf{query}, truncate(cmap), \emptyset\rangle$

Input $\mathsf{write}(v)_i$
Effect:
  if $\neg failed$ then
    if $status \neq \mathsf{idle}$ then
      $pnum1 \leftarrow pnum1 + 1$
      $\langle op.pnum, op.type, op.phase, op.cmap, op.acc,$
       $op.value\rangle$
        $\leftarrow \langle pnum1, \mathsf{write}, \mathsf{query}, truncate(cmap), \emptyset, v\rangle$

Internal $\mathsf{query\text{-}fix}_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type \in \{\mathsf{read}, \mathsf{write}\}$
  $op.phase = \mathsf{query}$
  $\forall k \in \mathbb{N}, c \in C : (op.cmap(k) = c)$
    $\Rightarrow (\exists R \in read\text{-}quorums(c) :$
        $R \subseteq op.acc)$
Effect:
  if $op.type = \mathsf{read}$ then
    $op.value \leftarrow value$
  else
    $value \leftarrow op.value$
    $tag \leftarrow \langle tag.seq + 1, i\rangle$
  $pnum1 \leftarrow pnum1 + 1$
  $op.pnum \leftarrow pnum1$
  $op.phase \leftarrow \mathsf{prop}$
  $op.cmap \leftarrow truncate(cmap)$
  $op.acc \leftarrow \emptyset$

Internal $\mathsf{prop\text{-}fix}_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type \in \{\mathsf{read}, \mathsf{write}\}$
  $op.phase = \mathsf{prop}$
  $\forall k \in \mathbb{N}, c \in C : (op.cmap(k) = c)$
    $\Rightarrow (\exists W \in write\text{-}quorums(c) :$
        $W \subseteq op.acc)$
Effect:
  $op.phase = \mathsf{done}$

Output $\mathsf{read\text{-}ack}(v)_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type = \mathsf{read}$
  $op.phase = \mathsf{done}$
  $v = op.value$
Effect:
  $op.phase = \mathsf{idle}$

Output $\mathsf{write\text{-}ack}_i$
Precondition:
  $\neg failed$
  $status = \mathsf{active}$
  $op.type = \mathsf{write}$
  $op.phase = \mathsf{done}$
Effect:
  $op.phase = \mathsf{idle}$
```

Fig. 5. RW_i: Read/write transitions

The query phase of a read or write operation terminates when a *query fixed point* is reached. This happens when RW_i determines that it has received recent responses from some read-quorum of each configuration in its current $op.cmap$. Then t, defined to be RW_i's tag at the query fixed point, is at least as great as the tag value that each process in each of these read-quorums had at the start of the query phase.

If the operation is a read, then RW_i determines at this point that its current $value$ is the value to be returned to its client. However, before returning, RW_i performs the propagation phase, whose purpose is to make sure that "enough" RW processes have acquired tags that are at least t. Again, the information is propagated in the background, and $op.cmap$ is managed as described above. The propagation phase ends once a *propagation fixed point* is reached, when RW_i has received recent responses from some write-quorum of each configuration in the current $op.cmap$. When this occurs, the tag of each process in each of these write-quorums is at least t.

Processing for a write operation starting with $\mathsf{write}(v)_i$ is similar to that for a read. The query phase is conducted exactly as for a read, but processing after the query fixed point is different: Suppose t, process i's tag at the query fixed point, is of the form (n, j). Then RW_i defines the tag for its write operation to be the pair $(n+1, i)$ and sets its tag to $(n+1, i)$ and its $value$ to the new value v. Then it performs its propagation phase. Now the purpose of the propagation phase is to ensure that "enough" processes acquire tags that are at least as great as $(n+1, i)$. The propagation phase is conducted exactly as for a read.

```
Internal $\mathsf{gc}(k)_i$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $gc.phase = \mathsf{idle}$
    $cmap(k) \in C$
    $cmap(k+1) \in C$
    $k = 0$ or $cmap(k-1) = \pm$
Effect:
    $pnum1 \leftarrow pnum1 + 1$
    $\langle gc.pnum, gc.phase, gc.acc, gc.index\rangle$
        $\leftarrow \langle pnum1, \mathsf{query}, \emptyset, k\rangle$

Internal $\mathsf{gc\text{-}query\text{-}fix}(k)_i$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $gc.phase = \mathsf{query}$
    $gc.index = k$
    $\exists R \in read\text{-}quorums(cmap(k))$ :
      $\exists W \in write\text{-}quorums(cmap(k))$ : $R \cup W \subseteq gc.acc$
Effect:
    $pnum1 \leftarrow pnum1 + 1$
    $\langle gc.pnum, gc.phase, gc.acc\rangle \leftarrow \langle pnum1, \mathsf{prop}, \emptyset\rangle$

Internal $\mathsf{gc\text{-}prop\text{-}fix}(k)_i$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $gc.phase = \mathsf{prop}$
    $gc.index = k$
    $\exists W \in write\text{-}quorums(cmap(k+1))$ :
      $W \subseteq gc.acc$
Effect:
    $cmap(k) \leftarrow \pm$

Internal $\mathsf{gc\text{-}ack}(k)_i$
Precondition:
    $\neg failed$
    $status = \mathsf{active}$
    $gc.index = k$
    $cmap(k) = \pm$
Effect:
    $gc.phase = \mathsf{idle}$
```

Fig. 6. RW_i: Garbage-collection transitions

New configurations and garbage collection. When RW_i learns about a new configuration id via a $\mathsf{new\text{-}config}$ input action, it simply records it in its $cmap$. From time to time, configuration ids get garbage-collected at i, in numerical order. The configuration ids used in performing query and propagation phases of reads and writes are those in $truncate(cmap)$, that is, all configurations that have not been garbage-collected and that appear before the first $\perp$.

There are two situations in which RW_i may garbage-collect the configuration id in $cmap(k)$. First, RW_i can garbage-collect $cmap(k)$ if it learns that another process has already garbage-collected it. This happens when RW_i receives a message in which $cm(k) = \pm$. Second, RW_i may acquire enough information to garbage-collect configuration k on its own. RW_i accomplishes this by performing a a two-phase garbage-collection operation, with a structure similar to the read and write operations. RW_i may initiate garbage-collection of configuration k when its $cmap(k)$ and $cmap(k+1)$ are both in C, and when any configurations with indices smaller than $k-1$ have already been garbage-collected. Garbage-collection may proceed concurrently with a read or write operation at the same node.

In the query phase of a garbage-collection operation, RW_i communicates with a read-quorum and a write-quorum of configuration k. The query phase accomplishes two tasks: First, RW_i ensures that all the processes in the read-quorum and write-quorum learn about configurations k and $k+1$, and also learn that all configurations smaller than k have been garbage-collected. If such a process, j, is contacted afterwards by someone who is using configuration k, j can tell that process about configuration $k+1$. Second, in the query phase, RW_i collects tag and $value$ information from the read-quorum and write-quorum. This ensures that, by the end of the query phase, RW_i's tag, t, is at least as great as the tag that each of the quorum members had when it sent a message to RW_i for the query phase. In the propagation phase, RW_i ensures that all the processes in a write-quorum of configuration $k+1$ have acquired tags that are at least t. Note that, unlike a read or write operation, a garbage-collection for k uses only two configurations—k in the query phase and $k+1$ in the propagation phase.

At any time while RW_i is garbage-collecting configuration k, it may discover that someone has already garbage-collected k; it discovers this by observing that $cmap(k) = \pm$. When this happens, RW_i may simply terminate its garbage-collection.

5.3 The Complete Algorithm

We assume point to point channels $Channel_{i,j}$, one for each $i, j \in I$ (including $i = j$). $Channel_{i,j}$ is accessed using $\mathsf{send}(m)_{i,j}$ input actions, by which a sender at location i submits message m to the channel, and $\mathsf{recv}(m)_{i,j}$ output actions, by which a receiver at location j receives m. Channels may lose and reorder messages, but cannot manufacture new messages or duplicate messages. Formally, we model $Channel_{i,j}$ as a multiset, where A $\mathsf{send}(m)_{i,j}$ input action adds one copy of m to the multiset and A $\mathsf{recv}(m)_{i,j}$ output removes one copy of m. A lose input action allows any sub-multiset of messages to be removed.

The complete implementation, which we call $\mathcal{S}$, is the composition of the $Joiner_i$, RW_i, and $Channel_{i,j}$ automata, and any automaton whose traces satisfy the $Recon$ safety specification, with all actions that are not external actions of RAMBO hidden.

6 Safety Proof

We show that $\mathcal{S}$ satisfies the safety guarantees of RAMBO, as given in Section 3, assuming the environment safety assumptions. An *operation* can be of type read, write, or garbage-collection. An operations is uniquely identified by its starting event: read_i, $\mathsf{write}(v)_i$, or $\mathsf{gc}(*)_i$ event.

We introduce the following history variables: (1) For every $k \in \mathbb{N}$: $c(k) \in C$. This is set when the first $\mathsf{new\text{-}config}(*, k)_*$ occurs, to the configuration id that appears as the first parameter of this action. (2) For every operation π: $tag(\pi) \in T$. This is set just after π's query-fix or gc-query-fix event, to the tag of the process running π. (3) For every read or write operation π: (a) $query\text{-}cmap(\pi)$, a $CMap$. This is set in the query-fix step of π, to the value of $op.cmap$ in the pre-state. (b) $prop\text{-}cmap(\pi)$, a $CMap$. This is set in the prop-fix step of π, to the value of $op.cmap$ in the pre-state.

For any read or write operation π, we designate the following events: (1) query-phase-start(π). This is defined in the query-fix step of π, to be the unique earlier event at which the collection of query results was started and not subsequently restarted (that is, $op.acc$ is set to $\emptyset$ in the effects of the corresponding step, and $op.acc$ is not later reset to $\emptyset$ following that event and prior to the query-fix step). This is either a read, write, or recv event. (2) prop-phase-start(π). This is defined in the prop-fix step of π, to be the unique earlier event at which the collection of propagation results was started and not subsequently restarted. This is either a query-fix or recv event.

Now we present several lemmas describing information flow between operations. All are stated for a generic execution α satisfying the environment assumptions. The first lemma describes information flow between garbage-collection operations. We say that a gc-prop-fix$(k)_i$ event is *initial* if it is the first gc-prop-fix$(k)_*$ event in α, and a garbage-collection operation is *initial* if its gc-prop-fix event is initial. The algorithm ensures that garbage-collection of successive configurations is sequential, in fact, for each k, the initial gc-prop-fix(k) event precedes any attempt to garbage-collect $k+1$. Sequential garbage-collection implies that tags of garbage-collection operations are monotone with respect to the configuration indices:

Lemma 1. *Suppose γ_k and γ_ℓ are garbage-collection operations for k and ℓ, respectively, where $k \leq \ell$ and γ_k is initial. Suppose a* gc-query-fix(ℓ) *event for γ_ℓ occurs in α. Then $tag(\gamma_k) \leq tag(\gamma_\ell)$.*

Proof. By induction on ℓ, for fixed k. For the inductive step, assume that $\ell \geq k+1$ and the result is true for $\ell - 1$. A write-quorum of $c(\ell)$ is used in the propagation phase of $\gamma_{\ell-1}$ and a read-quorum of $c(\ell)$ is used in the query phase of γ_ℓ. The quorum intersection property for $c(\ell)$ guarantees propagation of tag information. □

The following lemma describes situations in which certain configurations must appear in the $query\text{-}cmap$ of a read or write operation π. First, if no garbage-collection operation for k completes before the query-phase-start event of π, then some configuration with index $\leq k$ must be included in $query\text{-}cmap(\pi)$. Second, if some garbage-collection for k completes before the query-phase-start event of π, then some configuration with index $\geq k+1$ must be included in the $query\text{-}cmap(\pi)$.

Lemma 2. *Let π be a read or write operation whose* query-fix *event occurs in α. (1) If no* gc-prop-fix(k) *event precedes the* query-phase-start(π) *event, then $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \leq k$. (2) If some* gc-prop-fix(k) *event precedes the* query-phase-start(π) *event, then $query\text{-}cmap(\pi)(\ell) \in C$ for some $\ell \geq k+1$.*

The next lemma describes propagation of tag information from a garbage-collection operation to a following read or write operation.

Lemma 3. *Let γ be an initial garbage-collection operation for k. Let π be a read or write operation whose* query-fix *event occurs in α. Suppose that the* gc-prop-fix(k) *event of γ precedes the* query-phase-start(π) *event. Then $tag(\gamma) \leq tag(\pi)$, and if π is a write operation then $tag(\gamma) < tag(\pi)$.*

The next two lemmas describe relationships between reads and writes that execute sequentially. The first lemma says that the smallest configuration index used in the propagation phase of the first operation is less than or equal to the largest index used in the query phase of the second operation. In other words, the second operation's query phase cannot use only configurations with indices that are less than any used in the first operation's propagation phase.

Lemma 4. *Assume π_1 and π_2 are two read or write operations such that the* prop-fix *event of π_1 precedes the* query-phase-start(π_2) *event in α.*
Then $\min(\{\ell : \textit{prop-cmap}(\pi_1)(\ell) \in C\}) \leq \max(\{\ell : \textit{query-cmap}(\pi_2)(\ell) \in C\})$.

Proof. Suppose not. Let $k = \max(\{\ell : \textit{query-cmap}(\pi_2)(\ell) \in C\})$. Then some gc-prop-fix($k$) event occurs before the prop-fix of π_1, and so before the query-phase-start(π_2) event. Lemma 2, Part 2, then implies that $\textit{query-cmap}(\pi_2)(\ell) \in C$ for some $\ell \geq k+1$, which contradicts the choice of k. □

The second lemma describes propagation of *tag* information between sequential reads and writes.

Lemma 5. *Suppose π_1 and π_2 are two read or write operations, such that the* prop-fix *event of π_1 precedes the* query-phase-start(π_2) *event in α. Then $tag(\pi_1) \leq tag(\pi_2)$, and if π_2 is a write then $tag(\pi_1) < tag(\pi_2)$.*

Proof. Let i_1 and i_2 be the processes that run operations π_1 and π_2, respectively. Let $cm_1 = \textit{prop-cmap}(\pi_1)$ and $cm_2 = \textit{query-cmap}(\pi_2)$. If there exists k such that $cm_1(k) \in C$ and $cm_2(k) \in C$, then the quorum intersection property for configuration k implies the conclusions of the lemma. So we assume that no such k exists. Lemma 4 implies that $\min(\{\ell : cm_1(\ell) \in C\}) \leq \max(\{\ell : cm_2(\ell) \in C\})$. Since the set of indices used in each phase consists of consecutive integers and the intervals have no indices in common, it follows that $k_1 < k_2$, where $k_1 = \max(\{\ell : cm_1(\ell) \in C\})$ and $k_2 = \min(\{\ell : cm_2(\ell) \in C\})$.

Since, for every $k \leq k_2 - 1$, $query.cmap(\pi_2)(k) \notin C$, Lemma 2, Part 1, implies that, for every such k, a gc-prop-fix(k) event occurs before the query-phase-start(π_2) event. For each such k, define γ_k to be the initial garbage-collection operation for k.

The propagation phase of π_1 accesses a write-quorum of $c(k_1)$, and the query phase of γ_{k_1} accesses a read-quorum of $c(k_1)$. By the quorum intersection property, there is some j in the intersection of these quorums. Let message m be the message sent from j to i_1 in the propagation phase of π_1, and let m' be the message sent from j to the process running γ_{k_1} in its query phase. We claim that j sends m before it sends m'. For if not, then information about configuration $k_1 + 1$ would be conveyed by j to i_1, who would include it in cm_1, contradicting the choice of k_1. Since j sends m before it sends m', j conveys *tag* information from π_1 to γ_{k_1}, ensuring that $tag(\pi_1) \leq tag(\gamma_{k_1})$.

Since $k_1 \leq k_2 - 1$, Lemma 1 implies that $tag(\gamma_{k_1}) \leq tag(\gamma_{k_2-1})$. Lemma 3 implies that $tag(\gamma_{k_2-1}) \leq tag(\pi_2)$, and if π_2 is a write then $tag(\gamma_{k_2-1}) < tag(\pi_2)$. Combining all the inequalities then yields both conclusions. □

Theorem 1. *Let β be a trace of $\mathcal{S}$. If β satisfy the* RAMBO *environment assumptions, then β satisfies the* RAMBO *service guarantees (well-formedness and atomicity).*

Proof. Let β be a trace of $\mathcal{S}$ that satisfies the RAMBO environment assumptions. We argue that β satisfies the RAMBO service guarantees. The proof that β satisfies the RAMBO well-formedness guarantees is straightforward from the code. To show that β satisfies atomicity (as defined in Section 3), assume that all read and write operations complete in β. Let α be an execution of $\mathcal{S}$ that satisfies the environment assumptions and whose trace is β. Define a partial order $\prec$ on read and write operations in α: totally order the writes in order of their tags, and order each read with respect to all the writes so that a read with tag t is ordered after all writes with tags $\leq t$ and before all writes with tags $> t$. Then we claim that $\prec$ satisfies the four conditions in the definition of atomicity. The interesting condition is Condition 2; the other three are straightforward.

For Condition 2, suppose for the sake of contradiction that π_1 and π_2 are read or write operations, π completes before π_2 starts, and $\pi_2 \prec \pi_1$. If π_2 is a write operation, then since π_1 completes before π_2 starts, Lemma 5 implies that $tag(\pi_2) > tag(\pi_1)$. But the fact that $\pi_2 \prec \pi_1$ implies that $tag(\pi_2) \leq tag(\pi_1)$, yielding a contradiction. On the other hand, if π_2 is a read operation, then since π_1 completes before π_2 starts, Lemma 5 implies that $tag(\pi_2) \geq tag(\pi_1)$. But the fact that $\pi_2 \prec \pi_1$ implies that $tag(\pi_2) < tag(\pi_1)$, again yielding a contradiction. □

7 Implementation of the Reconfiguration Service

The *Recon* algorithm is considerably simpler than the *RW* algorithm. It consists of a $Recon_i$ automaton for each location i, which interacts with a collection of global consensus services $Cons(k, c)$, one for each $k \geq 1$ and each $c \in C$, and with a point-to-point communication service.

$Cons(k, c)$ accepts inputs from members of configuration c, which it assumes to be the $k - 1^{st}$ configuration. These inputs are of the form $\mathsf{init}(c')_{k,c,i}$, where c' is a proposed new configuration. The configuration that $Cons(k, c)$ decides upon (using $\mathsf{decide}(c')_{k,c,i}$ outputs) is deemed to be the k^{th} configuration. The validity property of consensus implies that this decision is one of the proposed configurations.

$Recon_i$ is activated by a $\mathsf{join}(\mathsf{recon})_i$ action, which is an output of $Joiner_i$. $Recon_i$ accepts reconfiguration requests from clients, and initiates consensus to help determine new configurations. It records the new configurations that the consensus services determine. $Recon_i$ also informs RW_i about newly-determined configurations, and disseminates information about newly-determined configurations to the members of those configurations. It returns acknowledgments and configuration reports to its client.

We implement $Cons(k, c)$ using the Paxos consensus algorithm [17], as described formally in [8]. Our complete implementation of $Recon$, $Recon_{impl}$, consists of the

$Recon_i$ automata, channels connecting all the $Recon_i$ automata, and the implementations of the $Cons$ services. We use the same kinds of channels as for RAMBO: point-to-point channels that may lose and reorder messages, but not manufacture new messages or duplicate messages. The complete RAMBO system (for a particular object) consists of $Joiner$, RW, and channel automata as described in Section 5, plus $Recon_{impl}$. We call the complete RAMBO system $\mathcal{S}'$.

8 Conditional Performance Analysis: Latency Bounds

We prove latency bounds for the full system $\mathcal{S}'$. To handle timing, we convert all the I/O automata to general timed automata (GTAs) as defined in [18], by allowing arbitrary amounts of time to pass in any state. Fix $d > 0$, the *normal message delay*, and $\epsilon > 0$.

RAMBO allows sending of messages at arbitrary times. For the purpose of latency analysis, we restrict RAMBO's sending pattern: We assume that each automaton has a local real-valued clock, and sends messages at the first possible time and at regular intervals of d thereafter, as measured on the local clock. Also, non-send locally controlled events occur just once, within time 0 on the local clock.

Our results also require restrictions on timing and failure behavior: We define an admissible timed execution to be *normal* provided that all local clocks progress at rate exactly 1, all messages that are sent are delivered within time d, and timing and failure behavior for all consensus services is "normal", as defined in [8].[2]

Next, we define a reliability property for configurations. In general, in quorum-based systems, operations that use quorums are guaranteed to terminate only if some quorums do not fail. Because we use many configurations, we attempt to take into account which configurations might be in use. We say that k is *installed* in a timed execution α provided that either $k = 0$ or there exists $c \in C$ such that (1) some $\mathsf{init}(*)_{k,c,*}$ event occurs, and (2) for every $i \in members(c)$, either $\mathsf{decide}(*)_{k,c,i}$ or fail_i occurs. (Thus, configuration $k-1$ is c, and every non-failed member of c has learned about configuration k.) We say that α is *e-configuration-viable*, $e \geq 0$, provided that for every c and k such that some $rec\text{-}cmap(k)_* = c$ in some state in α, there exist $R \in read\text{-}quorums(c)$ and $W \in write\text{-}quorums(c)$ such that either (1) no process in $R \cup W$ ever fails in α, or (2) $k+1$ is installed in a finite prefix α' of α and no process in $R \cup W$ fails in α by time $\ell time(\alpha') + e$. (Quorums remain non-failed for at least time e after the next configuration is installed.)

The e-configuration-viability assumption is useful in situations where a configuration is no longer needed for performing operations after time e after the next configuration is installed. This condition holds in RAMBO executions in which certain timing assumptions hold; the strength of those assumptions determines the value of e that must be considered. We believe that such an assumption is reasonable for a reconfigurable system, because it can be reconfigured when quorums appear to be in danger of failing.

We prove a bound of $2d$ on the time to join, and a bound of $11d + \epsilon$ for the time for reconfiguration, based on a bound of $10d + \epsilon$ for consensus. We also establish a

[2] This means that all messages are delivered within time d, local processing time is 0, and information is gossiped at intervals of d.

situation in which a system is guaranteed to produce a positive response to a reconfiguration request. We prove a bound of $4d$ on the time for garbage-collection, assuming that enough of the relevant processes remain non-failed. We prove a bound of $4d$ on the latency for read and write operations in a "quiescent" situation, in which all joins and configuration management events have stopped, and the configuration map of the operation's initiator includes the latest configuration and has value $\pm$ for all earlier configurations. More generally, we show that this bound holds even if this map contains more than one configuration: since the configurations are used concurrently, the use of multiple configurations does not slow the operation down.

We show that all participants succeed in exchanging information about configurations, within a short time: if i and j have joined at least time e ago and do not fail, then any information that i has about configurations is conveyed to j within time $2d$. Using this result, we show that, if reconfiguration requests are spaced sufficiently far apart, and if quorums of configurations remain alive for sufficiently long, then garbage collection keeps up with reconfiguration.

The main latency theorem bounds the time for read and write operations in the "steady-state" case, where reconfigurations do not stop, but are spaced sufficiently far apart. Fix $e \geq 0$.

Theorem 2. *Let α be a normal admissible timed execution of S' such that:*
(1) If a $\mathsf{recon}(*, c)_i$ *event occurs at time t then for every $j \in members(c)$,* $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ *occurs by time $t - e$. (2) If* $\mathsf{join\text{-}ack}(\mathsf{rambo})_i$ *and* $\mathsf{join\text{-}ack}(\mathsf{rambo})_j$ *both occur by time t, and neither i nor j fails by time $t+e$, then by time $t+e$, $i \in world_j$.*
(3) For any $\mathsf{recon}(c, *)_i$ *that occurs in α, the time since the corresponding* $\mathsf{report}(c)_i$ *event is $\geq 12d + \varepsilon$. (4) α satisfies $11d$-configuration-viability. (5) α contains* decide *events for infinitely many configurations.*
Suppose that a read_i *(resp.,* $\mathsf{write}(*)_i$*) event occurs at time t, and* $\mathsf{join\text{-}ack}_i$ *occurs strictly before time $t - (e + 8d)$. Then the corresponding* $\mathsf{read\text{-}ack}_i$ *(resp.,* $\mathsf{write\text{-}ack}(*)_i$*) event occurs by time $t + 8d$.*

Proof. The various spacing properties and bounds on time to disseminate information imply that each phase of the read or write completes with at most one restart for learning about a new configuration. Therefore, each phase takes time at most $4d$, for a total of $8d$.
□

In the full paper we also present latency results analogous to those described above, for executions that have normal timing and failure characteristics after some point in the execution. These results are similar to the previous results, but include dependence on the time when normal behavior begins.

9 Future Work

In future work, we plan to implement the complete *Rambo* algorithm in LAN, WAN, and mobile settings. We will extend our performance analysis and compare it with empirical results. We will investigate ways of increasing the concurrency of garbage-collection and of reducing the amount of communication. Finally, this work leaves open the question of how to choose good configurations, for various kinds of platforms.

References

1. *Communications of the ACM*, special section on group communications, vol. 39, no. 4, 1996.
2. Y. Amir, D. Dolev, P. Melliar-Smith and L. Moser, “Robust and Efficient Replication Using Group Communication” Tech. Rep. 94-20, Dept. of Computer Science, Hebrew Univ., 1994.
3. H. Attiya, A. Bar-Noy and D. Dolev, “Sharing Memory Robustly in Message Passing Systems", *J. of the ACM*, vol. 42, no. 1, pp. 124-142, 1996.
4. P.A. Bernstein, V. Hadzilacos and N. Goodman, “Concurrency Control and Recovery in Database Systems”, Addison-Wesley, Reading, MA, 1987.
5. F. Cristian and F. Schmuck, “Agreeing on Processor Group Membership in Asynchronous Distributed Systems”, TR. CSE95-428, Dept. of Comp. Sci., Univ. of California San Diego.
6. S.B. Davidson, H. Garcia-Molina and D. Skeen, “Consistency in Partitioned Networks", *ACM Computing Surveys*, vol. 15, no. 3, pp. 341-370, 1985.
7. R. De Prisco, A. Fekete, N. Lynch, A. Shvartsman, “A Dynamic Primary Configuration Group Communication Service”, *13th Int-l Conference of Distributed Computing*, 1999.
8. Roberto De Prisco, Nancy Lynch, Alex Shvartsman, Nicole Immorlica and Toh Ne Win “A Formal Treatment of Lamport’s Paxos Algorithm”, manuscript, 2002.
9. C. Dwork, N. A. Lynch, L. J. Stockmeyer, “Consensus in the presence of partial synchrony”, *J. of ACM*, 35(2), pp. 288-323, 1988.
10. B. Englert and A.A. Shvartsman, Graceful Quorum Reconfiguration in a Robust Emulation of Shared Memory, in *Proc. International Conference on Distributed Computer Systems* (ICDCS’2000), pp. 454-463, 2000.
11. A. Fekete, N. Lynch and A. Shvartsman “Specifying and using a partitionable group communication service”, *ACM Trans. on Computer Systems*, vol. 19, no. 2, pp. 171–216, 2001.
12. H. Garcia-Molina and D. Barbara, “How to Assign Votes in a Distributed System," *J. of the ACM*, vol. 32, no. 4, pp. 841-860, 1985.
13. D.K. Gifford, “Weighted Voting for Replicated Data", in *Proc. of 7th ACM Symp. on Oper. Sys. Princ.*, pp. 150-162, 1979.
14. M.P. Herlihy, “Dynamic Quorum Adjustment for Partitioned Data", *ACM Trans. on Database Systems*, 12(2), pp. 170-194, 1987.
15. R. Guerraoui and A. Schiper, “Consensus Service: A Modular Approach For Building Fault-Tolerant Agreement Protocols in Distributed Systems", *Proc. of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pp. 168-177, 1996.
16. I. Keidar and D. Dolev, “Efficient Message Ordering in Dynamic Networks”, in *Proc. of 15th Annual ACM Symp. on Princ. of Distr. Comput.*, pp. 68-76, 1996.
17. Leslie Lamport, "The Part-Time Parliament", *ACM Transactions on Computer Systems*, 16(2) 133-169, 1998.
18. N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
19. N. Lynch and A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *27th Int-l Symp. on Fault-Tolerant Comp.*, pp. 272-281, 1997.
20. Nancy Lynch and Alex Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. MIT-LCS-TR-856, 2002
21. E. Upfal and A. Wigderson, How to share memory in a distributed system, *Journal of the ACM*, 34(1):116–127, 1987.

Ad Hoc Membership for Scalable Applications

Tal Anker[1,2], Danny Dolev[2], and Ilya Shnayderman[2]

[1] Radlan Computer Communications, Israel
[2] School of Engineering and Computer Science., The Hebrew University of Jerusalem, Israel {anker,dolev,ilia}@cs.huji.ac.il

Abstract. The paper describes an ad hoc approach realized in a practical distributed transport layer called Xpand [1] to improve the message transmission service over a WAN. The current technology focuses on applications that require strong semantics. The ad hoc membership approach increases the asynchrony of handling both control and message flows in order to overcome membership changes with minimal effect on ongoing streams of messages. This approach is beneficial for a variety of applications. Its implementation is expandable to address stronger semantics for applications that need them.

1 Introduction

There are two major widely-recognized approaches toward building distributed data-based applications and replicating objects. The first approach, known as Group Communication Systems (GCSs) [2], presents powerful building blocks for supporting consistency and fault-tolerance in distributed applications, whereas the Paxos [3] approach focuses on ordering actions among a group of servers. While implementing either of the approaches, one observes the system's performance to degrade significantly as group size and message transmission volume increase. These performance problems become even worse in wide-area network environments where message latency is often high and unpredictable. Only few implementations specifically address WAN environment, e.g., Spread [4], Inter-Group [5] and *Xpand* [1]).

In this paper we present an ad hoc membership algorithm that is implemented in *Xpand* [1]. The membership services offered by Xpand are designed to be flexible. On one hand, they can be used for maintaining a group of participants and potential participants of a group by applications that do not need strong semantics.[1]. On the other hand, while focusing on efficiency, these services allow to consistently provide stronger semantics for applications that require it.

The efficiency of the ad hoc membership is a result of the separation it implies between message flow, membership algorithms, and failure detection (Network Event Notification Service). Moreover, two separate reliable message dissemination services for control messages and for application messages are used. This

[1] For instance, these applications may not want to block or perform roll back actions per change in the system as GCS or Paxos require.

enables us to remove reliability maintenance overhead from the critical path of delay-sensitive applications.

Within the ad hoc approach, the membership notifications serve only as *approximations* of the current group membership, without being synchronized with message stream. Continuous message reliability is guaranteed only among those group members that remain permanently alive and interconnected. This approach allows handling any number of simultaneous join/leave events concurrently, without waiting for the network or the group of members to stabilize. The traditional membership approaches require a stable period of time in order to consent on a new membership. As a result, they tend to limit the adoption of joining processes during periods of instability. The ad hoc approach provides stability of message flow against on-going membership changes, which is better fit for large groups and for wide area networks where the probability of unstable periods increases. The scalability of Xpand is accounted for by both its hierarchical architecture and the ad hoc membership approach.

The development of the ad hoc membership approach faces three challenges, interrelated to one another. (1) Architectural issue: which channel to prefer for which message: the more reliable "sequential" channel (a slow one), or the less reliable concurrent channel where a loss of some messages does not slow down the delivery of other messages. (2) Algorithmic issue: the dual channel architecture increases the asynchrony among various blocks that provide membership service and, as a result, produce conflicting race conditions that need to be controlled. (3) Design issue: how to design a system that takes advantage of the above, while still maintaining the ability to provide a stronger semantics for applications that require it.

The ad hoc approach best suits the requirements of collaborative networking applications, fault-tolerant applications that require weaker synchronization among a set of servers, one to many push applications (e.g., stock market monitoring) and the like. In Section 2.1 we present more examples of such applications.

One can try to obtain the properties of an ad hoc membership using other WAN toolkits, like combining Spread [4] and PBcast [6], though the challenge remains of providing the properties we look for in an efficient and robust way. Moreover, Spread and similar toolkits assume a predefined global set of servers, whereas Xpand [1] does not require an a priory knowledge of the potential set of group members.

The ad hoc membership is fully implemented within the Xpand middleware. Section 5 presents performance results of the implementation showing that membership changes have a negligible effect on the existing message flows.

2 Xpand Membership Service Model

Xpand membership algorithm is optimized to fulfill the following properties:

Smooth-Join: A new member joins the group without affecting the current message flow in the group;

Fast-Join: Once a joiner is registered in its LAN, it can start emitting messages to the group;

Dynamic Membership: Allows processes to dynamically join and leave the group in a WAN setting.

Xpand membership algorithm achieves these three properties without compromising the following basic message delivery services of Xpand:

FIFO order: Any receiver that starts receiving messages from a given source will get all emitted messages in FIFO order from the moment it joins the message flow;

Gap-Free: Two processes that remain connected during consecutive membership changes continue to receive each other's messages without any gap in the message stream.

The above-listed membership services combined with Xpand's reliable multicast service can be used as a dynamic and reliable point-to-multipoint service layer, on top of which stronger semantic services can be built. For instance, a virtual synchrony layer can be implemented as an extension of the ad hoc membership service Moshe [7]. Another example is Paxos, which can be implemented on top of our ad hoc service by having the leader communicating to existing majorities via the ad hoc services to carry out the three phases of Paxos [3].

2.1 Implementation Issues

In order to enable maintaining of large groups of clients, Xpand is implemented using a two-level hierarchical architecture. Each LAN is represented by a delegate that coordinates the protocol activities of this LAN within the WAN group.

In the current implementation, we assume that the number of LANs with processes that emit messages to the group is of the order of several dozens. The number of processes per LAN is assumed to be of the order of a couple of hundreds. These limitation is imposed due to the need for maintaining state information per sender and per receiver. We can further improve the implementation by enabling a client to join either as a receiver-only, a sender-only, or as a full member (i.e. both a sender and a receiver). Such an approach will enable to increase the number of processes, while keeping a reasonable amount of state information. To increase scalability, the architecture can be used for aggregation. A delegate can represent a collection of senders in its LAN, and the senders should not be explicit members of the WAN group. They will forward their messages to the delegate which will emit them to the WAN group.

We expect the above improvements to enable the ad hoc system to handle tens of thousands of clients.

The protocol is presented in this paper as a collection of state machines within various processes. The actual implementation of the protocol closely followed the state machines. This method enabled us to easily detect protocol and implementation bugs that appear in unforeseen transitions. We found out that by adding history (log) within every state machine we were able to drastically shorten the debugging time.

The ad hoc approach best suits the requirements of collaborative networking applications where the importance of service timeliness prevails over message reliability and consistency requirements are weaker than those of replicated database. For example, multimedia conferencing and distance learning applications benefit from the ad hoc GCS services in workspace sharing and parties coordination [8,9].

Another example is a loosely coupled set of servers. For example, in the fault-tolerant video-on-demand service by Anker et al. [10], a GCS based on the ad hoc membership service can be used for video server fault-tolerance and QoS negotiation[2]. The efficiency of handling membership changes by our protocol allows for smooth client hand-offs with smaller video-frame buffers at the client.

Yet another application is a case when multiple servers emit information to a loosely coupled set of clients. Each client wishes to receive the stream as soon as possible. For example, in stock market monitoring application messages are generated at several centers spread all over the world. The accessibility to the information is critical, but there is no justification for blocking the stream of messages during network changes. In an on-going audio conference, a newcomer can benefit from Xpand in the sense that Xpand will deliver a reliable multicast service from every session participant to the newcomer (and vice versa) right after the establishment of the corresponding new connections, without affecting previous established connections.

3 The Environment Model

We assume the message-passing environment to be asynchronous, processes communicate solely by exchanging messages and there is no bound on message delivery time. Processes fail by crashing and may later recover, or may voluntarily choose joining or leaving. Communication links may fail and recover.

Xpand exploits the following underlying services:

– A network event notification service (Subsection 3.1), through which Xpand learns about the status of processes and links;

– An integral reliable FIFO communication layer within the network event notification service. It is assumed that notification events and messages delivered via this service are causally consistent;
– A reliable point-to-point service for control messages of the protocol;
– A simple reliable point-to-multipoint service *in a LAN* for control messages of the protocol.[3]

The reliability of the above transport services means that messages sent via them are eventually received, or the recipient will eventually be suspected by

[2] The effect of not using a virtual synchrony in this VoD implementation is that the client's machine may receive, in transient situations, duplications of video frames.

[3] This mechanism is not a substitute for the general services of Xpand, but a simple mechanism focused on guaranteeing reliability over a small number of messages exchanged among protocol participants in a single LAN.

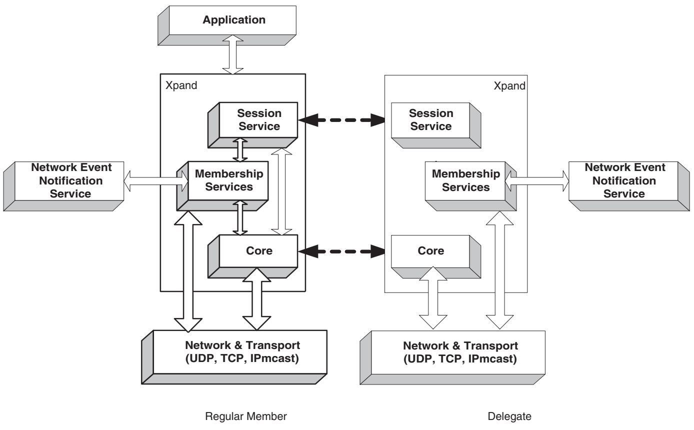

Application
Xpand
Session Service
Membership Services
Core
Network Event Notification Service
Network & Transport (UDP, TCP, IPmcast)
Xpand
Session Service
Membership Services
Core
Network Event Notification Service
Network & Transport (UDP, TCP, IPmcast)
Regular Member
Delegate

Fig. 1. The Layer Structure of the Protocol Participants.

the network notification service. For efficiency reasons, all the reliable transport services are used only for control messages (i.e. for protocol messages but not for user application messages).

3.1 Network Event Notification Service (NS)

Clients use the notification service to request joining or leaving the groups, or to get updated information regarding the group. The notification service accumulates and disseminates failure detection information along with information about these requests. The service is provided to clients by an interface that consists of the following basic functions:
NS_join(G) is a request by a client to make it a member of group G;
NS_leave(G) is a request to be removed from the membership of G;
NS_resolve(G) is a request to receive the current membership list (in a Resolve_Reply message/event) as it is reflected by the notification service;
NS_sendmsg(data, destination) is a request to reliably send a message (data) to a set of receivers (destination) via the notification service.

Clients of the notification service receive notifications via process events, that indicate the type of the event and the data associated with it. The event associated with the reception of a message via NS (a message that was sent using the *NS_sendmsg* function) is called *NS Recvmsg*.

The notification service contains a failure detector module. Since we assume an asynchronous environment, the notification service is bound to be unreliable in some runs [11], which means it may wrongly suspect correct processes. Since we deal with a service that can be implemented in an asynchronous environment, we do not require the notification service to be accurate. However, we assume that the notification service is always complete [11]. The overall liveness of Xpand does not depend on the notification service providing consistent sets, since it communicates with any connected set of clients. Congress [12] fulfills the

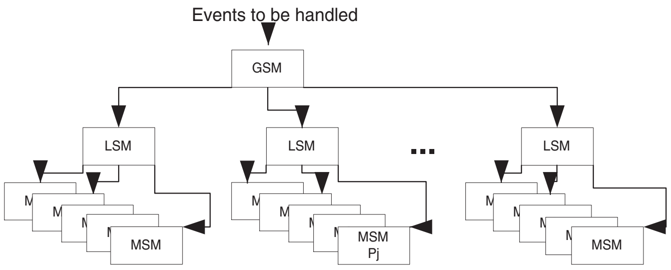


Fig. 2. State Machine Flow.

requirements of the notification service. Other group communication membership services (cf. Spread [4]) can also provide similar notification service.

3.2 Xpand's System Structure

The Xpand group communication system is fully described in [1]. Here we present only the essence of the system and the relevant assumptions. The processes participating in Xpand system are grouped into clusters so that all members of each cluster belonged to the same LAN. The clusters are spread over a WAN. We distinguish between two types of processes: *regular* and *delegate*. The regular processes run the user's application. In each LAN, the delegate is a designated daemon that serves as a representative of its LAN to all the other Xpand clusters.

Although a LAN delegate is replicated for fault-tolerance purposes, only a single delegate is active at each particular moment. This delegate is called an *active delegate*. All the other LAN delegates, called *backup delegates*, serve as warm backups of the active delegate. In the context of this paper, we refer only to a single delegate per LAN and ignore the issue of replacing a failed delegate.

For the sake of simplicity, the membership algorithm is described in this paper in the context of a single invocation of a process, which means that it may join and leave only once. In practice, an instance identifier per process distinguishes among different incarnations of the same process. The relationships between Xpand's components and its environment are shown in Figure 1.

4 Xpand's Ad Hoc Membership Algorithm

Xpand's membership algorithm deals with "views" of a group G. The view at p (meaning the view currently available to a specific process p's) is the current list of connected and alive members of G, as perceived by p. The maintenance of the view of G within p is based on an initial Resolve_Reply event received by p from NS. p updates its view by applying the information received in NS notifications, such as processes' joining or leaving.

Before describing the membership algorithm, we focus on the goals it aims to achieve. The algorithm should allow a new member p_i to join as fast as possible, while maintaining the following properties:

- p_i will start receiving messages from any active member (sender) p_j as soon as possible (given that the sender is alive and transmitting messages).
- For each sender p_j, a new member p_i will receive messages sent by p_j in the same order they were sent by p_i (i.e., FIFO order will be maintained).
- Once p_i receives a message from a specific member, which remains alive and within the same network partition as p_i, p_i will receive all the messages sent by that member from this message on without any gaps. The only situation in which gaps are allowed in the received message stream is when the source is declared disconnected by NS.

The first property will be obtained if a new joining member is notified of the sequence number of the following message from any sending member without any unjustified delay. In order to guarantee the other two properties, there is no need for a global synchronization on sequence numbers that are distributed to any new member with respect to a specific member p_j. This means, for example, that if any two new members p_1 and p_2 are joining simultaneously, a sending member p_j can announce a sequence number $SeqN$ to p_1 and a short time interval later announce $SeqN + k$ to p_2, in case it had managed to transmit k messages during this interval of time.

The asynchrony makes it impossible to describe the protocol in the traditional manner. The protocol can be viewed as an embedded set of state machines executed at the regular member and at the delegate. The top state machine is the *group state machine (GSM)*. That state machine invokes multiple *LAN state machine (LSM)*, one per LAN that contain members in the current view. The LAN state machine invokes *member state machines (MSM)*, one per each member that is listed in the view and is residing within that specific LAN. Figure 2 presents the general flow of control among the different state machines. The GSM receives events and forwards them to the appropriate LAN state machines from which the events are applied to the corresponding MSMs. The events are handled concurrently by the relevant state machines.

For brevity, we included the details of only some of the state machines into the paper. We have chosen to present the group state machine of a regular member and the LAN state machine of a delegate. For the sake of simplicity, we removed handling of various timeouts from our presentation of the state machines.

The membership algorithm uses the set of messages described in Figure 3. All the messages in the figure are messages sent by either delegates or regular members. These messages are sent via the transport services (Section 3).

The ad hoc membership algorithm handles the joining/leaving event, as well as network partitions and network merges. The state machines respond to external events received from NS, as well as to control messages sent by other members of the group (via the corresponding state machines).

The simplified pseudo code of a regular member joining is shown in Figure 4. The code covers the state machine[4] shown in Figure 6 and portions of other two state machines related to this specific case (see GSM-delegate and MSM-

[4] The related states and state transitions in Figure 6 are indicated.

Message Type	From	To	Message role
Force_Join	Regular member	Delegate	Causes Delegate to join a group
Start_Sync	Regular member	Delegate	Causes Delegate to reply with information about group members
Sync_Reply	Delegate	Regular member	Delegate's reply to the Sync_Reply (contains a list of members message sequence numbers as currently known to Delegate)
View_Id	Delegate	Delegate	Causes Receiver to reply with information about its **local** members
Cut_Request	Regular member	Delegate	Causes Delegate to reply with information about specific group members
Cut_Reply	Delegate	Delegate / Regular member	Delegate's reply to the View_Id / Cut_Request messages (contains a list of the requested members message sequence numbers as currently known to Delegate)

Fig. 3. Protocol messages.

delegate in [13])[5]. A regular joining member p_1 is notified about a new member p_2 in the group by receiving either a NS Join message or a Sync_Reply/Cut_Reply message (sent by its own delegate). In either case, p_1 initiates a new member state machine for p_2. In the former case, the MSM goes directly to "active" state, in which messages sent by p_2 will be processed. In the latter case, the NS notification regarding the joining of p_2 has not arrived yet. Thus, the MSM goes into "semi-active". In this state, p_1 waits for the proper NS Join message concerning p_2. When the NS Join message arrives, the MSM goes into "active" state. This complication is caused by the asynchrony resulted from the separation of the membership algorithm within Xpand and the external NS mechanism.

To limit some of the potential race conditions, the Start_Sync message is sent via NS. This ensures that by the time a delegate receives this message via NS, it has already received and processed all the previous NS messages[6], especially those received by the regular member at the moment of sending this request. This doesn't cover the multi-join case that is created when several members join at once, some of them not yet having delegates in the group. In a more complicated case, we may face a situation when members are in the group, but their remote delegates are not, or a situation when the Sync_Reply doesn't include all the necessary information about them. The regular LSM (in [13]) copes with those situations. LSM essentially needs to identify the case and to wait for its resolution, while allowing the sender to begin sending its information. While waiting for the delegate of a LAN to join the group, the member can

[5] E.g., the NS_Join in the code is actually part of the GSM of the delegate.

[6] By the causality assumption of the reliable FIFO comm. layer within the NS.

```
        Invoke NS_join(G) (piggyback next data message SeqN);
        label wait for RR:
(1)     wait for NS Resolve_Reply msg (RR);
(1->2)  if RR includes local delegate {
(1->2)       NS_sendmsg(Start_Sync of G, local delegate);
(2)          wait for a Sync_Reply message;
             allow the user application to send data
messages to G;
(2->3)       for each LAN listed in the Sync_Reply message
                 invoke the corresponding LSM;
(2->3)       for each new LSM
                 invoke the corresponding MSM;
(2->3)       for each new MSM {
                 extract sender message SeqN;
                 init sender data structure
             }
        }
        otherwise {
(1->5)       build Force_Join message m for G;
(1->5)       NS_sendmsg(m, local delegate);
(5)          wait for local delegate to join G;
(5->1)       issue a NS_resolve(G);
(1)          goto wait for RR
        }
```

Fig. 4. Highlights of Join Operation at Regular Member.

process further NS messages for LANs on which it has the full information[7]. Other NS messages need to be buffered and processed after the delegate joins and integrates within the group.

As regards the delegate part, a delegate joins a new group when its first local member joins it and "forces" the delegate to join also by sending a Force_Join message. Upon receiving such a message, the delegate invokes a GSM for the relevant group. Upon receiving the resolve reply, the GSM invokes a set of LSMs which, in turn, invokes a set of MSMs per LSM. The delegate GSM and MSM appear in [13]. Below we discuss the delegate LSM (LAN State Machine).

Figure 5 shows the simplified pseudo code of the delegate, executed upon receiving a Force_Join. The code covers the state machine[8] in Figure 7 and *portions* of the other two state machines related to this specific case (see MSM and GSM of delegate in [13])[9]. The Force_Join message causes the delegate to join a specific group. When the delegate joins the group and receives the resolve reply through NS, it needs to spawn LAN state machines per LAN that is represented in that resolve reply message. While the delegate is waiting for the resolve reply

[7] Full information here means acquiring message sequence numbers for each known member in the remote LAN.

[8] The related states and state transitions in Figure 7 are indicated.

[9] E.g., the NS_Join in the code is actually a part of the GSM of the delegate.

```
  label init-group:
        A Force_Join for group $G$ is received from a local member
        NS_join($G$);
        wait for NS Resolve_Reply msg ($RR$);

        split $RR$ into separate LAN lists;
        for each LAN list in $RR$ {
              invoke LAN state machine (Figure 7);
              if remote delegate is NOT listed in the $RR$
                 wait for remote delegate to join $G$;

  label peer sync:
(1->2)        NS_sendmsg(View_Id msg, remote delegate);
(2)           wait for Cut_Reply msg corresponding to View_Id msg;
(2->3)        for each member listed in the Cut_Reply msg {
                 invoke the corresponding MSM and within it:
                        extract member message SeqN;
                        init member data structure;
              };
(3)           put LSM into ``active'' state
        } /* for each LAN in the group... */
```

Fig. 5. Highlights of First Join Operation at Delegate.

message, it may receive View_Id messages or Start_Sync messages via NS. Those messages will be buffered and processed later when the resolve reply is received.

When the delegate is already a member of a group and is notified about a new member of its own LAN (via an NS Join notification), it takes the new member message SeqN that is listed in NS Join notification and initializes the new member data structure.

A different case occurs when a remote member joins the group while its own delegate is not a member yet. In this case, the local delegate waits for the remote delegate to join via the NS. Once the remote delegate has joined, the local delegate performs the protocol in Figure 5 starting from peer sync label.

In the above description, we have dealt only with a common case from the overall protocol. A careful use of the NS channel enables us to cope with some of consistency problems. The full protocol is a combination of all the state machines being executed concurrently while responding to the arriving events. Due to the lack of space, we are not considering all the particular issues. The description of the state machines here and in [13] enables to identify some of those issues.

5 Implementation and Performance Results

To test the efficiency of the ad hoc membership algorithm and its implementation, we conducted several tests to measure the effect of membership changes

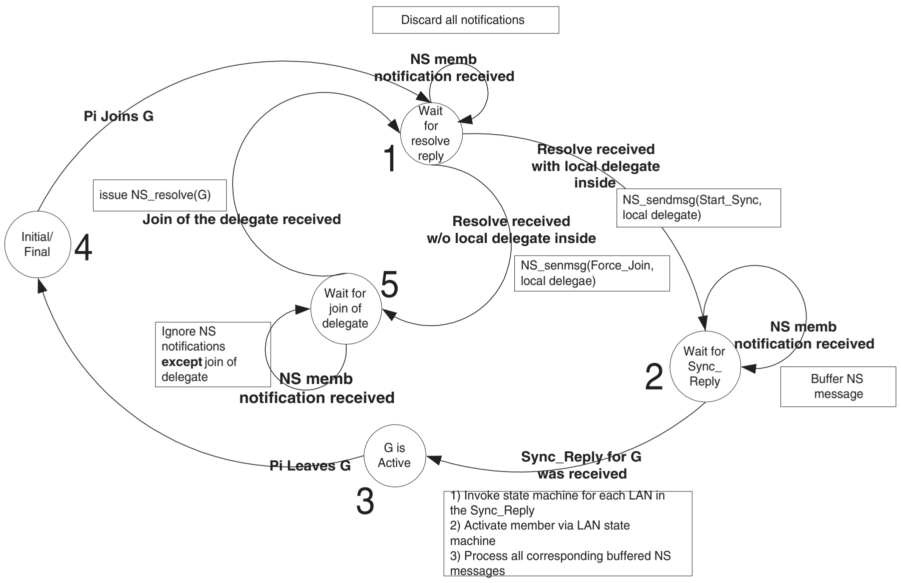

Discard all notifications
NS memb notification received
Wait for resolve reply
1
Pi Joins G
Resolve received with local delegate inside
NS_sendmsg(Start_Sync, local delegate)
issue NS_resolve(G)
Join of the delegate received
Resolve received w/o local delegate inside
NS_senmsg(Force_Join, local delegae)
Initial/ Final
4
Wait for join of delegate
5
Ignore NS notifications except join of delegate
NS memb notification received
Wait for Sync_ Reply
2
NS memb notification received
Buffer NS message
G is Active
3
Pi Leaves G
Sync_Reply for G was received
1) Invoke state machine for each LAN in the Sync_Reply
2) Activate member via LAN state machine
3) Process all corresponding buffered NS messages

Fig. 6. Group State Machine at Regular Member.

on ongoing message flow. Specifically, we will show its fast_join and smooth_join (Section 2) properties.

The ad hoc algorithm was tested in multi-continent (WAN) setting. The tests included three sites: The Hebrew University of Jerusalem in Israel (HUJI), the Massachusetts Institute of Technology (MIT) and the National Taiwan University (NTU). We had estimated the round trip times and the loss percentage between these sites, in order to have the characteristics of the network interconnecting them. The round trip times measured between MIT and HUJI and between MIT and NTU were both about 230ms. The round trip time between HUJI and NTU was about 390ms. The loss percentage was not as persistent as that of the round trip times, varying from 1 percent to 20 percent[10].

In all the three sites, the machines used were PC computers running the Linux operating system. As there is no IP multicast connectivity among these sites, a propriety application layer multicast (ALM) mechanism was used [1]. The obtained message distribution tree is presented in Figure 8(a). HUJI was selected to be the tree root by the algorithm used in the dynamic ALM tree construction. In all the tests the senders emitted a message every 100ms.

Fast_Join

In the first test, we measured the time it takes a new joiner to start receiving messages from active sources, the participants being 4 senders at HUJI (S1-S4) and one sender at NTU (S5). There were two joiners, one at MIT and one at

[10] In some cases, a higher loss percentage was observed when the interval between the ping (ICMP) messages was less then 100ms.

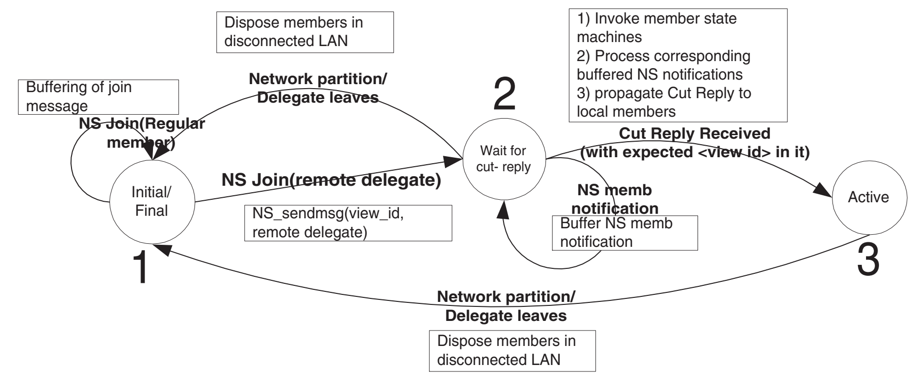
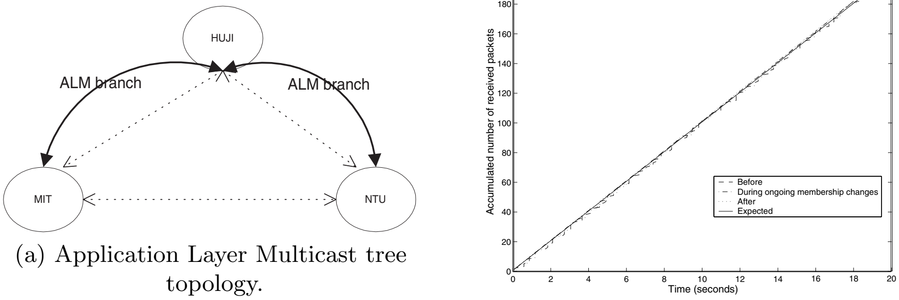

Dispose members in disconnected LAN
Network partition/ Delegate leaves
1) Invoke member state machines
2) Process corresponding buffered NS notifications
3) propagate Cut Reply to local members
Buffering of join message
NS Join(Regular member)
2
Wait for cut- reply
Cut Reply Received (with expected <view id> in it)
Initial/ Final
NS Join(remote delegate)
NS memb notification
Active
NS_sendmsg(view_id, remote delegate)
Buffer NS memb notification
1
3
Network partition/ Delegate leaves
Dispose members in disconnected LAN

Fig. 7. LAN State Machine at Delegate.



(a) Application Layer Multicast tree topology.

(b) Continuous test.

Fig. 8. The Basic Layout.

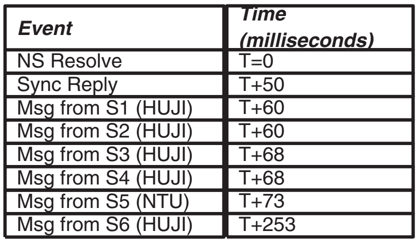
HUJI (S6). Both joined the group during the test and started sending messages right after joining the group.

The graph in Figure 9(a) shows that once the joiner at MIT (which recorded the events) receives the proper event notification (Sync_Reply message), it begins receiving messages from all current senders within a short time (23ms). It takes the joiner longer time to begin receiving from the another joiner. This extra time is a result of the difference between the arrival of the Resolve_Reply notifications between MIT and HUJI.

To evaluate the impact of our results, it should be mentioned that in Moshe [7], in a similar scenario, the minimal time should be at least one and a half round trip, which amounts to at least 600ms.

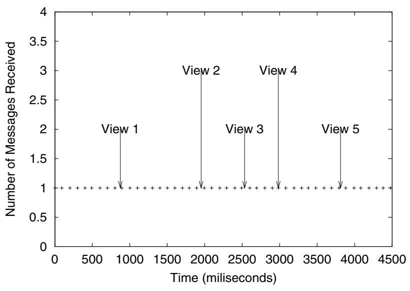
Smooth_Join

In this test, we measured the impact that a set of joiners might have on the ongoing message flow. The sender was at HUJI, the receiver at MIT, and during the test, 4 processes joined at HUJI and one at NTU. All the joiners left later on. The tests show that the impact on the message flow was negligible 9(b). Messages are still received every 100ms.



Event	Time (milliseconds)
NS Resolve	T=0
Sync Reply	T+50
Msg from S1 (HUJI)	T+60
Msg from S2 (HUJI)	T+60
Msg from S3 (HUJI)	T+68
Msg from S4 (HUJI)	T+68
Msg from S5 (NTU)	T+73
Msg from S6 (HUJI)	T+253

(a) First message receiving time.



(b) Simultaneous join impact.

Fig. 9. Impact of membership changes.

In the virtual synchrony implemented by Keidar et. al [7], the message flow needs to be stopped for at least one round-trip time (in the best possible scenario). In our specific setting it should have taken at least 390ms.

Continuous behavior

The third test measured the behavior of the ad hoc membership implementation over a long period of time, during which the loss rate was rather high. In this test, the sender was located at HUJI, the receiver at MIT, while the process at NTU and the processes at HUJI joined and left repeatedly. The tests show that the behavior of the system during the periods with no membership changes and the periods with such changes is almost identical 8(b), showing the efficiency of the algorithm.

Discussion

All the tests described above proved that the ad hoc approach enables processes to join the group promptly, with minimal impact on ongoing message flow. Applications that do not need strong characteristics will face the minimal impact we observed. Applications that require stronger semantics will still need to wait for full synchronization, as measured by Keidar et. al [7].

6 Related Work

Congress [12] and Maestro [14] were the first to observe that separating maintenance of membership from a group multicast will better enhance scalability of fault-tolerant distributed applications. This separation was later adopted by researchers who addressed the WAN environment ([7,15,16]). InterGroup [5] presents another WAN approach to address scalability.

Several research projects in the past sought to relax the semantics of the middleware for distributed applications ([4,6,14,16,17,18,19,20,21]). Our work takes advantage of both approaches, in order to find a better balance between efficiency, scalability and guaranteed semantics.

Recently, new implementations [22,23] of reliable multicast have appeared, whose protocols use peer-to-peer overlay systems. Those systems scale for large group, while members are not required to keep group membership information, which might be critical for some applications. An interesting implementation of application layer multicast is Narada project [24]. We are considering using the latter approach as an application layer in Xpand.

7 Conclusions

The focus of this paper is to address the needs of a class of distributed applications that require high bandwidth, reliability and scalability, whilebut not requiring the strong semantics of current distributed middleware solutions. Since current middleware cannot scale well when it is required to guarantee the strong semantics, there is a need to identify a better tradeoff between the semantics and the efficiency.

The ad hoc membership algorithm that we have developed, together with its implementation, present such a tradeoff. The performance results prove that our approach is feasible and can scale well.

The implementation shows that it is possible to integrate an external membership service with a hierarchical system for message distribution. We believe that other systems with hierarchical architecture and/or external membership service may apply similar techniques to their algorithms. A reliable multicast based on forward error correction is a simple one-to-many application. However, the rest of the applications listed in Section 2.1 need better reliability and coordination than our approach offers. Future research is intended to provide better characterization of these classes.

References

1. T. Anker, G. Chockler, I. Shnaiderman, and D. Dolev, “The Design and Performance of Xpand: A Group Communication System for Wide Area Networks,” Tech. Rep. 2001-56, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, August 2001, See also the previous version TR2000-31.
2. ACM, *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.
3. Leslie Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.
4. Y. Amir, C. Danilov, and J. Stanton, “A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication,” in *Proceedings of ICDSN'2000*, 2000.
5. K. Berket, Deborah A. Agarwal, P. M. Melliar-Smith, and Louise E. Moser, “Overview of the intergroup protocols,” in *International Conference on Computational Science (1)*, 2001, pp. 316–325.
6. Mark Hayden and Kenneth Birman, “Probabilistic Broadcast,” TR 96-1606, dept. of Computer Science, Cornell University, Jan 1996.
7. I. Keidar, J. Sussman, K. Marzullo, and D. Dolev, “A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs,” in *20th International Conference on Distributed Computing Systems (ICDCS)*, April 2000, pp. 356–365, Full version to appear in TOCS.

8. G. Chockler, N. Huleihel, I. Keidar, and D. Dolev, “Multimedia Multicast Transport Service for Groupware,” in *TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies*, September 1996.
9. I. Rhee, S. Cheung, P. Hutto, and V. Sunderam, “Group Communication Support for Distributed Multimedia and CSCW Systems,” in *17th Intl. Conference on Distributed Computing Systems*, May 1997.
10. Tal Anker, Danny Dolev, and Idit Keidar, “Fault Tolerant Video-On-Demand Services,” in *Proceedings of the 19th International Conference on Distributed Computing Systems, (ICDCS’99)*, June 1999.
11. Tushar Deepak Chandra and Sam Toueg, “Unreliable failure detectors for reliable distributed systems,” *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.
12. T. Anker, D. Breitgand, D. Dolev, and Z. Levy, “CONGRESS: Connection-oriented group-address resolution service,” in *Proceedings of SPIE on Broadband Networking Technologies*, November 2-3 1997.
13. T. Anker, I. Shnaiderman, and D. Dolev, “Ad Hoc Membership for Scalable Applications,” Tech. Rep. 2002-21, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, April 2002.
14. K. P. Birman, R. Friedman, M. Hayden, and I. Rhee, “Middleware support for distributed multimedia and collaborative computing,” in *Proceedings of the Multimedia Computing and Networking (MMCN’98)*, 1998.
15. T. Anker, G. Chockler, D. Dolev, and I. Keidar, “Scalable group membership services for novel applications,” in *Networks in Distributed Computing (DIMACS workshop)*, Marios Mavronicolas, Michael Merritt, and Nir Shavit, Eds. 1998, vol. 45 of *DIMACS*, pp. 23–42, American Mathematical Society.
16. Jeremy B. Sussman, Idit Keidar, and Keith Marzullo, “Optimistic virtual synchrony,” in *Symposium on Reliability in Distributed Software*, 2000, pp. 42–51.
17. D. Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing*, Springer-Verlag, 1991.
18. D. Dolev and D. Malki, “The Transis Approach to High Availability Cluster Communication,” *Communications of the ACM*, vol. 39, no. 4, April 1996.
19. R. van Renesse, T. M. Hickey, and K. P. Birman, “Design and Performance of Horus: A Lightweight Group Communications System,” TR 94-1442, dept. of Computer Science, Cornell University, August 1994.
20. R. Friedman and R. van Renesse, “Strong and weak virtual synchrony in Horus,” in *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems, (SRDS’96)*, October 1996.
21. Katherine Guo and Luis Rodrigues, “Dynamic Light-Weight Groups,” in *Proceedings of the 17th International Conference on Distributed Computing Systems, (ICDCS’97)*, May 1997.
22. S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, “Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination,” 2001.
23. Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel, “SCRIBE: The design of a large-scale event notification infrastructure,” in *Networked Group Communication*, 2001, pp. 30–43.
24. Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang, “A Case for End System Multicast,” *IEEE Journal on Selected Areas in Communication (JSAC)*, To appear.

Assignment-Based Partitioning in a Condition Monitoring System

Yongqiang Huang and Hector Garcia-Molina

Stanford University, Stanford, CA 94305
{yhuang, hector}@cs.stanford.edu

Abstract. A condition monitoring system tracks real-world variables and alerts users when a predefined condition becomes true, e.g., when enemy planes take off, or when suspicious terrorist activities and communication are detected. However, a monitoring server can easily get overwhelmed by rapid incoming data. To prevent this, we partition the condition being monitored and distribute the workload onto several independent servers. In this paper, we study the problem of how to make a partitioned system behave "equivalently" to a one-server system. We identify and formally define three desirable properties of a partitioned system, namely, orderedness, consistency, and completeness. We propose assignment-based partitioning as a solution that can handle opaque conditions and simplifies load balancing. We also look at a few typical partitioned systems, and discuss their merits using several metrics that we define. Finally, an algorithm is presented to reduce complex system configurations to simpler ones.

1 Introduction

A *condition monitoring system* is used to track the state of certain real-world variables and alert the users when pre-defined conditions about the variables are satisfied. For example, soldiers in a battlefield need to be notified when the location of enemy troops is within a certain range. Authorities must be alerted if suspicious money transfer transactions or communication messages are detected that fit into a "terrorist" pattern. The manager of a nuclear plant has to get a message on his/her Personal Data Assistant (PDA) whenever the temperature of the reactor is higher than a safety limit.

Figure 1(a) illustrates such a condition monitoring system. It consists of one or more *Data Monitors* (DM), a *Condition Evaluator* (CE), and one or more *Alert Presenters* (AP). A Data Monitor tracks the state of a real world variable, such as the reactor temperature. Periodically or whenever the variable changes, the DM sends out an *update*, i.e., a temperature reading. The stream of updates arrive at the Condition Evaluator, which uses them to evaluate a predefined user condition c, e.g., "reactor temperature is over 3000 degrees." If the condition c is satisfied, an *alert* is sent to the Alert Presenter, which is responsible for alerting the user. In this case, the user will be notified by a message on his/her PDA that the reactor has overheated. If the PDA is off or disconnected, the CE logs the alert, and sends it later, when the AP becomes available.

We call systems with a single Condition Evaluator, such as the one in Figure 1(a), *centralized* monitoring systems. One problem with a centralized system is that the CE can easily get overloaded [1]. High volume of updates may arrive at the CE at a rapid

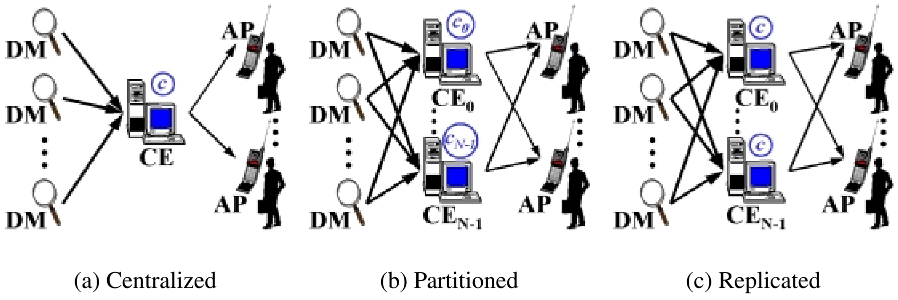


Fig. 1. Condition monitoring systems

pace. For each new update received, the CE needs to match it against c. The matching can become a time-consuming operation if, for example, the CE needs to extract enemy plane movement information from satellite images, or to analyze text messages in order to determine the topic. Furthermore, for each update that matches, an alert has to be generated, logged, and sent out to all interested APs. In a system with frequent new updates and many users, the resources required to match updates and to send alerts may well exceed the capability of a single machine.

The above problem can be alleviated by introducing multiple independent Condition Evaluators to share the workload (Figure 1(b)). In a *partitioned* monitoring system, each CE$_i$ monitors a modified "condition partition" c_i instead of c. The CEs independently make their own decisions about when to generate an alert. Ideally, the work at each CE$_i$ is reduced to a fraction of the centralized case, while the outcome is kept "equivalent."

One way of obtaining the c_i's is to break up c into sub-expressions. For example, assume c is "temperature is over 3000 degrees OR temperature has risen for more than 200 degrees since last reading." We can then split the disjunction to obtain c_0 ("temperature is over 3000 degrees") and c_1 ("temperature has risen for more than 200 degrees"). Such *structure-based partitioning* takes advantage of knowledge about c's internal structure, and the resulting condition partitions are usually subexpressions of c.

Structure-based partitioning often gives a natural and efficient way of breaking up c. However, this type of partitioning is only applicable when the internal structure of c is known and amenable to being broken apart. Moreover, it is often difficult to evenly balance the workload among the CEs because one condition partition may be much more costly to monitor than another. Instead, we propose a different approach, suitable for partitioning "opaque" conditions, whose expression c is treated like a black box.

In *assignment-based partitioning*, each CE$_i$ is ultimately responsible for the whole c, but only for some fraction of the updates. For example, two CEs can partition the workload so that one evaluates c on the even updates, while the other on the odd updates. More formally, each CE$_i$'s condition partition c_i takes on the form $c_i = p_i \wedge c$, where p_i is an "assignment test" on the updates. Intuitively, when a new update arrives at CE$_i$, the assignment test is performed first. If the test is passed, we say that the new update has been *assigned* to CE$_i$. In this case, processing of the update continues as in the case of a centralized system. On the other hand, if the assignment test fails, CE$_i$ does not even evaluate c (in other words, the conjunction in $p_i \wedge c$ is short-circuited). In this way, each CE$_i$ is only responsible for (i.e., evaluates and generates alert for) those updates assigned to it. Thus, load balancing is achieved by controlling the fraction of updates that are assigned to each CE$_i$.

To illustrate the benefit of partitioning, let us assume that processing a new update takes exactly one unit of time in a centralized system monitoring condition c. Such a system can handle a "Maximum Sustained Update Rate" (defined more formally in Section 5.1) of 1 new update per unit time. Further assume that CE$_i$ only takes $\frac{1}{N}$ time unit on average to process an update, either because c_i is a much simpler expression, or because most of the updates received are not assigned to CE$_i$ and can be disregarded quickly. Consequently, such a system can withstand a Maximum Sustained Update Rate of roughly N updates per time unit, resulting in an N-fold increase in capability.

Another benefit of a partitioned system is increased partial reliability of the monitored condition. Since the condition is monitored by several CEs together, even if one of them goes down, the user should still be able to receive some alerts, unlike in the centralized case. In a previous paper [2], we considered full replication as a solution for reliability. In a replicated monitoring system (Figure 1(c)), multiple CEs all monitor the same condition to guard against failures. However, the Maximum Sustained Update Rate of such a system is not improved compared to a centralized system, even though more CEs are deployed. A partitioned system is able to reap some of the benefits of a replicated system without its full cost.

This paper addresses the problem of partitioning a condition so that it can be handled by multiple CEs in parallel. In particular,

- We define a set of desirable properties that contribute to making a partitioned system "equivalent" to a centralized system (Section 3.1).
- We prove some fundamental properties of partitioned systems in general (Section 3), and assignment-based partitioning in particular (Section 4).
- We propose and compare two methods of doing assignment-based partitioning. We also present a few representative systems, and develop performance metrics to measure their relative merits (Section 5).
- Finally, we develop methods to apply our analysis to more complex system configurations involving multiple variables (Section 6).

2 Problem Specification

In this section, we give more details on the workings of a condition monitoring system, using the nuclear reactor temperature sensing example from Section 1. The Data Monitor is a temperature sensor attached to the reactor. It is also connected to a communications network which allows it to send temperature readings to other devices. We assume that each DM monitors only one variable, as a sensor which simultaneously monitors two targets can be thought of as two DMs co-located on the same device.

An update has the format $u(varname, data, seqno)$ where $varname$ is an identifier of the real world variable being tracked, and $data$ reports the new state of this variable. The $seqno$ field uniquely identifies this update in the stream of updates from the same variable. We assume that sequence numbers of updates sent from the same DM are consecutive. In our reactor example, an update $u(x, 3000, 7)$ denotes the seventh update sent by this DM for reactor x, reporting a temperature reading of 3000 degrees. In the remainder of this paper, we will use 7^x to denote such an update.

A condition c is a predicate defined on values of real world variables. The set of variables that appear in a condition's predicate expression is the *variable set* of that condition, denoted by V. The CE receives updates from all DMs that monitor variables in V. When a new update arrives, the CE re-evaluates its condition. The update is said to match (or trigger) c if the data contained in it causes c's predicate to evaluate to true. For example, condition $c1$ ("reactor temperature is over 3000 degrees") is triggered whenever the temperature reading exceeds 3000.

Note that to evaluate condition $c1$, only the current temperature reading is needed. However, to monitor another condition $c2$ ("reactor temperature has risen for more than 200 degrees since last reading"), the CE needs to remember the previous update in addition to the current one. Thus, we generalize to say that a condition is defined on a set (H) of "update histories," one for each variable in V. An *update history* for variable x, denoted H_x, is a sequence of D x-updates received by the CE. Specifically, $H_x = \langle H_x[0], H_x[-1], H_x[-2], \ldots, H_x[-(D-1)]\rangle$, where $H_x[-i]$ is the ith most recently received update of variable x. (See later for how to choose D). When a new x-update is received, it is first incorporated into H_x, which is then used to evaluate the condition. For instance, after update 7^x arrives, $H_x[0]$ becomes 7^x, and $H_x[-1]$ is 6^x, and so on.

The number D, called the degree of H_x, is determined by the condition. We say that a condition c is of degree D with respect to variable x if the evaluation of c needs an H_x of at least degree D. The degree of a condition is inherent in the nature of the condition itself, and it dictates how many x-updates the CE will need to store locally (i.e., the degree of H_x). Thus, condition $c1$ can be expressed more formally as $c1(H) = (H_x[0].data > 3000)$, and is of degree 1 to variable x. On the other hand, $c2(H) = (H_x[0].data - H_x[-1].data > 200)$, and is of degree 2.

When the condition evaluates to true, an alert is sent out by the CE. An Alert Presenter collects such alerts and displays them to the end user, e.g., by a pop-up window or an audible alarm. We assume that the alert is of the form $a(condname, histories)$, where $condname$ identifies the condition that was triggered, and $histories$ are all the update histories used by the CE in evaluating the condition. The histories are included if the AP needs them for a final round of processing on the alerts before presenting the alerts to the user, as will become clear later.

In a partitioned system, N CEs (CE$_0$ through CE$_{N-1}$) work in concert to handle a single condition c. Instead of c, each CE$_i$ monitors a modified *condition partition* c_i. When a new update arrives at CE$_i$, an alert is triggered if and only if $c_i(H)$ is true. Although the CEs monitor their respective condition partitions independently, the goal is to produce an "equivalent" overall effect to a single CE monitoring the original c.

Finally, we assume that the links connecting the CE to the DMs and APs provide ordered and lossless delivery. The focus of this paper is not on reliability, and the reader is referred to [2] for discussion of issues when the links assumption does not hold.

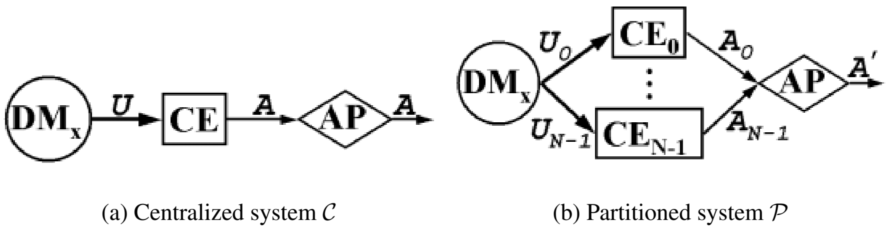

DM_x
U
CE
A
AP
A
(a) Centralized system $\mathcal{C}$
DM_x
U_0
CE_0
A_0
AP
A'
U_{N-1}
CE_{N-1}
A_{N-1}
(b) Partitioned system $\mathcal{P}$

Fig. 2. Analysis model for single variable systems

3 Single Variable Conditions

In this section, we present some general results of partitioned systems, regardless of whether the structure-based or assignment-based method is being used. For now, we restrict our discussions to conditions involving only one real world variable x, i.e., $V = \{x\}$. Hence, the monitoring system contains only one Data Monitor, and all relevant updates will have x as their $varname$. Single variable systems are important both as basis for more complex configurations, and in their own right. Many useful real-world scenarios, such as single-issue stock tracking and danger alerts, can be formulated as a single variable monitoring problem. In Section 6 we will investigate multiple DMs.

Figure 2(a) depicts a centralized condition monitoring system $\mathcal{C}$. Let U represent the sequence of updates sent out by the DM over some period of time. The Condition Evaluator receives the sequence of updates U as input, and generates alert sequence A as output, according to the definition of c, the condition it is monitoring.

Figure 2(b) illustrates a partitioned one-variable system $\mathcal{P}$ based on the centralized system $\mathcal{C}$. For now, we assume that all CEs in $\mathcal{P}$ receive the same input as the single CE in the corresponding centralized system $\mathcal{C}$, i.e., $U_0 = U_1 = \cdots = U_{N-1} = U$. In other words, the update stream is fully replicated to all CEs. Note that replicating the update stream N times will likely mean more work for the DM. If it is desired to keep DM itself simple (since it can be a dumb device such as a networked thermometer), an additional "update replicator" can be installed to receive updates from the DM and forward them to all CEs. Section 4 will also investigate scenarios where the full update replication assumption can be relaxed.

Since each CE$_i$ monitors a different c_i, their outputs will be different. Specifically, we use A_i to denote the sequence of alerts generated by CE$_i$. Finally, the AP collects all A_i's and merges them to produce a final alert sequence A', which are displayed to the end user. To produce A', the AP uses a simple duplicate elimination algorithm called ExactDuplicateRemoval to filter out some alerts. In particular, the AP discards an alert if it is "identical" to another alert that has already been presented to the user. Two alerts are considered identical if their history sets H are the same, i.e., if they triggered on the same set of updates.

3.1 System Properties

We propose three desirable properties of a partitioned system, defined by comparing the output of $\mathcal{P}$ to that of $\mathcal{C}$. We define an alert a's sequence number to be $H_x[0].seqno$, namely, the sequence number of the x-update that triggered a. A sequence A of such alerts is *ordered* if the sequence numbers of all alerts contained in A form a ordered sequence. Furthermore, we use ΦA to denote the (unordered) bag consisting of these sequence numbers.

A partitioned system $\mathcal{P}$ is said to have each of the following properties if every alert sequence A' it outputs satisfies the corresponding criterion.

1. **Orderedness:** A' is ordered.
2. **Consistency:** $\Phi A' \subseteq \Phi A$.
3. **Completeness:** $\Phi A' = \Phi A$.

The three properties measure how the behavior of $\mathcal{P}$ "conforms" to that of $\mathcal{C}$. Specifically, orderedness looks at the order in which alerts are presented to the user, while the other two criteria deal with what alerts are presented. Orderedness indicates that alerts are delivered to the user in increasing sequence number order. Since a centralized system $\mathcal{C}$ always delivers alerts in this order, a partitioned system $\mathcal{P}$ that is ordered behaves similarly in this respect.

If a partitioned system $\mathcal{P}$ is consistent, the user can expect to receive (although perhaps in a different order) a subset of those alerts that would have been generated by the corresponding centralized system $\mathcal{C}$. In contrast, an inconsistent system is able to generate "extraneous" alerts that one would not normally expect from $\mathcal{C}$. Therefore, it is easy for a user behind an inconsistent system to tell that partitioning is being used when he/she sees these "extraneous" alerts.

Completeness is a stricter criterion than consistency. For a partitioned system $\mathcal{P}$ to be complete, it will have to generate all alerts and only those alerts that would have been generated by $\mathcal{C}$. Trivially, completeness implies consistency, while the reverse is not true. An incomplete system implies that the user may miss some alerts as a result of partitioning the workload among several CEs.

3.2 Discussion

How a condition c is partitioned among the CEs (specifically, the definition of c_i's) largely determines the properties achieved by the resulting system $\mathcal{P}$. The following theorems discuss the details. Due to lack of space, we omit all proofs in this paper. Bear in mind that the discussion of this section pertains to single variable conditions, with ExactDuplicateRemoval filtering, and where every CE$_i$ sees the same input.

Theorem 1. *A partitioned system $\mathcal{P}$ is ordered if and only if $\nexists U$, such that $c_i(H) = c_j(H') = true$ and $c_j(H) = false$ for some $i, j \in [0..N-1], i \neq j$ and $H, H' \in U, H_x[0].seqno < H'_x[0].seqno$.*

We deduce from Theorem 1 that in most cases orderedness is not achieved in a partitioned system. This is because normally, when the output alert sequences of different CEs are merged together at the AP, it is possible that alerts will be delivered out of order due to the different interleaving order of these sequences. Exceptions do exist, but often involve an impractical "trivial" partitioning. An example of such a trivial partitioning is where $c_0 = c$ and $c_i \equiv \mathit{false}$ for all other c_i's.

If orderedness is desired, however, it can be enforced by having the AP perform additional filtering on top of ExactDuplicateRemoval. For example, the AP can remember the last alert it has delivered to the user, and discard new alerts that arrive out of order. However, orderedness is gained in this situation at the expense of throwing out some potentially useful alerts.

Theorem 2. *A partitioned system $\mathcal{P}$ is consistent iff* $c_0 \vee c_1 \vee \cdots \vee c_{N-1} \Longrightarrow c$.

Intuitively, if $c_0 \vee c_1 \vee \cdots \vee c_{N-1}$ logically implies c, then any update that triggers at some CE$_i$ in $\mathcal{P}$ will also trigger condition c in $\mathcal{C}$. Analogously, the following theorem states that $\mathcal{P}$ is complete if and only if $c_0 \vee c_1 \vee \cdots \vee c_{N-1}$ and c are logically equivalent.

Theorem 3. *A partitioned system $\mathcal{P}$ is complete iff* $c_0 \vee c_1 \vee \cdots \vee c_{N-1} \iff c$.

Example 1. Condition c is defined as "temperature is over 3000 degrees OR temperature has risen for more than 200 degrees", or $c(H) = (H_x[0].data > 3000 \vee H_x[0].data - H_x[-1].data > 200)$. Splitting the disjunction using structure-based partitioning, we obtain $c_0(H) = (H_x[0].data > 3000)$, and $c_1(H) = (H_x[0].data - H_x[-1].data > 200)$. Applying the theorems above, we can prove that $\mathcal{P}$ is complete, but unordered.

4 Assignment-Based Partitioning

In this section and the next, we focus on assignment-based partitioning. We have $c_i(H) = p_i(u) \wedge c(H)$, where p_i is the assignment test defined on the newly received update u. We assume that evaluating $p_i(u)$ is a fast operation, with negligible cost, compared to evaluating $c(H)$. This ensures that the goal of partitioning is achieved, namely, that the average workload at each CE$_i$ is reduced compared to the single CE in a centralized system, as long as CE$_i$ is assigned only a portion of the steam of updates in U.

4.1 Discussion

The following theorems, derived from those in Section 3.2, illustrate how the definition of p_i determines the properties of an assignment-based system $\mathcal{P}$. As such, the theorems apply to single variable conditions, with ExactDuplicateRemoval filtering, and where every CE$_i$ sees the same input.

Theorem 4. *An assignment-based partitioned system $\mathcal{P}$ is ordered if and only if $\nexists U$ such that $p_i(u) = p_j(u') = true$ and $p_j(u) = false$ for some $i, j \in [0..N-1], i \neq j$ and $u, u' \in U, u.seqno < u'.seqno$.*

Theorem 5. *An assignment-based partitioned system $\mathcal{P}$ is always consistent.*

Theorem 6. *An assignment-based partitioned system $\mathcal{P}$ is complete if and only if $\forall u$, $\mathbf{C}_{i=0}^{N-1}(p_i(u) = true) \geq 1$, where $\mathbf{C}_{i=0}^{N-1}(p_i(u) = true)$ counts the number of i's in $[0..N-1]$ that make $p_i(u)$ true.*

Theorem 6 says that $\mathcal{P}$ is complete as long as every update is assigned to at least one CE$_i$. We observe that duplication of assignment (i.e., one update assigned to more than one CEs) results in replication of the condition (i.e., the condition being monitored by multiple CEs simultaneously), which may actually be beneficial in terms of reliability. Although replication does not affect the system properties (because duplicate alerts are eventually removed by the AP), it has its own issues which are not studied here. Instead, we will focus on systems where each update is assigned to exactly one CE$_i$.

4.2 Types of Assignment

Assignment-based partitioned systems are further divided into two categories depending on how the assignment is determined. In *autonomously assigned* systems, the assignment is done by the CEs autonomously. The CEs agree on the definitions of p_i's before the system starts running. The DM, not knowing these definitions, simply replicates all updates to all CEs concerned.

A *centrally assigned* partitioned system, on the other hand, relies on a central control to decide on the assignment at run time. Before an update u is sent out, the DM inserts an $aceid$ tag, which contains the id of the CE that this update is assigned to ($0 \leq aceid < N$).[1] At the other end, CE$_i$ recognizes that an update has been assigned to it if the $aceid$ matches its own id. In effect, $p_i(u) = (i = u.aceid)$.

While Theorems 4 to 6 above apply to both types of assignment-based systems, the following lemmas are tailored specifically for a single variable partitioned system with central assignment. Such a system is not ordered except in the "trivial" case where all updates are assigned to one particular CE$_i$. It is also always complete because it assigns each update to exactly one CE.

Lemma 1. *A centrally assigned partitioned system $\mathcal{P}$ is ordered if and only if $\exists k \in [0..N-1]$ such that $\forall u, u.aceid \equiv k$.*

Lemma 2. *A centrally assigned partitioned system $\mathcal{P}$ is always complete.*

Central assignment can potentially avoid some shortcomings of an autonomously assigned system. For example, when a new CE joins or an existing CE leaves an autonomously assigned system, all other CEs need to be notified in order to redistribute workload to achieve balance. This is because the definition of p_i usually depends on N, the total number of CEs. Hence when N changes, all p_i's must be redefined. Analogously, when one CE goes down or gets overloaded temporarily, it is hard to dynamically adjust the work distribution to adapt to the change without inter-CE communication. With

[1] Note that this requires a more intelligent DM. As noted before, an "update replicator" can be used to keep the DM itself simple. Also, note that the use of a central control in this particular case does not create an additional single point of failure as the DM is needed even with autonomous assignment.

central assignment, on the other hand, the exact assignment of an update u is finalized just before it is sent out (when $aceid$ is tagged on). Thus, a centrally assigned system has much more flexibility in adapting to a dynamic environment.

4.3 Dropping Updates

Another problem with autonomous assignment is that the update stream U must be fully replicated N times ($U_0 = U_1 = \cdots = U_{N-1} = U$). Because the DM does not know about the definition of p_i, it has no way of predicting which CE or CEs will need a particular update. Hence it must send any update to all CEs to make the system work. Without an efficient multicast protocol, this duplication of update sends can be quite wasteful.

In contrast, as an optimization of a centrally assigned system, the DM can potentially drop some updates to certain CEs in order to avoid unnecessarily sending them. Since the DM controls update assignment, it is also in a position to predict whether a particular update u needs to be sent to a particular CE$_i$ in order for the system to function. In essence, we can reduce U_i, the input to CE$_i$, to a particular subsequence of U, instead of the full U, while keeping the system outcome the same.

One might think that only assigned updates need actually be sent to CE$_i$ (in other words, $U_i = \langle u \mid u \in U$ AND $u.aceid = i\rangle$). However, as the following example shows, this naive approach does not work. In fact, the minimal U_i also depends on D, the degree of the condition being monitored.

Example 2. Assume a condition of degree 2: "reactor temperature has risen for more than 200 degrees since last reading". Also assume that update 7^x is assigned to CE$_1$, but 6^x is not. If $U_i = \langle u \mid u \in U$ AND $u.aceid = i\rangle$, then $7^x \in U_1$, but $6^x \notin U_1$. However, in order for CE$_1$ to correctly evaluate the condition when it receives 7^x, it will also need the data from 6^x. Therefore, even though 6^x is not assigned to CE$_1$, it must still be sent to it.

The following lemma shows precisely when an update can be dropped. In short, u is sent to CE$_i$ if it is assigned to CE$_i$, or if a later update (u_k) will be assigned to CE$_i$ and the condition evaluation of u_k depends on u.

Lemma 3. *The output of a centrally assigned partitioned system remains the same if U_i is changed from U to $\langle u \mid u \in U$ AND $\exists u_k \in U$ such that $u_k.aceid = i$ and $0 \leq u_k.seqno - u.seqno < D\rangle$.*

5 Sample Assignment-Based Systems

In this section we show a few sample ways of constructing an assignment-based partitioned system $\mathcal{P}$. For each system, both an autonomously assigned version and a centrally assigned version exist, and they behave identically (except that the latter can drop certain updates as an optimization).

Table 1 summarizes the major results which we explain next. If the definition of a system is given as $\mathbf{d}$ in the table, then the autonomously assigned version is obtained by $p_i(u) = (i = \mathbf{d})$, while the centrally assigned version is defined as $p_i(u) = (i = u.aceid)$ and $u.aceid = \mathbf{d}$.

Table 1. Comparison of assignment-based systems. $\Re_i$ is the i-th element of an infinite random sequence of integers, with each element between 0 and $N-1$, inclusive

Name	Definition	Comp.	Ord.	MSUR	AWT	PUD
Trivial	0	$\surd$	$\surd$	1	0	$1-\frac{1}{N}$
Random	$\Re_{u.seqno}$	$\surd$	X	$N-\epsilon$	$2(\frac{N}{\epsilon}-1)$	$(1-\frac{1}{N})^D$
RoundRobin	$u.seqno \bmod N$	$\surd$	X	N	0	$\max(0, 1-\frac{D}{N})$
q-RndRbn	$\lfloor \frac{u.seqno}{q} \rfloor \bmod N$	$\surd$	X	N	$\frac{(N-1)(q-1)}{2}$	$\max(0, 1-\frac{q+D-1}{qN})$

5.1 Performance Metrics

In order to quantitatively compare different systems, we develop three performance metrics to measure aspects of a system such as its throughput and mean response time. We use a relatively simple analysis model to capture the important system tradeoffs. Our model assumes that updates in U are generated at a constant rate of α. For example, if $\alpha = 2$, then a new update appears every half time unit. When a new update arrives at a CE, the quick assignment test is performed first. If assigned, the update is processed by this CE. Processing time is assumed to take 1 time unit exactly. However, if the CE is currently busy processing another previous update, the new update is appended to an update queue at this CE.

Our first metric measures the throughput of the overall system. The *Maximum Sustained Update Rate* (MSUR) of a partitioned system is the maximum α at which the system is stable, i.e., the update queues at all CEs reach a steady state. As a reference, a centralized system $\mathcal{C}$ has an MSUR of 1 (update/time unit). At update rates greater than 1, updates are generated at a higher pace than they can be consumed by the CE, and the queue length increases without bound.

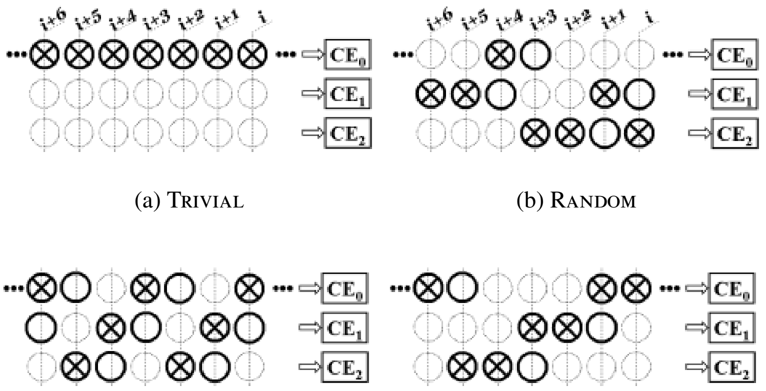
The *Average Wait Time* (AWT) is the average time an update has to spend waiting in the update queue, while the system is running at MSUR. AWT measures the average response time of the system. In the centralized example, the AWT is 0 (time unit) because a new update arrives just when the previous one finishes processing.

Finally, the *Percentage of Updates Dropped* (PUD) metric gives the average percentage of updates dropped in the centrally assigned version of a partitioned system. For example, if the centrally assigned version only needs to send each update to half of the N CEs on average, then the system has a PUD of 50%. The larger the PUD number, the less work the central control does. If PUD $= 0$, the centrally assigned solution does not save any effort over autonomous assignment.

5.2 Comparison

Due to space limitations, we will only present here a brief comparison of systems, and omit details such as derivations of various results summarized in Table 1. Our first sample system, named Trivial (Figure 3(a)), is a "trivial" system because it assigns all the work to one particular CE, CE$_0$. In the second system Random (Figure 3(b)), an update is assigned to a random CE every time. Note that to implement the autonomously assigned version of Random, the CEs simply need to agree on a pseudo-random algorithm and a seed at the beginning in order to avoid communication during run time. RoundRobin

(Figure 3(c)) assigns each update to the next CE in turn. Finally, q-RoundRobin is a variation on regular RoundRobin where each CE gets assigned q consecutive updates in U before next CE's turn. Naturally, RoundRobin is simply a special case of q-RoundRobin with $q = 1$.[2]



(a) Trivial (b) Random

(c) RoundRobin (d) q-RoundRobin

Fig. 3. Running illustrations of sample partitioned systems, with $N = 3$, $D = 2$, and $q = 2$. A crossed circle ($\bigotimes$) means that an update is assigned to a particular CE. A circle ($\bigcirc$) implies that the update is not assigned but still must be delivered to this CE. A dotted circle can be dropped by a centrally assigned system

Figure 4 plots PUD against N, the total number of CEs, based on the formulas in Table 1. The various curves represent several different sample systems. The figure shows that in general PUD rises with more CEs, implying that the advantage of central assignment in dropping updates is more significant. Incidentally, Trivial has the best PUD among all partitioned systems because the DM in a centrally assigned Trivial only needs to send updates to one CE, CE_0. However, since one CE handles all the real work, the system obviously does not benefit from being a partitioned system. As such, Trivial has the same MSUR (1) as a centralized system $\mathcal{C}$.

Since Random distributes the work more evenly among the N CEs, it is able to significantly improve its throughput compared to Trivial. Its MSUR is $N - \epsilon$, where ϵ is an arbitrarily small positive number.[3] Note that its MSUR can get infinitesimally close to N, but not equal to N, due to the randomness in how often each CE gets assigned. Moreover, as MSUR approaches N, the AWT suffers as a result.

Because of its regular assignment pattern, RoundRobin further improves on Random with an MSUR of N and no wait time for the updates (AWT $= 0$). On the other hand, as Figure 4 shows, RoundRobin's PUD curve falls below that of Random, espe-

[2] In fact, Trivial can also be considered as a special case of q-RoundRobin with $q \to \infty$.
[3] The calculation assumes Poisson random arrival at each CE_i (M/D/1 queue), which is a close approximation especially for small time units.

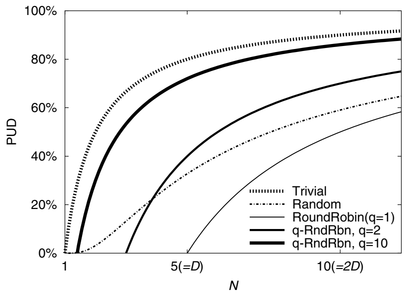


Fig. 4. Comparison of PUD performances

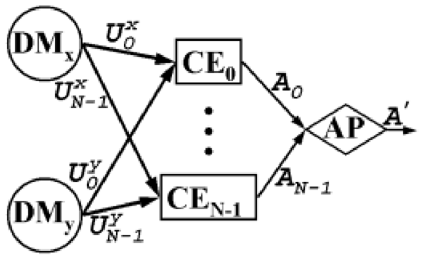


Fig. 5. A partitioned system monitoring a condition with two variables: x and y

cially for small N's. The figure also points out that there is a threshold to N ($N = D$) below which no updates can be dropped by RoundRobin.

Finally, q-RoundRobin is designed to improve RoundRobin's PUD performance, while preserving its MSUR at N. As shown in Figure 4, a bigger q enhances the PUD by both reducing the N threshold and increasing the rate at which the curve rises with N. For instance, at $N = 2D$, an average update can be dropped to half (50%) of the CEs if $q = 1$, but almost 70% if $q = 2$.

The tradeoff of a bigger q, on the other hand, is that the stream of updates assigned to a particular CE$_i$ will be more "bursty", i.e., a bigger chunk of consecutive update assignment, followed by a longer inactivity period. As a result, updates have to wait longer in the queue on average. For example, with 10 CEs, regular RoundRobin (i.e., $q = 1$) has no delay in processing updates, whereas an update has to wait on average 18 time units when $q = 5$. Hence, q represents a tradeoff between savings in dropped updates and faster system response time.

6 Multi-variable Conditions

So far we have dealt with conditions involving only a single variable. In this section we study conditions whose variable set contains more than one variable, i.e., $|V| > 1$. To illustrate, Figure 5 shows a system with two independent data sources, x and y. An example of such a condition is "temperature difference between two reactors x and y exceeds 100 degrees."

Some of the results in previous sections can be naturally extended to the multi-variable case. For example, we can define orderedness, consistency and completeness for multi-variable systems by extending the definitions of Section 3.1 in a straightforward manner. Similarly, performance metrics for multi-variable systems can also be defined. The detailed definitions are omitted here to avoid redundancy. However, additional complications do arise in a multi-variable system, which we explore next.

6.1 Discussion

As in the single variable case, it can be shown that orderedness is seldom achieved in multi-variable systems except for some trivial configurations. Therefore, we will focus only on consistency and completeness in the following discussion. For now, we assume that no updates are dropped (in Figure 5, $U_0^x = U_1^x = \cdots = U_{N-1}^x$, and likewise for y). We will explore the interesting issue of dropping updates later in this section.

From Figure 5, we observe that the input to CE$_i$, U_i, is a mixed sequence of x- and y-updates, obtained from interleaving U_i^x and U_i^y. Without any safeguard mechanism, it is possible for U_i^x and U_i^y to interleave differently at different CEs, resulting in the CEs seeing a different input from each other. Consequently, a multi-variable partitioned system is no longer consistent even if the condition in Theorem 2 is satisfied. The following example illustrates.

Example 3. Assume the condition c being monitored is "temperature difference between two reactors x and y exceeds 100 degrees" ($c(H) = (|H_x[0].data - H_y[0].data| > 100)$). We split c into $c_0(H) = (H_x[0].data - H_y[0].data > 100)$, and $c_1(H) = (H_y[0].data - H_x[0].data > 100)$. Notice that $c_0 \vee c_1 \implies c$.

Let $U_0^x = U_1^x = \langle 1^x(1000^\circ), 2^x(1200^\circ)\rangle$, and $U_0^y = U_1^y = \langle 1^y(1000^\circ), 2^y(1200^\circ)\rangle$. That is, both reactors start at 1000 degrees, and then both increase to 1200 degrees. Further assume that streams U_0^x and U_0^y are interleaved at CE$_0$ such that $U_0 = \langle 1^x, 1^y, 2^x, 2^y\rangle$. However, a different interleaving at CE$_1$ makes $U_1 = \langle 1^x, 1^y, 2^y, 2^x\rangle$.

In this case, CE$_0$ triggers when it receives 2^x, and CE$_1$ will also trigger when it receives 2^y. The user will receive both alerts (both will pass through the AP's filtering system since they are not considered duplicates). However, one can prove that no centralized monitoring system could generate such an alert combination. Hence the partitioned system violates consistency.

There are two general approaches to remedy the above consistency problem in multi-variable systems. First, measures can be taken to ensure that each CE sees the same input. For example, a centralized "update replicator" can receive updates from all DMs involved in a condition and then forward them to the CEs checking that condition, as hinted in earlier sections. As another example, a physical clock system as defined in [3] can be put in place to enforce a total ordering of x- and y- updates at all CEs.

In the second approach, the CEs are allowed to see different input. However, the Alert Presenter is required to take on a more active role in filtering out alerts that can potentially lead to inconsistency, as illustrated in the next example. The disadvantage is that the AP may potentially discard useful alerts.

Example 4. We assume the same setup as in Example 3. Furthermore, each alert sent to the AP is tagged with the updates that it triggered on. For example, the alert generated by CE$_0$ will be tagged with $\{2^x, 1^y\}$. The AP records this information before passing the alert on to the user. When the second alert arrives from CE$_1$ tagged with $\{1^x, 2^y\}$, the AP will be able to detect inconsistency (algorithm detail omitted due to space constraint) and thereby discard the new alert.

In the rest of our discussion, we assume that all CEs see the same input. In fact, the following theorem tells us that, with such an assumption, all earlier single-variable results on consistency and completeness become valid for multiple variables as well.

Theorem 7. *Given $U_0 = U_1 = \cdots = U_{N-1}$, Theorems 2, 3, 5 and 6, and Lemma 2 apply to multi-variable partitioned systems.*

```
1. Partition (arbitrarily) the set of $N$ CEs into $|V|$ disjoint and non-empty groups, one $G_v$ for
   each variable $v \in V$.
2. For each $G_v$, pick any single variable partitioning scheme (such as those presented in Section 5),
   which will be used to assign $v$-updates to CEs in $G_v$.
3. For each update $u$,
   3-1. Assign $u$ to one of the CEs in $G_{u.varname}$, according to the group's chosen single variable
        scheme. We thus have $\text{CE}_{u.aceid} \in G_{u.varname}$.
   3-2. Then, send $u$ to all CEs in all groups $G_v$ where $v \neq u.varname$, and in addition, to the
        subset of CEs in $G_{u.varname}$ as dictated by the single variable scheme.
```

Fig. 6. Algorithm MVP. For simplicity we assume that $N \geq |V|$

6.2 Dropping Updates

Another interesting consequence of multiple variables is that the DMs in a centrally assigned system can no longer drop updates as freely. Previously in the single variable case, the DM for variable x controls the assignment of x-updates, and it can accurately predict whether a particular x-update u will not be needed by CE$_i$ in its condition matching. With multiple independent DMs, however, such a prediction can no longer be safely made in general because, even when DM$_x$ considers u unnecessary for CE$_i$, u might still be needed in evaluating the condition when a later y-update is sent by DM$_y$.

Example 5. Given a two-variable centrally assigned system, assume that a certain x-update, 6^x, is not assigned to a particular CE$_i$. Further assume that the condition is of degree 1 to variable x, i.e., only the most recent x value is needed in condition evaluation. Thus, it would appear that 6^x can be safely dropped by DM$_x$ to this CE.

The problem comes, however, when a y-update, say, 3^y, arrives next. Assume 3^y is assigned to this CE. To evaluate the condition, the CE needs the most recent value of x, which should have been in 6^x. However, since the CE never received 6^x, value from an earlier 5^x is used instead. Thus the output of this CE is potentially affected by the dropping of 6^x. In fact, the resulting system is no longer consistent.

In the most general case it is difficult for the DMs to safely drop an update without communication between themselves. However, specific circumstances exist where it is possible to do so, and the following lemma presents one such scenario. In essence, Lemma 4 says that an x-update can be dropped to CE$_i$ if no updates of any variable other than x are assigned to CE$_i$, and if the update will not be used to evaluate conditions triggered by assigned x-updates.

Lemma 4. *An update u can be dropped to CE$_i$ if $\nexists u_k \in U_i$ such that $u_k.aceid = i$ AND ($u_k.varname \neq u.varname$ or $0 \leq u_k.seqno - u.seqno < D$).*

Based on the above lemma, we have developed an algorithm (Figure 6) to perform the partitioning in a centrally assigned multi-variable system. Algorithm Multi-Variable-Partitioning (MVP) associates a subset of CEs to each variable in the condition. It then reduces an overall multi-variable problem to single variable partitioning problems within each variable subset. To illustrate how the algorithm works, we walk through a simple example next.

Example 6. Assume a system with four CEs and two variables. For simplicity, further assume that the condition is of degree 1 to both x and y. Using Algorithm MVP, let $G_x = \{\mathrm{CE}_0, \mathrm{CE}_1\}$ and $G_y = \{\mathrm{CE}_2, \mathrm{CE}_3\}$. DM_x decides to use RoundRobin assignment within its group, while DM_y uses Random.

Based on RoundRobin, all x-updates with odd sequence numbers are assigned to CE_1, while even ones are assigned to CE_0. In other words, $\forall u$ with $u.varname = x$, we let $u.aceid = u.seqno \bmod |G_x| = u.seqno \bmod 2$. Furthermore, odd sequence numbered updates are sent to CE_1, its assigned CE, as well as to both CEs in G_y. Analogously for even sequence numbered x-updates. Similarly, a y-update is assigned randomly to either CE_2 or CE_3 according to Random. Then it is sent to its assigned CE in G_y, plus to CE_0 and CE_1.

Intuitively, an even numbered x-update u can be safely dropped to CE_1 because it is not needed to evaluate the condition when CE_1 receives an assigned odd numbered x-update. Furthermore, since CE_1 is never assigned any y-updates, it does not have any other evaluations which could have required u. Finally, CE_1 is guaranteed to have the correct y-value in its evaluations since it still receives all y-updates. Because each update is sent to exactly three out of four CEs, the system has an overall PUD of 25%.

As Theorem 8 shows, Algorithm MVP results in systems that are guaranteed to be complete, while still allowing updates to be systematically dropped. We further observe that the performance of the overall system is closely tied to that of the single variable schemes selected in Step 2, and a detailed analysis is omitted due to space limitations.

Theorem 8. *A centrally assigned multi-variable partitioned system produced by Algorithm MVP is complete as long as each single variable scheme selected in Step 2 generates complete systems.*

7 Related Work

Modern content-based publish/subscribe systems [4,5,6] route and filter events from their sources to interested destinations using a network of servers. There is certain overlap of functionality between these systems and the condition monitoring systems we are interested in. However, the focus of this work is on partitioning a condition to be handled by multiple servers while maintaining the same semantics.

SIFT [1] implemented an earlier monitoring system that provided batched filtering of newsgroup articles. Based on its experience in operation, SIFT pointed out and motivated the need for workload distribution. It also proposed the SIFT Grid, which is a distributed mechanism resembling a variation of our centrally assigned scheme. However, the SIFT Grid only dealt with degree 1 conditions, and properties about the partitioning were never analyzed formally.

Large scale World Wide Web servers often use dynamic load balancing and failover for increased capacity and availability [7]. Although that research mostly focuses on dynamic fault detection and traffic redirection for stateless servers, some of the techniques can very well be adapted to augment our centrally assigned monitoring systems.

Data stream systems are a recent research topic that has generated lots of interest and activities [8]. Although such systems also deal with streams of updates, the direction taken (e.g., data models for continuous queries) is quite different from ours. However, some of our techniques and analysis can very well be applicable in that context.

8 Conclusion

As real-time monitoring of sensors and information sources becomes more widespread, it will be critical to deal efficiently with large volume of updates and complex conditions. Condition partitioning allows multiple servers (CEs) to share the load, but can potentially lead to undesirable outcomes. In this paper we have studied what is needed to preserve orderedness, consistency and completeness in partitioned systems. With assignment-based partitioning, we have shown that CEs have a more balanced load because they handle the same condition over different updates. The metrics and analysis we have presented make it possible to compare assignment-based systems. In particular, our model suggests that RoundRobin performs well in throughput and mean response time, while a centrally assigned q-RoundRobin can trade system response time for less work.

References

1. Yan, T.W., Garcia-Molina, H.: The SIFT information dissemination system. ACM Transactions on Database Systems **24.4** (1999) 529–565
2. Huang, Y., Garcia-Molina, H.: Replicated condition monitoring. In: Proceedings of the 20th ACM Symposium on Principles of Distributed Computing. (2001) 229–237
3. Lamport, L.: Time, clocks, and the ordering or events in a distributed system. Communications of the ACM **21.7** (1978) 558–565
4. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing. (1999) 53–61
5. Zhao, Y., Strom, R.: Exploiting event stream interpretation in publish-subscribe systems. In: Proc. of the 20th ACM Symposium on Principles of Distributed Computing. (2001) 219–228
6. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Achieving scalability and expressiveness in an Internet-scale event notification service. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing. (2000) 219–27
7. Bourke, T.: Server Load Balancing. O'Reilly (2001)
8. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proc. of the 2002 ACM Symposium on Principles of Database Systems. (2002)

Tight Bounds for Shared Memory Systems Accessed by Byzantine Processes
(Extended Abstract)

Michael Merritt[1], Omer Reingold[1], Gadi Taubenfeld[1,2], and Rebecca N. Wright[3]

[1] AT&T Labs, 180 Park Ave., Florham Park, NJ 07932 USA
[2] The Open University and the Interdisciplinary Center, Israel
[3] Computer Science Dept., Stevens Institute of Technology, Hoboken, NJ 07030 USA

Abstract. We provide efficient constructions and tight bounds for shared memory systems accessed by n processes, up to t of which may exhibit Byzantine faults, in a model previously explored by Malkhi et al. [MMRT00]. We show that sticky bits are universal in the Byzantine failure model for $n \geq 3t + 1$, an improvement over the previous result requiring $n \geq (2t+1)(t+1)$. Our result follows from a new strong consensus construction that uses sticky bits and tolerates t Byzantine failures among n processes for any $n \geq 3t + 1$, the best possible bound on n for strong consensus. We also present tight bounds on the efficiency of implementations of strong consensus objects from sticky bits and similar primitive objects.

1 Introduction

Although Byzantine fault tolerance in message-passing systems has been extensively investigated, it was only recently that Malkhi et al. initiated the study of Byzantine fault tolerance in asynchronous shared memory systems [MMRT00]. Their work establishes a formal model and shows how the use of access control lists (ACLs) can constrain Byzantine behavior and permit reliable distributed computation. They investigate *universal* objects, which can be used to implement any shared object. Specifically, they show that sticky bits, a simple shared memory primitive long known to be universal in the crash failure model [Plo89], are also universal in the Byzantine failure model, provided that $n \geq (2t+1)(t+1)$, where n is the number of processes and t bounds the number of processes that may fail (exhibiting unconstrained, or Byzantine, behavior). One of the main results of this paper is to strengthen their result, showing that sticky bits are universal in the Byzantine failure model for any $n \geq 3t + 1$.

The universality results of [MMRT00] first use constructions from sticky bits to build strong consensus objects, then use strong consensus objects in an explicit universal construction of an arbitrary shared object. (Definitions of sticky bits, weak and strong consensus, and other objects mentioned in this introduction are provided in Section 2.) The bound $n \geq (2t + 1)(t + 1)$ follows from the construction in the first step, building strong consensus from sticky bits. In this paper, we present a novel construction of strong consensus from sticky bits for any $n \geq 3t + 1$. The consequence for universality is immediate: "Constructions of strong consensus from sticky bits for larger values of t would imply a more resilient universality result." [MMRT00]. Malkhi et al. demonstrate that strong consensus objects can only exist if $n \geq 3t + 1$, so our result is the best possible unless a different universal construction is used. Beyond strengthening the universality result for sticky bits, we present tight bounds on the efficiency of implementations of strong consensus objects from sticky bits and similar "powerful" shared objects. (Powerful objects include sticky bits that can be set by more than one process, but exclude registers and single-writer sticky bits. "Powerful" operations are those other than wait-free reads and writes.)

In Section 2, we review the model and definitions. Section 3 presents a general protocol schema that can be used to implement strong consensus from sticky bits. Instantiated separately for $n = 3t + 1$ and $n = (t + 1)^2$, the schema results in two strong consensus algorithms, which use $\binom{2t+1}{t}$ and $t+1$ powerful sticky bits, correspondingly. The contrast in efficiency of these constructions is striking: exponentially many powerful objects for $n = 3t + 1$ and just $t + 1$ for $n = (t + 1)^2$. We then modify the latter protocol to show that t powerful sticky bits are sufficient, provided $n \geq t^2 + 5t + 1$. Surprisingly, this algorithm works despite the fact that all of the t powerful shared objects could be written by t Byzantine processes.

In Section 4, we demonstrate bounds and tradeoffs among n, t, and the number and type of objects used. We show that the protocols of Section 3 are essentially optimal in the number of powerful shared objects used. In Section 4.1, we present a general lower bound that constrains the types of powerful shared objects and associated access control lists required to implement even weak consensus. Sections 4.2 and 4.3 investigate the tradeoff on the number of powerful shared objects that are necessary and sufficient for constructing strong consensus objects as n increases relative to t. The sufficiency results are constructive, providing explicit instantiations of the protocol schema of Section 3.

To summarize the main contributions of this paper, we present:

– A t-resilient strong consensus protocol (and therefore a universality result) from sticky bits, for $n \geq 3t + 1$.
– A t-resilient strong consensus protocol using only t powerful shared objects. (Surprising since t Byzantine processes can access all the powerful shared objects.)
– A proof that at least t shared objects (such as sticky bits) must be used in t-resilient strong consensus protocols.
– A proof that any weak consensus protocol that tolerates t crash failures must use at least one object on which at least $t + 1$ processes can invoke powerful operations.
– A tight tradeoff characterizing the number of a kind of powerful shared object that must be used and how big n must be as a function of t. Specifically, we show that the number k of powerful shared objects needed to implement consensus is essentially $k = t \cdot 2^{\Theta(t^2/n)}$.

– A polynomial-time strong consensus protocol (and therefore universality result) for $n = O(t^2/\log t)$, an improvement over the previous $n = \Omega(t^2)$.

2 Model and Definitions

The model of computation we consider was introduced by Malkhi et al., and parts of this section are adapted from [MMRT00]. This model consists of an asynchronous collection of n processes, denoted $p_1, \ldots, p_n$, that communicate via shared objects. Wait-free shared memory fault models assume no bound on the number of potentially faulty processes—each operation by a process p on a shared object must terminate, regardless of the concurrent actions of other processes. Following [MMRT00], this model differs in two ways: we make the more pessimistic assumption that process faults are Byzantine, and we make the more optimistic assumption that the number of faults is bounded by t, where t is less than the total number n of processes. In any run any process may be either *correct* or *faulty.* Correct processes are constrained to obey their specifications. A faulty processes can either crash or behave in a Byzantine way. A process that follows its protocol up to a certain point and then stops sending messages or stops accessing shared objects is called a *crashed process* or a *crash fault*. A process that deviates from its protocol either by crashing or by performing incorrect operations is called a *Byzantine process* or a *Byzantine fault*. We generally use t to denote the maximum number of faulty processes. Whenever we discuss faulty processes we identify the type of fault assumed.

2.1 Shared Objects with Access Control Lists

Each shared object presents a set of operations. For example, $x.\mathsf{op}$ denotes operation op on object x. For each such operation $x.\mathsf{op}$ on x, there is an associated access control list, denoted $\mathrm{ACL}(x.\mathsf{op})$, which is the set of processes allowed to *invoke* that operation. Each operation execution begins with an invocation by a process in the operation's ACL, and remains pending until a response is received by the invoking process. The ACLs for two different operations on the same object can differ, as can the ACLs for the same operation on two different objects. The ACLs for an object do not change. For any operation $x.\mathsf{op}$, we say that x is k-op if $|\mathrm{ACL}(x.\mathsf{op})| = k$. A process not in the ACL for $x.\mathsf{op}$ cannot invoke $x.\mathsf{op}$, regardless of whether the process is correct or Byzantine (faulty). That is, a (correct or faulty) process cannot access an object in any way except via the operations for which it appears in the associated ACLs. Byzantine faulty processes can, for example, write different values than their specifications suggest, or refuse to invoke an operation they are supposed to invoke, but they remain constrained against invoking operations for which they are not in the ACL.

Many abstract objects support read operations: operations that return information about the state of the object, without constraining its future behavior (see [Her91]). In this paper, we assume the primitive objects (registers and sticky bits) support wait-free read operations by all processes, and focus on the ACLs for non-read operations. Atomic registers (and some other abstract objects) are *historyless* [FHS98] in that they support only read operations and operations such as wait-free write() operations: operations that do not return a value, and that constrain future object behavior independently of the state in which they are invoked. These operations have long been known to be weak synchronization primitives [LA87,Her91]. Accordingly, we define an operation to be *powerful* if it is neither a wait-free read nor a wait-free write() operation, and for an object (or object type) x, we define $\mathrm{ACL}_{pow}(x)$ to be the union of $\mathrm{ACL}(x.\mathsf{op})$ for all powerful operations $x.\mathsf{op}$ of x. Moreover, we call an object (or object type) x *powerful* if $\mathrm{ACL}_{pow}(x) \geq 2$. That is, powerful objects support powerful operations by at least two different processes. Thus, neither registers nor sticky bits (defined below) writable by only one process are powerful, while sticky bits writable by more than one process are powerful.

2.2 Object Definitions

Next, we define some of the types of object used in this paper.

Atomic registers: An atomic register x is an object with two operations: $x.\mathsf{read}$ and $x.\mathsf{write}(v)$ where $v \neq \perp$. An $x.\mathsf{read}$ that occurs before the first $x.\mathsf{write}()$ returns $\perp$. An $x.\mathsf{read}$ that occurs after an $x.\mathsf{write}()$ returns the value written in the last preceding $x.\mathsf{write}()$ operation.

Sticky bits: A sticky bit x is an object with two operations: $x.\mathsf{read}$ and $x.\mathsf{set}(v)$ where $v \in \{0,1\}$. An $x.\mathsf{read}$ that occurs before the first $x.\mathsf{set}()$ returns $\perp$. An $x.\mathsf{read}$ that occurs after an $x.\mathsf{set}()$ returns the value written in the first $x.\mathsf{set}()$ operation. We will be concerned with wait-free sticky bits. (To highlight the specific semantics of the $x.\mathsf{set}()$ operation and to distinguish it from the write() of atomic registers, we depart from previous work [Plo89,MMRT00], which uses write() to denote both operations.)

Weak consensus objects: [MMRT00] A weak (binary) consensus object x is an object with one operation: $x.\mathsf{propose}(v)$, where $v \in \{0,1\}$, satisfying: (1) In any run, the $x.\mathsf{propose}()$ operation returns the same value, called the *consensus value*, to every correct process that invokes it. (2) In any finite run in which all participating processes are correct (no Byzantine faults), if the consensus value is v, then some process invoked $x.\mathsf{propose}(v)$.

Strong consensus objects: [MMRT00] A strong (binary) consensus object x strengthens the second condition above to read: (2) If the consensus value is v, then some *correct* process invoked $x.\mathsf{propose}(v)$.[1]

As indicated in [MMRT00], strong consensus objects do not have sequential runs: the additional condition, using redundancy to mask failures, requires at least $t+1$ processes to invoke $x.\mathsf{propose}()$ before any correct process returns from this operation. It is also shown in [MMRT00] that the notion of strong consensus objects that can tolerate t Byzantine faults is ill-defined unless $n \geq 3t+1$. To

[1] Note that in the binary case, this definition of strong consensus coincides with one that only requires the consensus value to be v when all correct processes have the same input. In the nonbinary case, this is a strictly stronger definition.

accommodate such a need to wait for other processes before progress can be made in specific constructions, the following properties were introduced:

For any operation $x.\mathsf{op}$, we say that $x.\mathsf{op}$ can *tolerate* t faults if $x.\mathsf{op}$, when executed by a correct process, eventually completes in any run ρ in which at least $n - t$ correct processes invoke $x.\mathsf{op}$.

t-resilience: For any operation $x.\mathsf{op}$, we say $x.\mathsf{op}$ is t-resilient if $x.\mathsf{op}$, when executed by a correct process, eventually completes in any run in which each of at least $n - t$ correct processes infinitely often has a pending invocation of $x.\mathsf{op}$.

As in [MMRT00], we use wait-free sticky bits to implement strong consensus objects that tolerate t Byzantine faults. These are in turn used to implement arbitrary t-resilient objects. (Throughout, atomic registers, sticky bits and any other primitive objects are assumed to be wait-free.)

3 Efficient Strong Consensus Protocols

In this section, we present strong consensus protocols based on an idea of Berman and Garay [BG89,Mis89] for performing consensus in the message passing model. Their idea was to run a protocol in phases, where each phase preserves agreement, and if coordinated by a correct process, assures validity. Concatenating $t + 1$ phases guarantees at least one is coordinated by a correct process.

We first show how to use sticky bits to implement a protocol phase with similar properties for the shared memory model. We then show how several strong consensus protocols with different desirable properties can be constructed from these protocol phases. Our protocols are easily modifiable to implement nonbinary consensus.

3.1 A Protocol Phase

A protocol phase makes use of two types of shared objects. First, a phase uses n *personal* sticky bits, $s_i : p_i \in P$, (where P is the set of all processes, $|P| = n$). Each s_i is writable only by the single process p_i. Second, a phase also uses a single powerful sticky bit, S, which is writable by some set of $t + 1$ processes. We call the $t + 1$ processes in ACL($s_i.\mathsf{set}()$) *active* in the phase. Each process p_i enters the phase with a proposed consensus value in_i and leaves with the output value out_i.

Operation of a protocol phase, for each process $p_i \in P$:

1. Perform the wait-free $s_i.\mathsf{set}(in_i)$ operation. (That is, assign in_i to the personal sticky bit s_i.)
2. Perform wait-free reads of the personal sticky bits $s_1 \ldots s_n$ until seeing at least $t + 1$ distinct occurrences of some value v other than $\perp$.
3. If p_i has write access to S, then perform the wait-free $S.\mathsf{set}(v)$ operation. (Of course, the first such scheduled process succeeds; the value of S does not change after that.)

4. Perform wait-free reads of S until returning a value *val* other than $\perp$.
5. Perform wait-free reads of the personal sticky bits $s_1 \ldots, s_n$ until at least $n - t$ return with values other than $\perp$. If *val* occurs in at least $t + 1$ of the values read, return $out_i = val$ (and say that p_i *supports val* in this phase), else return $out_i = \overline{val}$.

Lemma 1. *A protocol phase has the following properties (given $n \geq 3t + 1$ and the bound t on the number of Byzantine processes):*

1. *If at least $n-t$ correct processes enter a phase, all correct processes eventually exit it.*
2. *The output value of any correct process is the input value of some correct process.*
3. *If all the active processes are correct, all correct processes exit the phase with the same value v.*

Proof. 1. The first and third steps are wait-free. Once $n - t$ correct processes finish the first step, since $n - t \geq 2t + 1$, some value must occur at least $t + 1$ times, and the second and fifth steps must eventually terminate. The fourth step of all processes must terminate once a single correct active process executes the fourth step, and this must eventually happen because there are $t + 1$ active processes.

2. This follows because the output value of any correct process appears in at least $t + 1$ personal sticky bits, so one must have been set by a correct process.

3. If all the active processes are correct, then the value that S is set to will be supported by every correct process. (Note that this property implies that if all correct processes enter the phase with the same input value, then all correct processes exit the phase with the same output value.) ■

Now consider a consensus protocol, *Schema*, constructed by chaining finitely many separate protocol phases (with different sticky bits) together in a fixed sequence. Each process enters the first phase with its proposed value, uses the value returned from each phase as the input to the next phase, and returns as the protocol output the value returned from the last phase.

Lemma 2. *Given $n \geq 3t + 1$ and the bound t on the number of Byzantine processes, if any phase has only correct active processes, Schema implements strong consensus.*

Proof. Part 1 of Lemma 1 guarantees the correct processes eventually exit each phase and so *Schema*. Part 2 guarantees the correct processes enter each phase with a valid input. The assumption and part 3 guarantee the correct processes eventually agree in a phase, and again part 2 guarantees the correct processes do not change their value thereafter. ■

3.2 Strong Consensus Protocols

We present three protocols that use the protocol phases of Section 3.1 to implement strong consensus. The first protocol (Theorem 1) works for any $n \geq 3t+1$, but requires exponentially many powerful objects. Since Malkhi et al. [MMRT00] show that $n \geq 3t+1$ is necessary for strong consensus in this model, our protocol is optimal in terms of the ratio between t and n. The second protocol (Theorem 2) uses only $t+1$ powerful objects, but requires $n \geq (t+1)^2$. The third protocol (Theorem 3) is surprising because it works even if the faulty processes have access to all the powerful objects. It modifies *Schema* by replacing the requirement that some phase contains only correct active processes by a voting step at the end. It uses only t powerful objects and requires $n \geq t^2 + 5t + 1$.

Theorem 1. *A strong consensus object tolerating* t *Byzantine faults can be implemented using* $(t+1)$-set(), n-read *sticky bits and* 1-set(), n-read *sticky bits, provided that* $n \geq 3t+1$.

Proof. Let P' be a subset of P with $2t+1$ processes. The protocol consists of $\binom{2t+1}{t}$ phases, each following the other, where the active processes in each phase consist of a distinct subset of $t+1$ processes from P'. Since only t processes are faulty, one such phase contains only correct active processes, and the theorem follows from Lemma 2. ■

The implication for universality is immediate from Malkhi et al. [MMRT00]:

Corollary 1. *Any t-resilient object can be implemented using* $(t+1)$-set(), n-read *sticky bits and* 1-set(), n-read *sticky bits, provided that* $n \geq 3t+1$.

Though optimal in n and t, the protocol of Theorem 1 is not efficient in time or the number of powerful objects, as it uses a number of rounds and of powerful objects exponential in t. We will show in Section 4 that the space bound is inherent: an exponential number of powerful objects is *required* when $n = 3t+1$.

The following instantiation of *Schema* uses only $t+1$ rounds and $t+1$ powerful objects, but requires $n \geq (t+1)^2$. In this instantiation, a completely new set of $t+1$ active processes is used for each protocol phase:

Theorem 2. *A strong consensus object that can tolerate* t *Byzantine faults can be implemented using* $t+1$ $(t+1)$-set(), n-read *sticky bits, together with* 1-set(), n-read *sticky bits, provided* $n \geq (t+1)^2$.

Proof. There are $t+1$ phases, each with a distinct set of active processes, so for at least one phase all active processes are correct. The result follows by Lemma 2. ■

Theorems 1 and 2 are two extreme points in a tradeoff between the number of powerful objects and the ratio of t to n. We examine this tradeoff more closely in Section 4.2. First, we present a final protocol that is surprising because there are only t powerful objects, and therefore it works even if t Byzantine processes can access *all* the powerful objects.

Theorem 3. *A strong consensus object tolerating t Byzantine faults can be implemented using t powerful $(t+1)$-$\mathsf{set}()$, n-read sticky bits, together with 1-$\mathsf{set}()$, n-read sticky bits, provided that $n \geq t^2 + 5t + 1$.*

Proof. We modify the protocol of Theorem 2 by omitting the last phase. That is, there are t phases, each involving a distinct set of active processes accessing the $(t+1)$-$\mathsf{set}()$, n-read sticky bit for that phase. We then designate exactly $4t+1$ additional processes (there are at least that many) not active in any phase as *voters*. Note that either some phase contains only correct active processes, or all the voters are correct. Each of the voting processes takes their output from the last phase and writes their resulting value in a personal sticky bit. After the last phase, all processes read the voters' personal sticky bits, and decide on the first value they see occurring $2t+1$ times.

To see that this works, first note that at most one value can occur $2t+1$ or more times among the $4t+1$ voters. We now show one value must occur at least that often, and that the value is valid. By parts (2) and (3) of Lemma 1, if there is a phase in which all the active process are correct, then all correct voters will write the same valid value to their own personal sticky bit. Since in this case at least $3t+1$ voters are correct, eventually at least $3t+1 \geq 2t+1$ votes will be written and will agree on some valid value v. If there is no phase in which all the active processes are correct, then as argued above, no voter is faulty. In this case, all voters will write a valid value, so one value will be written at least $\lceil (4t+1)/2 \rceil = 2t+1$ times. ■

Since there are only t powerful objects, this algorithm works even in the case that no single powerful object is accessible exclusively by correct processes. In this case, the size, $t+1$, of the ACLs is needed to ensure that each powerful object will eventually be written (and so other processes can wait for it to be set), rather than ensuring that some powerful object will be written by a correct process. Other protocols derived from the protocol *Schema*, such as the algorithm in the proof of Theorem 1, can be similarly modified, removing one protocol phase and replacing it with a set of $4t+1$ voters.

One might expect that such tricks could be used to reduce the number of powerful objects even further. For example, can strong consensus be implemented with even fewer than t powerful sticky bits? Do t sticky bits suffice for $n = 3t+1$? (The t bit algorithm of Theorem 3 requires $n \geq t^2 + 5t + 1$.) We explore these questions in the next section.

4 Lower Bounds and Tradeoffs

We begin this section by showing in Theorem 4 that t is in fact a lower bound, regardless of n, on the number of powerful sticky bits needed for strong consensus. In the ensuing subsections, we examine the general tradeoff of number of sticky bits and n.

Theorem 4. *No t-resilient strong consensus object can be implemented from fewer than t powerful sticky bits (using no other powerful objects).*

Proof. Suppose such a protocol exists. Since it uses at most $t-1$ powerful sticky bits, the protocol must remain 1-resilient even in the case that $t-1$ Byzantine processes first set these bits to 0 before any correct process takes a step, and subsequently take no action. These sticky bits are clearly useless: omitting them and the $t-1$ Byzantine processes results in a 1-resilient protocol with *no* powerful objects that uses registers and single writer sticky bits. Obviously, this protocol is also correct in the crash-fault model (against the failure of a single process). But in the crash model, single writer sticky bits can be implemented by registers, and the resulting protocol contradicts the well-known results of Fischer, Lynch and Paterson [FLP85] and Loui and Abu-Amara [LA87]. ■

4.1 The ACL Theorem

We next prove a general theorem establishing a necessary condition for implementing *weak* consensus even in the presence of only *crash* faults. Inspired by proofs of [FLP85,LA87], we show that when at most t out of n processors may crash, the weak consensus problem is only solvable in shared memory systems containing a powerful object o such that $\mathrm{ACL}_{pow}(o) \geq t+1$. Trivially, a weak consensus object o defined with $|\mathrm{ACL}(o.\mathsf{propose}())| = t+1$ shows this condition is sufficient. Similarly, a sticky bit o with $|\mathrm{ACL}(o.\mathsf{set}())| = t+1$ easily implements a weak consensus object that tolerates up to t Byzantine faults.

Theorem 5. *Any weak consensus protocol that tolerates t crash failures must use at least one object, o, such that $|ACL_{pow}(o)| \geq t+1$.*

Proof (sketch). For the purposes of this proof, we denote a run of a protocol by the sequence in which processes invoke operations. A finite run x is *v-valent* if in all extensions of x where a decision is made, the decision value is v ($v \in \{0,1\}$). A run is *univalent* if it is either 0-valent or 1-valent, otherwise it is *bivalent.* In the following, P denotes a set of processes, x and x' denote runs and $x'p$ is an extension of the run x' by one step of process p.

Assume π is a t-resilient consensus protocol. By familiar arguments, π has an empty bivalent run x_0. We begin with x_0 and pursue the following round-robin *bivalence-preserving scheduling* discipline:

```
$x := x_0$; $P := \phi$; $i := 0$;
repeat
        $j := i + 1$
        if $x$ has a bivalent extension $x'p_j$
        then $x := x'$
        else $P := P + p_j$
        $i := i + 1 (\mathrm{mod}\ n)$
until $|P| = t + 1$.
```

If this procedure does not terminate, then there is an $(n-t)$-fair run with only bivalent finite prefixes. However, the existence of such a run contradicts the definition of t-resilient consensus protocols. Hence, the procedure will terminate with some bivalent finite run x, and a set P of $t+1$ processes such that any extension $x'p$ of x, for any process p in P, is univalent.

Pick any $p \in P$, and let v be such that the run xp is v-valent. Since x is bivalent, there is a (shortest) extension z of x which is $\overline{v}$-valent.

Let z' be the longest prefix of z that does not contain a step of p, and note that either $z = z'$ or $z = z'p$. From the assumption about z', it follows that $z'p$ is $\overline{v}$-valent, and $z' \neq x$.

Consider the extensions of x that are also prefixes of z'. Since xp and $z'p$ have opposite valencies, there must exist a P-free extension y of x and a process $q \neq p$, $x \leq y < yq \leq z'$, such that yp and yqp are univalent but with opposite valencies.

Now familiar case analyses [FLP85,LA87] preclude p from invoking wait-free read or write() operations: If p is a read, then ypq and yqp are indistinguishable to processes other than p, yet their p-free extensions must have opposite valencies, a contradiction. If p is a write(), then yp and yqp are indistinguishable to processes other than q, yet their q-free extensions must have opposite valencies, a contradiction. Since we chose p arbitrarily from P, we can conclude that no member of P invokes wait-free read or write() operations as their next step.

Finally, let o be the object accessed by p in the last step of yp. If q does not access o in yq, then ypq and yqp are indistinguishable, another contradiction. Suppose some $p' \in P$ accesses an object other than o. Either yp' is v-valent, and the indistinguishability of $yp'qp$ and $yqpp'$ leads to a contradiction, or yp' is $\overline{v}$-valent, and similarly the indistinguishability of ypp' and $yp'p$ leads to a contradiction. It follows that all $t+1$ processes in P invoke powerful operations on o. We conclude that $|\mathrm{ACL}_{pow}(o)| \geq t+1$. ■

In the next subsection, Theorem 5, in combination with the protocols of the previous section, is a powerful tool in establishing an asymptotic tradeoff on the number of sticky bits (or other powerful objects) necessary to implement strong consensus, as n varies relative to t.

4.2 On the Number of Powerful Objects

Theorem 5 states that any weak consensus protocol that tolerates t crash failures must use at least one object on which at least $t+1$ processes can invoke powerful operations. Section 3, and in particular Theorem 1, shows that sticky bits are examples of such objects that allow consensus protocols tolerating t Byzantine failures, even for $n = 3t+1$. The combination of these results shows that a primitive shared object type, o, with $|\mathrm{ACL}_{pow}(o)| = t+1$, is necessary and sufficient to implement strong consensus.

However, these results ignore the natural efficiency consideration of the *number* of powerful objects that are needed. In particular, the protocol for $n = 3t+1$ uses an exponential number (in t) of powerful sticky bits. Can we do with much fewer (e.g., polynomially many) powerful objects? Since a strong consensus object sc would of course trivially implement itself (with $\mathrm{ACL}_{pow}(sc.\mathsf{propose}()) = n$), some care is needed to generalize the question from the specific case of how many sticky bits are necessary. Accordingly, we define an object o to be *subvertible* if it is powerful and any Byzantine process in ACL(o) can cause operations by correct processes to be useless—that is, the return values of correct processes are dependent only on the operations of the Byzantine process. Sticky bits are subvertible, since a Byzantine process can invoke $\mathsf{set}(0)$ before any correct process takes a step. Obviously, if the object supports $\mathsf{write}()$ operations it is subvertible, and of course, strong consensus itself is not subvertible.

This section uses a combinatorial analysis to provide a tight asymptotic tradeoff between the number, k, of subvertible objects that must be used and the number of processes, n (as a function of the number of possible failures t), in the case that only subvertible objects are used. This tradeoff is essentially given by $k = t \cdot 2^{\Theta(t^2/n)}$. In particular, when $n = 3t + 1$, an exponential number of subvertible objects is indeed necessary. Interestingly, we obtain a protocol for $n = O(t^2/\log t)$ that uses a polynomial number of sticky bits (such a protocol was only previously known for $n = \Omega(t^2)$ [MMRT00]).

More formally, this section gives a proof of the following complementary theorems.

Theorem 6. *For any $n \geq 3t + 1$, there exists a strong consensus protocol that tolerates t Byzantine failures and the only powerful objects it uses are* $\max\{t + 1, t \cdot 2^{O(t^2/n)}\}$ *sticky bits with access lists of size $t + 1$.*

Theorem 7. *For any $n \geq 3t + 1$ and any constant $0 < \alpha < 1$, any strong consensus protocol that tolerates t Byzantine failures using only subvertible objects must use at least $t \cdot 2^{\Omega(t^2/n)}$ subvertible objects with $|ACL_{pow}(o)| \geq (1 - \alpha)t$ for each such object o.*

As a corollary of Theorem 6 we get the promised protocol for $n = O(t^2/\log t)$ that only uses a polynomial number of sticky bits

Corollary 2. *For any $n \geq 3t + 1$ such that $n = O(t^2/\log t)$, there exists a strong consensus protocol that tolerates t Byzantine failures and the only powerful objects it uses are a polynomial number (in t) of sticky bits with access lists of size $t + 1$.*

Correctness of the first two protocols in the previous subsection depend on their being enough such objects that at least one is accessed by *only* correct processes. The third allows all the powerful objects to be accessed by Byzantine processes, but in this case the protocol uses the guarantee that there are many additional processes, none of whom is even crash faulty. The former property is a combinatorial property of the sticky bit access control lists, which we call being "t-immune": no set of t (faulty) processes can subvert all the powerful sticky bits. This is formalized in the following definition: A collection of subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of the domain $[n] = \{1, 2, \ldots, n\}$ is *m-immune* if for every set $T \subseteq [n]$ of m elements there exists $S_j \in \mathcal{S}$ such that $T \cap S_j = \emptyset$. We note that related objects have been studied extensively in the past [EFF85,GGL95].

Theorem 3 shows that strong consensus protocols can exist even when the access lists of the powerful objects are not t-immune. However, we now argue that m-immunity (for some m that depends on t) is a necessary condition. As a simple corollary of Theorem 5 (and generalizing Theorem 4) we have the following:

Corollary 3. *Let π be any strong consensus protocol that tolerates $t_1 \geq 1$ crash failures and $t_2 \geq 1$ Byzantine failures and uses only subvertible objects. Let $\mathcal{S}$ be the collection of $ACL_{pow}(o)$ for all powerful objects o used by π that are of size at least $t_1 + 1$. Then $\mathcal{S}$ is t_2-immune.*

Proof (sketch). Assume to the contrary that $T_2 = \{i_1, \ldots, i_{t_2}\}$ is a set of t_2 processes that cover $\mathcal{S}$ (i.e. for all $S_j \in \mathcal{S}$, $T_2 \cap S_j \neq \emptyset$). Then π contains runs in which all the processes in T_2 are Byzantine. These processes can subvert all the objects with large (size $t_1 + 1$) access lists, making these objects useless in fighting the remaining t_1 crash failures. More formally, it is easy to modify π so that it will still tolerate t_1 crash failures but will use no object o with $\mathrm{ACL}_{pow}(o)$ bigger than t_1. Since this is a contradiction to Theorem 5, the corollary follows. ∎

The next corollary follows trivially from Corollary 3.

Corollary 4. *Let π be any strong consensus protocol that tolerates t Byzantine failures and uses only subvertible objects. Let $0 < \alpha < 1$ be some constant such that $(1-\alpha)t > 1$, and let $\mathcal{S}$ be the collection of $ACL_{pow}(o)$ for all powerful objects o used by π that are of size at least $\lfloor (1-\alpha)t \rfloor + 1$. Then $\mathcal{S}$ is $\lceil \alpha t \rceil$-immune.*

4.3 Bounds on the Size of m-Immune Collections

Combining Theorem 1 and Corollary 3, both upper and lower bounds on the number of subvertible objects needed by Byzantine consensus protocols (Theorems 6 and 7) can be derived from corresponding bounds on the number of sets in m-immune collections. Such bounds will be given in this section. The techniques we use to derive these bounds are quite common but we were not able to deduce our results directly from previous work. We note that the results given here contain improvements and significant simplifications due to Noga Alon [Alo02].

Theorem 8. *Let $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ be an m-immune collection of subsets of the domain $[n]$. If all the sets $S_j \in \mathcal{S}$ contain at least $t + 1$ elements then*

$$k \geq \max\{m + 1, m \cdot 2^{\Omega(t \cdot m/n)}\}.$$

Proof. That $k \geq m + 1$ is trivial (any collection of less than $m + 1$ subsets can be covered by m elements). To obtain the rest of the bound we apply the probabilistic method (see [ASE92]). Consider a random subset $T' \subset [n]$ of size $m/2$ (we assume without loss of generality that m is even). For any $j \in [k]$,

$$\Pr[T' \cap S_j = \emptyset] < \left(1 - \frac{t+1}{n}\right)^{m/2}.$$

Therefore, the expected number of $j \in [k]$ such that $T' \cap S_j = \emptyset$ is smaller than $k\left(1 - \frac{t+1}{n}\right)^{m/2}$. If $k\left(1 - \frac{t+1}{n}\right)^{m/2} \leq m/2$ than there exists an assignment to the set T' that covers all but $m/2$ of the subsets S_j. The remaining subsets can be covered by additional $m/2$ elements in $[n]$, which contradicts the assumption that $\mathcal{S}$ is m-immune. We can therefore conclude that $k > m/2 \cdot \left(1 - \frac{t+1}{n}\right)^{m/2}$, completing the proof. ■

An upper bound on the number of sets in m-immune collections can also be obtained by an application of the probabilistic method. Nevertheless, we are interested in an *explicit construction* of the m-immune collection (so that the resulting consensus protocol will be explicit as well). The proof of the following theorem provides such a construction.

Theorem 9. *For any n, t and m with $n > t + m + 1$, there exists an m-immune collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of subsets of the domain $[n]$, such that (1) all the sets $S_j \in \mathcal{S}$ contain at least $t + 1$ elements and (2)*

$$k \leq \max\{m + 1, m \cdot 2^{O(t \cdot m/n)}\}.$$

Proof. We can assume without loss of generality that $n < (m + 1)(t + 1)$ and that $n, t + 1$ and m are all (positive) powers of 2. The first assumption is easy to justify: if $n \geq (m + 1)(t + 1)$ we can simply define $\mathcal{S}$ to be a collection of $m + 1$ disjoint subsets of size $t + 1$. To justify the second assumption, we note that it is sufficient to construct our collection for $n' \leq n$, $t' \geq t$, and $m' \geq m$ that are the closest values such that $n', t' + 1$ and m' are all powers of 2. For this argument to work we must have that $n' > t' + m' + 1$ which is true as long as $n > 4(t + m) + 1$. (Otherwise, if $n \leq 4(t + m) + 1$, we have a simple direct construction of the desired collection: Just take all the subsets of size $t + 1$ of the first $t + m + 1$ elements. Since $n \leq 4(t + m) + 1$ we have that $k = \binom{t+m+1}{m} < 2^{t+m+1} = 2^{O(t \cdot m/n)}$ as claimed.)

We can now define the desired m-immune collection. Set $\ell = 2(t + 1)m/n$. By our assumptions ℓ is an integer and furthermore ℓ divides $t + 1$ (as both are powers of 2 and $\ell \leq t + 1$). Set $r = n/2(t + 1)$. By our assumptions this also is an integer and furthermore $r < m$. Now, let us divide the set $[n]$ into $2\ell r$ disjoint subsets of equal size: $A_{1,1}, \ldots A_{1,2\ell}, A_{2,1}, \ldots, A_{2,2\ell} \ldots, A_{r,2\ell}$. We can now define the collection $\mathcal{S}$. For any $i \in [r]$ and any ℓ distinct indices $j_1, j_2, \ldots, j_\ell$ in $[2\ell]$, the collection will contain the set

$$S_{i,j_1,j_2,\ldots,j_\ell} = A_{i,j_1} \cup A_{i,j_2} \cup \ldots \cup A_{i,j_\ell}.$$

By definition, the number of sets in the collection is $r \cdot \binom{2\ell}{\ell} < r \cdot 2^{2\ell} = m \cdot 2^{O(t \cdot m/n)}$. Furthermore, each set contains $n/2r = t + 1$ elements. It remains to show that $\mathcal{S}$ is m-immune. Let T be any subset of $[n]$ of size m. It is easy to see that there exists at least one $i \in [r]$ such that T contains at most $m/r = \ell$ elements in $\cup_{j=1}^{2\ell} A_{i,j}$. Therefore, T contains elements in at most ℓ sets $\{A_{i,j}\}_{j=1}^{2\ell}$. Therefore, there exist distinct indices $j_1, j_2, \ldots, j_\ell$ in $[2\ell]$, such that $T \cap S_{i,j_1,j_2,\ldots,j_\ell} = \emptyset$. ■

5 Conclusions

We presented a t-resilient strong consensus protocol (and therefore universality result) from sticky bits for $n \geq 3t+1$. We demonstrated a tight tradeoff between the fault tolerance and the efficiency of any strong consensus protocol using subvertible objects such as sticky bits, in particular showing that any strong consensus protocol using only subvertible objects must use a super-polynomial number of objects. The tradeoff also implies a polynomial time t-resilient strong consensus protocol for $n = O(t^2/\log t)$.

It remains open whether sticky bits are universal for $n \leq 3t$. Since strong consensus is not possible for $n \leq 3t$, a different universality construction not going through strong consensus would be needed.

Acknowledgements. We are grateful to Noga Alon for his simplifications and improvements that appear in Section 4.3. We would also like to thank Yuval Ishai and Juan Garay for useful discussions.

References

[Alo02] N. Alon. Private communication.

[ASE92] N. Alon, J.H. Spencer, and P. Erdős. *The probabilistic method.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley and Sons, Inc., 1992.

[BG89] P. Berman and J.A. Garay. Asymptotical optimal distributed consensus. *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP 89)*, LNCS 372, pp. 80–94, 1989.

[EFF85] P. Erdös, P. Frankl and Z. Füredi. Families of finite sets in which no set is covered by the union of r others. *Israel Journal of Mathematics* 51:75-89, 1985.

[FHS98] F. Fich, M. Herlihy, and N. Shavit, On the Space Complexity of Randomized Synchronization. *Journal of the ACM*, 45(5):843–862, September 1998.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[GGL95] M. Grötschel, R. L. Graham, and L. Lovász, *Handbook of Combinatorics.* Vol. 1, 2. MIT Press. 1995.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3):463–492, July 1990.

[Her91] M.P. Herlihy. Wait-free synchronization, *ACM Transactions on Programming Languages and Systems* 13(1):124–149, January 1991.

[LA87] M.C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[Mis89] J. Misra. A simple proof of a simple consensus algorithm. *Information Processing Letters*, 33(1):21–24, 1989.

[MMRT00] D. Malkhi, M. Merritt, M. Reiter, and G. Taubenfeld. Objects shared by Byzantine processes. *Proceedings of the 14th International Symposium on Distributed Computing (DISC 2000)*, LNCS 1914, pp. 345–359, 2000.

[Plo89] S.A. Plotkin. Sticky bits and universality of consensus. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, August 1989.

Failure Detection Lower Bounds on Registers and Consensus
(Extended Abstract)

Carole Delporte-Gallet[1], Hugues Fauconnier[1], and Rachid Guerraoui[2]

[1] Laboratoire d'Informatique Algorithmique: Fondements et Applications
Université Paris VII – Denis Diderot
`{cd,hf}@liafa.jussieu.fr`
[2] Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne
`rachid.guerraoui@epfl.ch`

Abstract. This paper addresses the problem of determining the weakest failure detector for consensus in a message passing system when t out of n processes can crash (including when $n/2 \leq t < n-1$), by addressing the problem of determining the weakest failure detector for register problem. We complement and, in a precise sense, generalize previous results on the implementability of consensus and registers in a message passing model (augmented with the failure detector abstraction).

1 Introduction

This paper considers an asynchronous distributed system augmented with the failure detector abstraction [5]. The system is made of n processes that communicate through reliable channels but t among these processes can fail by crashing [11]. The system is asynchronous in the sense that there is no timing assumption on communication delays and process relative speeds. In fact, timing assumptions [7] are encapsulated within the failure detector abstraction: a distributed oracle that provides each process with hints about failures that have occurred in the system.

Several classes of failure detectors were proposed in the literature, each gathering a set of failure detectors that ensure some abstract axiomatic properties. In particular, three interesting classes were identified in [5]: the class $\mathcal{P}$ of *Perfect* failure detectors that eventually suspect all crashed processes and never make false suspicions, the class $\mathcal{S}$ of *Strong* failure detectors that eventually suspect all crashes and never make false suspicions on at least one correct process (if there is such a process), and the class $\diamond\mathcal{S}$ of *Eventually Strong* failure detectors that eventually suspect all crashes and eventually never make false suspicions on at least one correct process (if there is such a process).

In [5], two algorithms using failure detectors were proposed to implement the consensus problem. In consensus, the processes propose an initial value and correct processes (those that do not crash) need to decide one among these values, such that no two processes decide differently [11]. We consider in this paper the *uniform* variant of consensus, which precludes any disagreement among two processes, even if one of them ends up crashing [15]. We will come back to the impact of this assumption in Section 6. The first algorithm of [5] implements consensus with any failure detector of $\mathcal{S}$ for any t, whereas the second algorithm implements consensus with any failure detector of $\Diamond\mathcal{S}$ if $t < n/2$. It was furthermore shown [6] that there is an algorithm that transforms any failure detector $\mathcal{D}$ that solves consensus into a failure detector of $\Diamond\mathcal{S}$. In the parlance of [6], $\Diamond\mathcal{S}$ is the *weakest* for consensus if $t < n/2$.

When $t \geq n/2$, $\Diamond\mathcal{S}$ is not the weakest for consensus [5]. *What is then the weakest failure detector class for consensus when* $t \geq n/2$? The question remained open for more than a decade now.

In a recent companion paper [8], we addressed this question for the specific case where $t \in \{n, n-1\}$. We first restricted the universe of failure detectors to those, we called *realistic*, that cannot predict the future. Basically, we focused on determining the weakest failure detector class among those that provide information about *past* failures and cannot provide information about failures that *will* occur [17,8]. Although failure detectors that predict the future are permitted in [5], they cannot be implemented even in the synchronous model of [22], where process relative speeds and communication delays are bounded, and these bounds are known. Note that $\Diamond\mathcal{S}$ is the *weakest* among *realistic* classes for consensus if $t < n/2$.

We then showed in [8] that, among *realistic* failure detectors, $\mathcal{P}$ is the weakest for consensus when $t \in \{n, n-1\}$. (We also showed that in this case the classes $\mathcal{P}$ and $\mathcal{S}$ are the same.) The case where $t \in \{n, n-1\}$ is actually very specific. *What happens when* $n/2 \leq t < n-1$? This question turns out to be challenging: none of the failure detector classes defined so far in the literature, including those of [5] (e.g., $\mathcal{S}$), is actually the weakest for consensus if $n/2 \leq t < n-1$.

The motivation of this work was precisely to address this question. While doing so, we faced another interesting problem: determining the weakest failure detector class for register problem when $n/2 \leq t \leq n$. What we actually mean here is to determine the weakest failure detector class for a *wait-free atomic register* in the sense of [4] [1], i.e., to implement a data abstraction accessed through *read()* and *write()* operations such that, despite concurrent accesses and failures of processes, (1) every operation invoked by a correct process eventually returns, (2) although it spans over an interval time, every operation appears to have been executed at a single point in time in this interval, and (3) every read operation returns the last value written. In a purely asynchronous message passing system model (e.g., without failure detectors) a register can be implemented iff $t < n/2$ [4]. To implement a register when $n/2 \leq t \leq n$, we need some information about failures. Several authors considered augmenting an asynchronous model with registers and failure detectors (e.g., [21,24]) but,

[1] The *wait-free* notion was introduced in [19] and the notion of *atomic register* was introduced in [20].

to our knowledge, the question of the weakest failure detector class for register problem in a message passing model was never addressed for $n/2 \leq t \leq n$.

This paper defines a new *generic* failure detector class $\mathcal{P}^k$ (*k-Perfect*). Instances of this generic class are determined by the integer k ($0 \leq k \leq n$). Failure detectors of class $\mathcal{P}^k$ eventually suspect all crashed processes and do not falsely suspect more than $max(n-k-1, 0)$ processes at every process. The processes might permanently disagree here on their perception on which processes have crashed, i.e., on the subset of $n-k-1$ processes each process falsely suspects. They might even change their mind about which processes they falsely suspect. For $k < n/2$, failure detectors of class $\mathcal{P}^k$ can be implemented in an asynchronous system if up to $t < n/2$ processes can crash, i.e., they do not encapsulate any timing assumption. For $k \geq n-1$, $\mathcal{P}^k$ is $\mathcal{P}$, i.e., $\mathcal{P}^{n-1} = \mathcal{P}^n = \mathcal{P}$.

We show that, if we assume that t processes can crash, then $\mathcal{P}^t \times \Diamond\mathcal{S}$ is the weakest failure detector class for consensus. To prove our result, we prove the interesting intermediate result that, if we assume that t processes can crash, then $\mathcal{P}^t$ is the weakest failure detector class for register problem. Our results hold among *realistic* failure detectors: the very fact that we exclude failure detectors that can predict the future is meaningful from a practical perspective [8], yet it has however some impact on our theoretical results, as we discuss in Section 6. To summarize, the contributions of this paper are the following:

1. We introduce a new failure detector class $\mathcal{P}^k$, and assuming that t processes can crash, we show that (among *realistic* failure detectors):
2. $\mathcal{P}^t$ is the weakest for register problem, and
3. $\mathcal{P}^t \times \Diamond\mathcal{S}$ is the weakest for consensus.

For the case where $n/2 \leq t < n-1$, we hence address the open questions of the weakest failure detector classes for consensus and atomic register. We also show that, in this case, $\mathcal{S}$ is strictly stronger than $\mathcal{P}^t \times \Diamond\mathcal{S}$, revisiting the first glance intuition that $\mathcal{S}$ might have been the weakest for consensus with $n/2 \leq t < n-1$. For $t \in \{n, n-1\}$, and given that in this case $\mathcal{P}^t$ is $\mathcal{P}$, our result comes down to the result of [8] as far as consensus is concerned, and we address the open question of the weakest failure detector for atomic register problem if any number of processes can crash. For $n/2 > t$, and given that in this case $\mathcal{P}^t$ can be implemented in an asynchronous system, our result simply comes down to the known result that, with a majority of correct processes, no timing assumption is needed to implement an atomic register [4], and that $\Diamond\mathcal{S}$ is the weakest for consensus [6].

Interestingly, the decoupled structure of our weakest failure detector class for consensus, *i.e.*, $\mathcal{P}^t \times \Diamond\mathcal{S}$, conveys its double role: $\mathcal{P}^t$ encapsulates the information about failures needed to ensure the safety part of consensus (i.e., to implement a register that will lock the consensus value and prevent disagreement), whereas $\Diamond\mathcal{S}$ encapsulates the information about failures needed to ensure the liveness part of consensus (i.e., to ensure that some correct will eventually stop being suspected and store a decision value in the register). We prove our results using simple algorithm reductions (like in [6] but unlike in [17]). Hence, by determining the weakest failure detector class for register problem (or consensus), we determine what exact information about failures processes need to know and effectively compute to implement a register (or consensus).

The rest of the paper is organized as follows. In Section 2, we sketch our system model and, in particular, we specify the universe of failure detectors within which we identify the weakest for consensus and atomic register. Section 3 defines our generic failure detector class $\mathcal{P}^k$. We compare instances of this class, and position them with respect to various failure detector classes introduced in the literature. Section 4 shows (sufficient condition) that, if t processes can crash, then $\mathcal{P}^t$ solves register problem. We then simply reuse the result of [21] to derive the fact than $\mathcal{P}^t \times \Diamond\mathcal{S}$ implements consensus. Section 5 shows (necessary condition) that if t processes can crash, then any failure detector for register problem can be transformed into a failure detector of $\mathcal{P}^t$. We then simply reuse the fact that consensus can implement a register and the result of [6] to show that any failure detector for consensus can be transformed into a failure detector of $\mathcal{P}^t \times \Diamond\mathcal{S}$. Section 6 concludes the paper by discussing the scope of our results. For space limitation, detailed proofs of our results are given in the full version of this paper [9] .

2 System Model

Our model of asynchronous computation with failure detection is the FLP model [11] augmented with the failure detector abstraction [5,6]. A discrete global clock is assumed, and Φ, the range of the clock's ticks, is the set of natural numbers. The global clock is used for presentation simplicity and is not accessible to the processes. We sketch here the fundamentals of the model. The reader interested in specific details about the model should consult [6,8].

2.1 Failure Patterns and Environments

We consider a distributed system composed of a finite set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ ($|\Pi| = n \geq 3$). A process p is said to *crash at time* τ if p does not perform any *action* after time τ (the notion of *action* is recalled below). Failures are permanent, i.e., no process *recovers* after a crash. A *correct* process is a process that does not crash. A *failure pattern* is a function F from Φ to 2^{Π}, where $F(\tau)$ denotes the set of processes that have crashed through time τ. The set of correct processes in a failure pattern F is noted *correct*(F). We say that a process p is *alive* at time τ if p is not in $F(\tau)$. An *environment* $\mathcal{E}$ is a set of failure patterns. Environments describe the crashes that can occur in a system. In this paper, we consider environments, denoted by $\mathcal{E}_t$, composed of all failure patterns with at most t crashes.

2.2 Failure Detectors

Roughly speaking, a failure detector $\mathcal{D}$ is a distributed oracle which gives hints about failure patterns. Each process p has a local failure detector module of $\mathcal{D}$, denoted by $\mathcal{D}_p$. Associated with each failure detector $\mathcal{D}$ is a range $R_{\mathcal{D}}$ (when the context is clear we omit the subscript) of values output by the failure detector. A *failure detector history* H with range R is a function H from $\Pi \times \Phi$ to R. For every process $p \in \Pi$, for every time $\tau \in \Phi$, $H(p, \tau)$ denotes the value of the failure detector module of process p at time τ, i.e., $H(p, \tau)$ denotes the value output by $\mathcal{D}_p$ at time τ. A *failure detector* $\mathcal{D}$ is defined as a function that maps each failure pattern F to a set of failure detector histories with range $R_{\mathcal{D}}$. $\mathcal{D}(F)$ denotes the set of possible failure detector histories permitted for the failure pattern F, i.e., each history represents a possible behavior of $\mathcal{D}$ for the failure pattern F.

In [5], any function of the failure pattern is a failure detector, including a function that, for a time τ, outputs information about crashes that will occur *after* τ. We restrict ourselves here to (*realistic* [8]) failure detectors $\mathcal{D}$ as functions of the *past*, i.e., we exclude failure detectors that can predict the future (we will come back to the ramifications of this assumption in Section 6). A failure detector cannot distinguish at a time τ two failure patterns based on what will happen after τ. More precisely, $\forall F, F' \in \mathcal{E}_n$, $\forall \tau \in \Phi$ s.t. $\forall \tau_1 \leq \tau, F(\tau_1) = F'(\tau_1)$, we have the following *realism* property: $\forall H \in \mathcal{D}(F), \exists H' \in \mathcal{D}(F')$ s.t. $\forall \tau_1 \leq \tau, \forall p \in \Pi : H(p, \tau_1) = H'(p, \tau_1)$.

Three classes of failure detectors introduced in [5] are of interest in this paper. These classes do all have range $R = 2^{\Pi}$: for any failure detector $\mathcal{D}$ in these classes, any failure pattern F, and any history H in $\mathcal{D}(F)$, $H(p, \tau)$ is the set of processes *suspected* by process p at time τ. (1) The class of *Perfect* failure detectors ($\mathcal{P}$) gathers all those that ensure *strong completeness*, i.e., eventually every process that crashes is permanently suspected by every correct process, and *strong accuracy*, i.e., no process is suspected before it crashes. (2) The class of *Strong* failure detectors ($\mathcal{S}$) gathers those that ensure *strong completeness* and *weak accuracy*, i.e., some correct process is never suspected (if there is such a process). (3) The class of *Eventually Strong* failure detectors ($\Diamond\mathcal{S}$) gathers those that ensure *strong completeness* and *eventual weak accuracy*, i.e., eventually, some correct process is never suspected (if there is such a process). Note that, in the context of this paper, we restrict these classes to failures detectors that are *realistic.*

2.3 Algorithms

An *algorithm* using a failure detector $\mathcal{D}$ is a collection A of n deterministic automata A_p (one per process p). Computation proceeds in steps of the algorithm. In each step of an algorithm A, a process p atomically performs the following three actions: (1) p receives a message from some process q, or a "null" message λ; (2) p queries and receives a value d from its failure detector module $\mathcal{D}_p$ ($d \in R_{\mathcal{D}}$ is said to be *seen* by p); (3) p changes its state and sends a message (possibly null) to some process. This third action is performed according to (a) the automaton A_p, (b) the state of p at the beginning of the step, (c) the message received in action 1, and (d) the value d seen by p in action 2. The message received by a process is chosen non-deterministically among the messages in the message buffer destined to p, and the null message λ. A *configuration* is a pair (I, M) where I is a function mapping each process p to its local state, and M is a set of messages currently in the message buffer. A configuration (I, M) is an initial configuration if $M = \emptyset$ (no message is initially in the buffer): in this case, the states to which I maps the processes are called *initial states.* A *step* of an algorithm A is a tuple $e = (p, m, d, A)$, uniquely defined by the algorithm A, the identity of the process p that takes the step, the message m received by p, and the failure detector value d seen by p during the step. A step $e = (p, m, d, A)$ is *applicable to a configuration* (I, M) if and only if $m \in M \cup \{\lambda\}$. The *unique* configuration that results from applying e to configuration $C = (I, M)$ is noted $e(C)$.

2.4 Schedules and Runs

A *schedule* of an algorithm A is a (possibly infinite) sequence $S = S[1]; S[2]; \dots S[k]; \dots$ of steps of A. A schedule S is applicable to a configuration C if (1) S is the empty schedule, or (2) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$ (the configuration obtained from applying $S[1]$ to C), etc.

Let A be any algorithm and $\mathcal{D}$ any failure detector. A *run of* of A *using* $\mathcal{D}$ is a tuple $R =< F, H, C, S, T >$ where H is a failure detector history and $H \in \mathcal{D}(F)$, C is an initial configuration of A, S is an infinite schedule of A, T is an infinite sequence of increasing time values, and (1) S is applicable to C, (2) for all k where $S[k] = (p, m, d, A)$, we have $p \notin F(T[k])$ and $d = H(p, T[k])$, (3) every correct process takes an infinite number of steps, and (4) every message sent to a correct process p is eventually received by p.[2]

2.5 Implementability

An algorithm A *implements* a problem B using a failure detector $\mathcal{D}$ in an environment $\mathcal{E}_t$ if every run of A using $\mathcal{D}$ in $\mathcal{E}_t$ satisfies the specification of B. We say that $\mathcal{D}$ solves B in $\mathcal{E}_t$ if there is an algorithm that implements B using $\mathcal{D}$ in $\mathcal{E}_t$. We say that a failure detector $\mathcal{D}1$ is *stronger* than a failure detector $\mathcal{D}2$ in environment $\mathcal{E}_t$ ($\mathcal{D}2 \preceq_t \mathcal{D}1$) if there is an algorithm (called a *reduction algorithm*) that transforms $\mathcal{D}1$ into $\mathcal{D}2$ in $\mathcal{E}_t$, i.e., that can *emulate* the output $\mathcal{D}2$ using $\mathcal{D}1$ in $\mathcal{E}_t$ [5]. The algorithm does not need to emulate all histories of $\mathcal{D}2$. It is required however that, for every run $R =< F, H, C, S, T >$ where $H \in \mathcal{D}1(F)$, the output of the algorithm with R be a history of $\mathcal{D}2(F)$. We say that $\mathcal{D}1$ is *strictly stronger* than $\mathcal{D}2$ in environment $\mathcal{E}_t$ ($\mathcal{D}2 \prec_t \mathcal{D}1$) if $\mathcal{D}2 \preceq_t \mathcal{D}1$ and $\neg(\mathcal{D}1 \preceq_t \mathcal{D}2)$. We say that $\mathcal{D}1$ is *equivalent to* $\mathcal{D}2$ in environment $\mathcal{E}_t$ ($\mathcal{D}1 \equiv_t \mathcal{D}2$), if $\mathcal{D}2 \preceq_t \mathcal{D}1$ and $\mathcal{D}1 \preceq_t \mathcal{D}2$.

Finally, we say that a failure detector $\mathcal{D}$ is the weakest for a problem B in environment $\mathcal{E}_t$ if (a. sufficient condition) $\mathcal{D}$ solves B in $\mathcal{E}_t$ and (b. necessary condition) any failure detector that solves B is stronger than $\mathcal{D}$ in $\mathcal{E}_t$.

[2] In fact, our results and proofs hold with a weaker assumption where we only require that (4') every message sent by a correct process to a correct process p is eventually received by p.

3 k-Perfect Failure Detectors

3.1 Definitions

We define in this section the generic class of *k-Perfect* ($\mathcal{P}^k$) failure detectors, where $0 \leq k \leq n$. This class is generic in the sense that its semantics depend on the value of the integer k. Failure detectors of class $\mathcal{P}^k$ output, at each process p and each time τ, a list of suspected processes $\mathcal{P}^k(p, \tau)$ (i.e., the range of $\mathcal{P}^k$ is 2^{Π}). These failure detectors ensure *strong completeness* as well as the following *k-accuracy* property: at any time τ, no process suspects more than $n - k - 1$ processes that are alive at time τ. More precisely:

– *k*- **Accuracy:** $\forall p \in \Pi$, $\forall \tau \in \Phi$ $|\mathcal{P}^k(p, \tau) \setminus F(\tau)| \leq max(n - k - 1, 0)$.

For $k \geq n - 1$, processes do not make false suspicions and $\mathcal{P}^k$ is $\mathcal{P}$. For $k < n-1$, processes can make false suspicions and can even permanently disagree on the processes they falsely suspect. To better illustrate the behavior of a *k-Perfect* failure detector, consider a system of 5 processes $\{p_1, p_2, p_3, p_4, p_5\}$ and the case $k = 2$. The failure detector should eventually suspect permanently all crashed processes and should not falsely suspect more that 2 processes at every process. Consider a failure pattern where p_1 and p_2 crash. It can be the case that after some time τ, p_3 permanently suspects $\{p_1, p_2, p_4, p_5\}$, p_4 permanently suspects $\{p_1, p_2, p_3, p_5\}$, and p_5 permanently suspects $\{p_1, p_2, p_3, p_4\}$. It can also be the case that after some time τ, p_5 forever alternately suspects $\{p_1, p_2, p_3\}$ and $\{p_1, p_2, p_4\}$.

The idea of limited accuracy is not new. Failure detectors of $\mathcal{S}$ already provide a limited form of accuracy, with respect to those of $\mathcal{P}$, in the sense that they only need to ensure *weak accuracy*: they can falsely suspect correct processes, as long as there is one correct process that is never suspected. Our notion of limited accuracy is different in that processes do not need to agree on a process they never suspect, as conveyed by our example above and specified by our Proposition 3 below. In [13,23], the notion of accuracy was further limited in the sense that only a subset of the processes need to satisfy it: again, and even if we consider the case of *weak accuracy*, the subset of processes should still agree on some process not to suspect. In [5], the authors introduced the notion of *k-Mistaken* failure detectors as those that can make k false suspicions. In our case, except when $k \in \{n - 1, n\}$ (*Perfect* failure detection), a failure detector of $\mathcal{P}^k$ can make an infinite number of mistakes.

We also define here the failure detector class $\mathcal{P}^t \times \diamond\mathcal{S}$. This class gathers failure detectors $\mathcal{D}$ of range $2^{\Pi} \times 2^{\Pi}$, such that given H_1 a history of a failure detector of $\mathcal{P}^t$ and H_2 a history of a failure detector of $\diamond\mathcal{S}$, for each process p and at each time τ, the value of the failure detector module $\mathcal{D}_p$ is a pair $(H_1(p, \tau), H_2(p, \tau))$.

3.2 Relationships

In the following, we give some general properties of $\mathcal{P}^k$ failure detectors. Certain properties follow trivially from the definition. Others are less trivial and can be found in the full paper [9].

Proposition 1. *$\forall t, k, k' \in [0, n]$, $k \leq k' \Rightarrow \mathcal{P}^k \preceq_t \mathcal{P}^{k'}$.*

Proposition 2. *$\forall t, t' \in [n/2, n-1]$, $t' < t \Rightarrow \mathcal{P}^{t'} \prec_t \mathcal{P}^t$.*

We already pointed out the fact that $\mathcal{P}^{n-1} = \mathcal{P}^n = \mathcal{P}$. We state below a relationship between $\mathcal{P}^t$ and $\mathcal{S}$.

Proposition 3. *$\forall t \in [1, n-2]$, $\mathcal{P}^t \times \Diamond\mathcal{S} \prec_t \mathcal{S}$.*

We show (later in the paper) that, for any t, $\mathcal{P}^t \times \Diamond\mathcal{S}$ is the weakest failure detector class for consensus in $\mathcal{E}_t$. Hence, a corollary of Proposition 3 above is that $\mathcal{S}$ is not the weakest for consensus in $\mathcal{E}_t$ if $t < n-1$. Note that we have shown in [8] that $\mathcal{P}$ (and hence $\mathcal{P}^{n-1}$ and $\mathcal{P}^n$) is equivalent to $\mathcal{S}$ in $\mathcal{E}_{(n-1)}$ and $\mathcal{E}_n$.[3]

We state below some relationships between $\mathcal{P}^k$ and $\perp$, the empty failure detector that never outputs anything: $\perp$ can thus be implemented in an asynchronous system. Through these relationships, we point out some situations in which $\mathcal{P}^k$ can be implemented in an asynchronous system.

Proposition 4. *$\forall t \in [0, n-1]$, $\mathcal{P}^{n-t-1} \equiv_t \perp$.*

Proposition 5. *$\forall t, t' \in [0, \lceil n/2 \rceil - 1]$, $\mathcal{P}^{t'} \equiv_t \perp$.*

To get an intuition of the implementability of $\mathcal{P}^k$ in an asynchronous system (i.e., the equivalence with $\perp$), consider a system of 5 processes and an environment where a majority of the processes are correct. We can implement $\mathcal{P}^2$ if up to 2 processes can crash as follows: processes periodically exchange messages, and every process waits for 3 messages and suspects the processes from which it did not receive messages (Figure 1).

Given that (as we show in the next section), for any $t \in [0, n]$, $\mathcal{P}^t$ is the weakest for register problem in $\mathcal{E}_t$, clearly, $\mathcal{P}^k$ cannot be implemented in an asynchronous system in any $\mathcal{E}_t$ where $k \geq t \geq n/2$.

[3] Remember that we consider realistic restrictions.

```
Every process $p$ executes the following code:
    Initialization:
        r:=0
    Task 1:
        repeat forever
            send($ARE\_YOU\_ALIVE, r$) to all
            wait until receive ($I\_AM\_ALIVE, r$) from $max(n-t, 1)$ processes
            $Output_p$:={$q$ | no message ($I\_AM\_ALIVE, r$) from $q$ received by $p$ }
            /* $Output_p$ is the output for $p$ of the failure detector $\mathcal{D}$ */
            r:=r+1
    Task 2:
        upon receive ($ARE\_YOU\_ALIVE, x$) from $q$
        send($I\_AM\_ALIVE, x$) to $q$
```

Fig. 1. Implementation of $\mathcal{P}^{n-t-1}$ in environment $\mathcal{E}_t$

4 The Sufficient Conditions

In this section, we show that, in any environment $\mathcal{E}_t$, (1) any failure detector of $\mathcal{P}^t$ solves register problem, and (2) any failure detector of $\mathcal{P}^t \times \diamond\mathcal{S}$ solves consensus. To show (2), we reuse the fact that consensus can be implementable with registers and any failure detector of $\diamond\mathcal{S}$ (in any environment) [21]. To show (1), we first give an algorithm (Figure 2) that implements a 1-writer–1-reader atomic register (for any reader or writer) using a failure detector of $\mathcal{P}^t$. Then, as in [2], we use the fact that a N-writer–M-reader atomic register can be implemented from 1-writer–1-reader atomic registers (see [18] for a tutorial on relationships between registers).

Our register implementation (Figure 2) is an adaptation of [4]. Roughly speaking, whereas [4] uses the assumption of a majority of correct processes to ensure a quorum property for read and write operations, we make use of $\mathcal{P}^t$ to ensure that quorum property. Basically, each process maintains the current value of the register. In order to perform its read (resp. write) operation, the reader (resp. the writer) sends a message to all and waits until it receives acknowledgments from (1) at least $max(n-t, 1)$ processes, and (2) from every process that is not suspected by its failure detector module.

Proposition 6. *With any failure detector of $\mathcal{P}^t$, Algorithm 2 solves 1-reader–1-writer atomic register (for any reader or writer) problem in environment $\mathcal{E}_t$.*

Corollary 1. *Any failure detector of $\mathcal{P}^t$ solves (n-writer–n-reader) register problem in $\mathcal{E}_t$.*

We reuse the result of [21] that consensus can be implementable with registers and any failure detector of $\diamond\mathcal{S}$ (in any environment) in order to deduce the following corollary:

Corollary 2. *Any failure detector of $\mathcal{P}^t \times \diamond\mathcal{S}$ solves consensus in $\mathcal{E}_t$.*

```
Every process $p$ (including $p_w$ and $p_r$) executes the following code:
    Initialization:
        $current := \bot$
        $last\_write := -1$
        upon receive $(WRITE, y, s)$ from the writer
        if $s > last\_write$ then
            $current := y$
            $last\_write := s$
            send$(ACK\_WRITE, s)$ to the writer
        upon receive $(READ, s)$ from the reader
            send$(ACK\_READ, last\_write, current, s)$ to the reader
Code for $p_w$ the (unique) writer:
    Initialization:
        $seq := 0$ /* sequence number */
    procedure $write(x)$
        send$(WRITE, x, seq)$ to all
        wait until receive $(ACK\_WRITE, seq)$
                from all processes not in $\mathcal{P}^t_{p_w}$
                and from at least $max(n-t, 1)$ processes
        $seq := seq + 1$
    end $write$
Code for $p_r$ the (unique) reader:
    Initialization:
        $rc := 0$ /* reading counter */
    function $read()$
        $rc := rc + 1$
        send$(READ, rc)$ to all
        wait until receive $(ACK\_READ, *, *, rc)$
                from all processes not in $\mathcal{P}^t_{p_r}$
                and from at least $max(n-t, 1)$ processes
        $a := \max\{v \mid (ACK\_READ, v, *, rc)$ is a received message$\}$
        if $a > last\_write$ then
            $current := v$ such that $(ACK\_READ, a, v, rc)$ is a received message
            $last\_write := a$
        return$(current)$
    end $read$
```

Fig. 2. Implementation of an 1-writer–1-reader atomic register

5 The Necessary Conditions

In this section, we show that, in any environment $\mathcal{E}_t$, (1) any failure detector class that solves register problem is stronger than $\mathcal{P}^t$, and (2) any failure detector class that solves consensus is stronger than $\mathcal{P}^t \times \diamond\mathcal{S}$.

To state (1) we give an algorithm in Figure 3 that emulates, with any failure detector $\mathcal{D}$ that solves 1-writer–n-reader register problem for any writer, a failure detector of $\mathcal{P}^t$. We only describe the algorithm (its proof is given in [9]). To state (2), we use (1) above plus the facts that, (2.1) one can implement a register using (uniform) consensus (e.g., through a fault-tolerant state machine replication approach [26]), and (2.2) any failure detector class that solves a consensus is stronger than $\Diamond\mathcal{S}$ [6].

The idea of the algorithm of Figure 3 is the following. We use exactly one 1-writer–n-reader register per process, and we also denote by p the register of process p. Each process p is the writer of its register. Periodically, each process writes in its register alternatively two different values: v_0 and v_1. The fact that the register has been implemented using a failure detector $\mathcal{D}$ has the nice consequence that, for any write operation that terminates at some time τ, all processes that have not crashed by time τ, except at most $max(n-t-1,0)$ processes, have *participated* in the operation (the fact that $\mathcal{D}$ is *realistic* is important here). A failure detector of $\mathcal{P}^t$ is emulated through a distributed variable where every process basically outputs (*suspects*) the list of processes that did not participate in a write operation. More precisely, each process p maintains, for each register r, a list of participants and the sequence number of the last write operation seen by the process p on r. Every message exchanged in the context of an operation is tagged with these lists and the sequence numbers of the last write operation on each register. When p ends its write operation, it suspects all processes that are not in its list for p. The details on how processes update the lists, as well as the sequence numbers of the write operations, are given in Figure 3.

Proposition 7. *For any failure detector class $\mathcal{D}$ that solves register problem in $\mathcal{E}_t$, we have: $\mathcal{P}^t \preceq_t \mathcal{D}$.*

From Proposition 7 and Corollary 1, we deduce:

Theorem 1. *For any t, $\mathcal{P}^t$ is the weakest failure detector class for register problem in environment $\mathcal{E}_t$.*

Given that a register cannot be implemented in asynchronous systems in environment $\mathcal{E}_t$ for $n \leq 2t$ [2], we deduce:

Corollary 3. *If $t \geq n/2$ then $\mathcal{P}^t$ cannot be implemented in an asynchronous system in $\mathcal{E}_t$.*

Let $\mathcal{D}$ be any failure detector that solves consensus in $\mathcal{E}_t$. From [6], $\mathcal{D}$ can be transformed into a failure detector of $\Diamond\mathcal{S}$ in environment $\mathcal{E}_t$. Moreover, with consensus it is possible to implement an atomic register. For this, we can first implement the strong version of uniform atomic broadcast (as defined in [1]) and with this kind of uniform atomic broadcast it is easy to implement an atomic register.[4]

Proposition 8. *For any failure detector class $\mathcal{D}$ that solves consensus in $\mathcal{E}_t$, we have: $\mathcal{P}^t \times \Diamond\mathcal{S} \preceq_t \mathcal{D}$.*

[4] Note that the classical definition of uniform atomic broadcast [16] is not sufficient for this.

```
/* Process $p$ repeatedly writes $v_0$, $v_1$ on register $p$ */
/* $Output_p$ is the output of the emulated failure detector */

Initialization:
    $\forall i : L[i] = \emptyset; last\_write[i] := 0;$

/*$L[p]$ is the list of participants in the write ($last\_write[i]$) operation on the register $p$ */
/*all messages are tagged with $(L, last\_write)$*/

When $p$ begins a new write operation on its register (register $p$)
    $last\_write[p] := last\_write[p] + 1$
    $L[p] := \{p\}$

When $p$ ends the current write operation on its register (register $p$)
    $Output_p = \Pi - L[p]$

When $p$ receives a message $m$ with tag $(M, lw)$
    forall $i$ do
        switch:
            case $lw[i] > last\_write[i]$:$L[i] := M[i] \cup \{p\}$
            case $lw[i] = last\_write[i]$:$L[i] := L[i] \cup M[i]$
            case $lw[i] < last\_write[i]$:skip
        $last\_write[i] := \max(last\_write[i], lw[i])$
```

Fig. 3. $T_{\mathcal{D}\rightarrow\mathcal{P}^t}$ - Emulation at process p of a failure detector in $\mathcal{P}^t$ from a register implementation using a failure detector.

From Proposition 8 and Corollary 2, we deduce that:

Theorem 2. *For any t, $\mathcal{P}^t \times \diamond\mathcal{S}$ is the weakest failure detector class for consensus in environment $\mathcal{E}_t$.*

6 Concluding Remarks

We discuss here the scope of our lower bound failure detection results. We first discuss the universe of failure detectors among which we determine the weakest class, then we consider the environment within which we state and prove our results, and finally we come back to the relationship between registers and consensus and the impact of uniformity.

6.1 The Failure Detector Universe

Our results hold among the original universe of failure detectors defined in [5], with one exception however: we exclude failure detectors that can guess the future [8]. These failure detectors cannot be implemented even in a synchronous system (in the sense of [22]).[5] From a theoretical perspective, excluding such

[5] Systems as in [10] that *enforce* perfect failure detection by explicitly crashing processes, make sure that these processes are suspected after they have crashed, i.e,

failure detectors has clearly an impact. For instance, there is a class (denoted by $\mathcal{M}$ in [14]) of failure detectors that outputs exactly the list of faulty processes since time 0, and which solves consensus and register problem for any number of possible crashes. This class is incomparable with $\mathcal{P}^t$ (this is not completely trivial and we give the proof in the full paper [9]). Hence, our results do not hold within the overall original universe of failure detectors [5]: this might explain why the questions we address in this paper remained open for more than a decade.

6.2 The Environments

Strictly speaking, our failure detection lower bound result on consensus does not generalize the result of [6]. Whereas [6] shows that any failure detector that implements consensus in *any* environment can be transformed into a failure detector of $\diamond\mathcal{S}$, we show that any failure detector that implements consensus in *any* environment where up to t processes can crash ($\mathcal{E}_t$), for *any* t, can be transformed into a failure detector of $\mathcal{P}^t \times \diamond\mathcal{S}$. Our result generalizes the result of [6] for environments of the form $\mathcal{E}_t$. It is not applicable to arbitrary environments, for instance where two specific processes p_1 and p_2 cannot crash in the same failure pattern.

6.3 The Uniformity of Consensus

We consider throughout the paper the uniform variant of consensus. If we consider a *correct-restricted* variant of consensus [25], where two processes can decide differently, as long as one of them is faulty, then $\mathcal{P}^t \times \diamond\mathcal{S}$ is not the weakest for consensus for $t \geq n/2$ in $\mathcal{E}_t$. Indeed, consider the class $\mathcal{P}_<$ of failure detectors (given in [12]) and defined through the strong accuracy property of $\mathcal{P}$ and the following *partial* completeness property: if a process p_i crashes, then eventually every correct process p_j such that $j > i$ permanently suspects p_i.[6] There is an algorithm given in [12] that implements correct-restricted consensus with $\mathcal{P}_<$ for any t. For $t \geq n/2$, $\mathcal{P}_<$ is not stronger than $\mathcal{P}^t$ in $\mathcal{E}_t$ (we give the proof in the full paper [9]). For $t \geq n/2$, $\mathcal{P}^t \times \diamond\mathcal{S}$ is thus not the weakest for consensus in $\mathcal{E}_t$ (we generalize here the observation of [8] for $t \in \{n-1, n\}$). In a sense, the equivalence we state in the paper between consensus in $\mathcal{E}_t$ and $\diamond\mathcal{S}$ plus a register resilient to t crashes does not hold for *correct-restricted* consensus, i.e., the latter does not always lead to implement a register in a message passing model (augmented with failure detectors). In other words, in such models, correct restricted consensus problem can be solved whereas an atomic register can not be implemented.

they do not predict the future. It is not clear however how our results apply to systems that somehow predict failures by turning suspicions into exclusions from the group [3].

[6] As with the failure detector class Ω_i, introduced in [24], $\mathcal{P}_<$ has a restricted completeness: some faulty process might never be suspected. However, the two classes are different: with $\mathcal{P}_<$, completeness is restricted *a priori*: the only faulty process that might never be suspected (by any one) is p_n. With Ω_i, except when $i = 1$, any faulty process might never be suspected (by any one).

Acknowledgements. Comments from Partha Dutta, Corine Hari, Idit Keidar, Petr Kouznetsov, and Bastian Pochon helped improve the quality of the presentation of this paper.

References

1. M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. *Thrifty Generic Broadcast.* Proceedings of the 14th International Conference, DISC 2000, Toledo, Spain. Springer Verlag (LNCS 1914), 2000.
2. H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.
3. K. Birman and R. van Renessee. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1993.
4. H. Attiya, A. Bar-Noy, and D. Dolev. *Sharing Memory Robustly in Message Passing Systems.* Journal of the ACM, 42(1), January 1995.
5. T. Chandra and S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems.* Journal of the ACM, 43(2), March 1996.
6. T. Chandra, V. Hadzilacos and S. Toueg. *The Weakest Failure Detector for Solving Consensus.* Journal of the ACM, 43(4), July 1996.
7. C. Dwork, N. Lynch and L. Stockmeyer. *Consensus in the presence of partial synchrony.* Journal of the ACM, 35(2), 1988.
8. C. Delporte-Gallet, H. Fauconnier and R. Guerraoui. *A Realistic Look at Failure Detectors.* Proceedings of the IEEE International Conference on Dependable Systems and Networks, Washington DC, June 2002.
9. C. Delporte-Gallet, H. Fauconnier and R. Guerraoui. *Failure Detection Lower Bounds on Registers and Consensus.* Technical Report LIAFA, (Paris 2002) and EPFL, IC/2002/030 (Lausanne 2002).
10. C. Fetzer. *Enforcing Perfect Failure Detection.* Proceedings of the IEEE International conference on Distributed Computing Systems, Phoenix, April 2001.
11. M. Fischer, N. Lynch and M. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* Journal of the ACM, 32(2), 1985.
12. R. Guerraoui. *Revisiting the Relationship Between the Atomic Commitment and Consensus Problems.* Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 972), 1995.
13. R. Guerraoui and A. Schiper. *Γ-Accurate Failure Detectors.* Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 1151), 1996.
14. R. Guerraoui. *On the Hardness of Failure Sensitive Agreement Problems.* Information Processing Letters, 79, 2001.
15. V. Hadzilacos. *On the Relationship Between the Atomic Commitment and Consensus Problems.* Proceedings of the International Workshop on Fault-Tolerant Distributed Computing, Springer Verlag (LNCS 448), 1986.
16. V. Hadzilacos, and S. Toueg. *A modular approach to fault-tolerant broadcasts and related problems.* Technical Report TR94-1425, Cornell University, May 1994.
17. J. Halpern and A. Ricciardi. *A Knowledge-Theoretic Analysis of Uniform Distributed Coordination and Failure Detectors.* Proceedings of the ACM Symposium on Principles of Distributed Computing, 1999.
18. P. Jayanti. *Wait-free Computing.* Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 972), 1995.

19. L. Lamport. *Concurrent Reading and Writing.* Communications of the ACM, 20(11), 1977.
20. L. Lamport. *On Interprocess Communication (parts I and II).* Distributed Computing, 1, 1986.
21. W-K. Lo and V. Hadzilacos. *Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems.* Proceedings of the International Workshop on Distributed Algorithms, Springer Verlag (LNCS 857), 1994.
22. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
23. A. Mostéfaoui and M. Raynal. *k-Set Agreement with Limited Accuracy Failure Detectors.* Proceedings of the ACM Symposium on Principles of Distributed Computing, 2000.
24. G. Neiger. *Failure Detectors and the Wait-Free Hierarchy.* Proceedings of the ACM Symposium on Principles of Distributed Computing, 1995.
25. G. Neiger and S. Toueg. *Simulating Synchronized Clocks and Common Knowledge in Distributed Systems.* Journal of the ACM, 40(2), April 1993.
26. F. Schneider. *Replication Management using the State Machine Approach.* Chapter in Distributed Systems, Addison-Wesley, 1993.

Improved Compact Routing Scheme for Chordal Graphs

Yon Dourisboure and Cyril Gavoille

LaBRI, Université Bordeaux
{Yon.Dourisboure,gavoille}@labri.fr

Abstract. This paper concerns routing with succinct tables in chordal graphs. We show how to construct in polynomial time, for every n-node chordal graph, a routing scheme using routing tables and addresses of $O(\log^3 n/\log\log n)$ bits per node, and $O(\log^2 n/\log\log n)$ bit not alterable headers such that the length of the route between any two nodes is at most the distance between the nodes in the graph plus two.

Keywords: Chordal graph, compact routing tables, tree-decomposition

1 Introduction

Delivering messages between pairs of processors is a basic activity of any distributed communication network. This task is performed using a routing scheme, which is a mechanism for routing messages in the network. The routing mechanism can be invoked at any origin node and be required to deliver a message to some destination node.

It is naturally desirable to route messages along paths that are as short as possible. The efficiency of a routing scheme is measured in terms of its *multiplicative stretch* (or *additive stretch*), namely, the maximum ratio (or surplus) between the length of a route produced by the scheme for some pair of nodes, and their distance. A straightforward approach to achieving the goal of guarantees optimal routes is to store a *complete routing table* in each node u in the network, specifying for each destination v the first edge (or an identifier of that edge, indicating the output port) along some shortest path from u to v. However, this approach may be too expensive for large systems since it requires $O(n \log d)$ memory bits for a node of degree d in an n-node network. Thus, an important problem in large scale communication networks is the design of routing schemes that produce efficient routes and have relatively low *memory requirements*.

The routing problem can be presented as requiring to assign two kinds of labels to every node of a graph. The first is the *address* of the node, whereas the second label is a data structure called the *local routing table*. The labels are assigned in such a way that at every source node u and given the address of any destination node v, one can decide the output port of an edge outgoing of u that leads to v. The decision must be taken locally in u, based solely on the two labels of u and with the address label of v. In order to allow each intermediate node to proceed similarly, a *header* is attached to the message to v. This header consists either of the destination label, or of a new label created by the current node.

It was shown in a series of papers (see, e.g., [PU89,ABNLP89,ABNLP90, AP90,AP92,TZ01b]) that there is a tradeoff between the memory requirements of a routing scheme and the worst-case stretch factor it guarantees. In [PU89] it is shown that every routing strategy that guarantees a multiplicative s stretched routing scheme for every n-node graph requires $\Omega(n^{1+1/(2s+4)})$ bits in total, so $\Omega(n^{1/(2s+4)})$ for local routing tables, for some worst-case graphs. Stronger lower bounds hold for small stretch factors. In particular, any multiplicative s stretched routing scheme must use $\Omega(\sqrt{n})$ bits for some nodes in some graphs for $s < 5$ [TZ01a], $\Omega(n)$ bits for $s < 3$ [FG97,GG01], and $\Omega(n \log n)$ bits for $s < 1.4$ [GP96]. More precisely, for $s = 1$ [GP96] showed that for every shortest path routing strategy and for all d and fixed $\epsilon > 0$ such that $3 \leqslant d \leqslant (1-\epsilon)n$, there exists a graph of degree bounded by d for which $\Omega(n \log d)$ bit routing tables are required simultaneously on $\Theta(n)$ nodes, matching with the memory requirements of complete routing tables. All the lower bounds presented above assume that routes and addresses can be computed and optimized by the routing strategy in order to decrease the memory requirement.

These lower bounds are motivations for the design of routing strategies with compact tables on more specific class of graphs. Here we non exhaustively list some of them. Regular topologies (as hypercubes, tori, cycles, complete graphs, etc.) have specific routing schemes using $O(\log n)$ bit for addresses and for routing tables (cf. [Lei92]). For non-regular topologies and wider class of graphs, several trade-offs between the stretch and the size of the routing tables have been achieved. In particular, for c-decomposable graphs [FJ90] (including bounded tree-width graphs), planar graphs [FJ89,Lu02], and bounded pagenumber graphs and bounded genus graphs [GH99]. More recently, a multiplicative $1+\epsilon$ stretched routing scheme for every planar graph, for every $\epsilon > 0$, with only $(\log n)^{O(1)}$ bit addresses and routing tables, has been announced in [Tho01]. For more detailed presentation of these schemes and for an overview of the other strategies and techniques, see [Gav01] and [Pel00a].

In this paper we investigate *chordal* graphs, namely the class of graphs containing no induced cycles of length greater than 3. Trees are chordal graphs. A member of this class is depicted on the left side of Fig. 1. Note that from an information theory point of view, there is no way to give a compact representation of the underlying topology of such graphs, unlike other regular topologies (as hypercubes, grid, Cayley graphs, etc.). Indeed, there are at least $2^{n^2/4-o(n^2)}$ non-isomorphic chordal graphs with n nodes by considering for instance *split-graphs*, namely a complete graphs of $\lceil n/2 \rceil$ nodes and with $\lfloor n/2 \rfloor$ extra nodes whose neighborhood is randomly selected into the clique.

Previous Works

If we insist on shortest path (i.e., optimal stretch $s = 1$), no strategy better than complete routing tables is known for chordal graphs. Nevertheless, every chordal graph whose its maximal cliques are of size $k + 1$ exactly, namely every k-tree, supports shortest path routing tables such that, at each node, the destination addresses using the same outgoing edge consists of at most 2^{k+1} blocks of consecutive addresses [NN98]. Actually, as shown in [Dou02], this result can be easily extended to every chordal graph of *maximum* clique $k+1$. Derived from [NN98], every chordal graph with maximum clique k has shortest path routing tables of $O((2^k + d)\log n)$ bits per node of degree d, and using addresses $\in [1, n]$.

Relaxing the address size from $\log n$ bits[1] to $2\log n$, [Dou02] has recently showed that chordal graphs of maximum clique k have an additive 2 stretched routing strategy with $O(k \log n)$ bit routing table per node. However, $k = \Theta(n)$ is possible, yielding a space bound not better than complete routing tables in the worst-case.

The only scheme providing a space bound independent of the size of the maximum clique is due to Peleg and Upfal [PU89]. Based on the construction of a multiplicative 3-spanner with $O(n \log n)$ edges [PS89] (namely, a spanning subgraph whose the distance between any two nodes does not exceed 3 times the original distance in the graph), [PU89] have constructed a multiplicative 3 stretched routing strategy for chordal graphs using $O(n \log^2 n)$ bits in total for tables and $O(\log^2 n)$ bit addresses. In their scheme, headers do not consist of the full destination addresses as in [NN98] and in [Dou02], but are of $O(\log n)$ bits only. However headers can change several times along the route. In particular, the scheme provides routes that are not loop-free path in the graph.

Anyway, the solutions of [NN98], of [Dou02], and of [PU89] do not guarantee short routing tables for all the nodes. In the worst-case, $\Omega(n \log n)$ bits might be required at some nodes.

Our Results

In this paper, we design for chordal graphs an additive 2 stretched routing scheme with addresses and routing tables of $O(\log^3 n/\log\log n)$ bits per node. The headers are of size $O(\log^2 n/\log\log n)$ bits, and once initialized by the source node of the message, headers are never changed along the route. As a consequence the routes provided by our scheme are loop-free. On the time complexity point of view, our scheme is polynomially constructible (actually the time complexity to setup all the data structures is bounded by $O(m + n\log^2 n)$ where m is the number of edges). Moreover, once the routing scheme is constructed, routing decisions at each node (including header initialization[2]) require a constant number of standard operations on $O(\log n)$ bit word.

[1] Logs are in base two.

[2] It should be clear that to get a constant time complexity, one does not take in account the time to copy headers from router's memory to the link registers. Otherwise, the time complexity depends on the header size.

Obviously, an additive 2 stretched routing scheme gives also a multiplicative 3 stretched scheme, since in the worst-case the route between two adjacent nodes might be of length 3. However, the routes provided by our scheme are relevant as we can show that, in the scheme of [PU89], there are some counter-example graphs with nodes at distance two and having routes of length 5. Moreover, since our routing scheme is loop-free it guarantees shortest path for trees, whereas in [PU89] the routes are not optimal whenever the depth of the tree is at least two.

Technically, our result is achieved by the use of two main ingredients: 1) the well known tree-decomposition in maximal cliques of chordal graphs; and 2) the recent compact and distributed data structures for trees, in particular answering efficiently routing queries and ancestor queries with small labels [AR01, KM01,KMS02,FG01,TZ01b]. At this step, it is worth to observe that additive r stretched routing scheme on chordal graphs cannot be reduced to the problem of routing in a suitable spanning tree of the graph. Indeed, as mentioned in [Pri97,BCD99], for every fixed integer r there is a chordal graph without tree r-spanners (additive as well as multiplicative).

The paper is organized as follows. In Section 2, we introduce some definitions as tree-decomposition of Robertson and Seymour, and hierarchical tree. In Section 3, we sketch the routing scheme, and then we give its correctness and its space complexity analysis.

2 Preliminaries

We need the notion of *tree-decomposition* used by Robertson and Seymour in their work on graphs minors [RS86].

Definition 1. *A tree-decomposition of a graph G is a tree T whose nodes are subsets of $V(G)$, such that (see an example on Fig. 1):*

1. $\bigcup_{X \in V(T)} X = V(G)$;
2. *for all* $\{u, v\} \in E(G)$, *there exists* $X \in V(T)$ *such that* $u, v \in X$; *and*
3. *for all* $X, Y, Z \in V(T)$, *if* Y *is on the path from* X *to* Z *in* T *then* $X \cap Z \subseteq Y$.

Proposition 1. (cf. [Die00]) *A graph G is a chordal graph if and only if there exists a tree-decomposition of G (polynomial-time constructible) such that for all $X \in V(T)$, X induced a maximal clique in G.*

From now we consider an arbitrary connected chordal graph $\mathcal{G}$ with n nodes. According to Proposition 1, let $\mathcal{T}$ be a tree-decomposition of $\mathcal{G}$ such that each one of its nodes induced a maximal clique in $\mathcal{G}$. It is clear that the number of maximal cliques in $\mathcal{G}$ is at most n, so $\mathcal{T}$ has at most n nodes.

It is well known that every tree T with n nodes has a node s, called *separator*, such that all connected components of $T \setminus \{s\}$ have at most $n/2$ nodes.

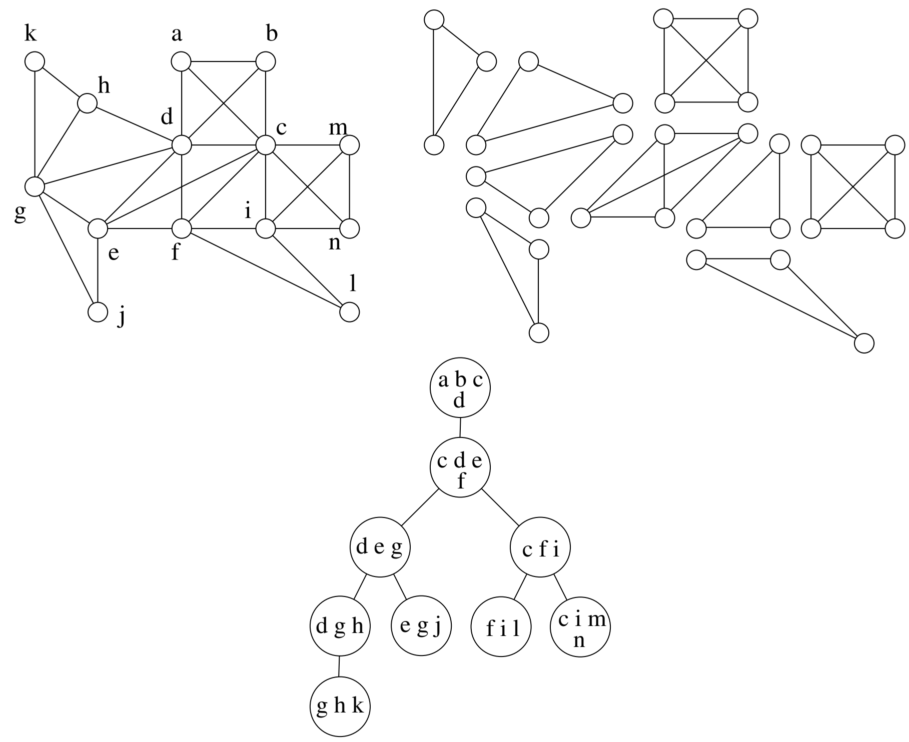


Fig. 1. From left-to-right: a chordal graph G, its set of maximal cliques, and a tree-decomposition of G satisfying Proposition 1.

A *hierarchical tree of* T is a rooted tree H defined recursively as follows: the root of H is s, the separator of T, and the children of s are the roots of the hierarchical trees of the connected components of $T \setminus \{s\}$. Observe that H and T share the same node set, and that the depth of H is at most $\log n$. Let $\mathcal{H}$ be any hierarchical tree of $\mathcal{T}$ (cf. Fig. 2).

We use standard notions of *children*, *parent*, *ancestors*, *descendants* and *depth* in rooted trees. For simplicity we assume that a node is an ancestor of itself. We denote by $nca_T(x,y)$ the nearest common ancestor of the nodes x and y in the rooted tree T. By construction of $\mathcal{H}$ we have:

Proposition 2. *Let Q be the path in $\mathcal{T}$ from U to V, and let $Z = nca_{\mathcal{H}}(U,V)$. Then $Z \in Q$, and Z is an ancestor in $\mathcal{H}$ of all the nodes of Q.*

We assume that $\mathcal{T}$ is rooted. For every node u of $\mathcal{G}$, the *ball* of u, denoted by $B(u)$, is a node X of $\mathcal{T}$ of minimum depth such $u \in X$. Observe that, once $\mathcal{T}$ has been fixed, $B(u)$ is unique for each u by Rule 3 of Definition 1.

To each node X of $\mathcal{H}$ we associate a shortest path spanning tree of $\mathcal{G}$, denoted by $\mathcal{S}_X$, rooted at any node r_X of $\mathcal{G}$ such that $B(r_X) = X$. Observe that for each X, the node r_X exists, otherwise X would not induce a maximal clique.

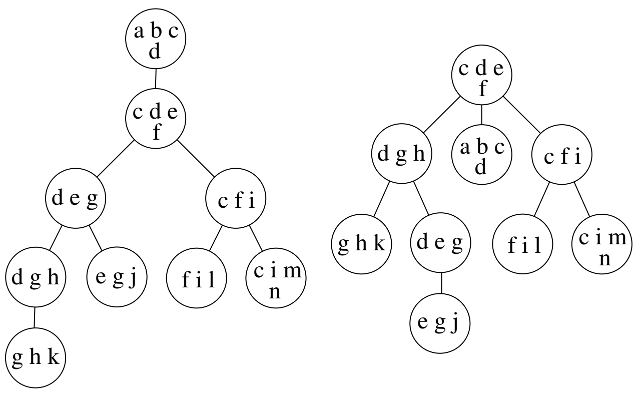


Fig. 2. The tree-decomposition $\mathcal{T}$ and the hierarchical tree $\mathcal{H}$.

Proposition 3. *Let u, v be two adjacent nodes of $\mathcal{G}$. One of these statements is true:*

1. *$B(u)$ is an ancestor of $B(v)$ in $\mathcal{T}$ and then $u \in B(v)$.*
2. *$B(v)$ is an ancestor of $B(u)$ in $\mathcal{T}$ and then $v \in B(u)$.*

Proof. Let u, v be two adjacent nodes of $\mathcal{G}$. If $B(u) = B(v)$ then the two statements are trivially trues (recall that we consider that a node is an ancestor of itself).

So, assume that $B(u) \neq B(v)$. By minimality of the depth of the balls and by definition of $\mathcal{T}$, every ball X containing a node x, is a descendant of $B(x)$ in $\mathcal{T}$. Moreover by Rule 2 of Definition 1, there exists a ball Y containing u and v. Thus Y is a descendant of $B(u)$ and of $B(v)$. Thus either $B(u)$ is an ancestor of $B(v)$ or the reverse. If $B(u)$ is an ancestor of $B(v)$ then $B(v)$ is on the path from $B(u)$ to Y in $\mathcal{T}$, by Rule 3 of Definition 1, $u \in B(v)$. Similarly if $B(v)$ is an ancestor of $B(u)$ then $v \in B(u)$. □

Definition 2. *Let P be a path in $\mathcal{G}$ from u to v, and let $Z = nca_{\mathcal{H}}(B(u), B(v))$. We define $except(P) = \{w \in P \mid B(w)$ is not a descendant of Z in $\mathcal{H}\}$.*

Lemma 1. *Let P be a shortest path in $\mathcal{G}$ from u to v, $except(P) \subset nca_{\mathcal{T}}(B(u), B(v))$. Therefore, $|except(P)| \leqslant 2$.*

Proof. Let $P = x_1, x_2, \ldots, x_l$ be a shortest path in $\mathcal{G}$ from $u = x_1$ to $v = x_l$, and let $Q = X_1, X_2, \ldots, X_m$ be the path in $\mathcal{T}$ from $B(u) = X_1$ to $B(v) = X_m$. By Proposition 2 each node x_i of P such that $B(x_i) \in Q$, is not in *except*(P). So, let us prove that each node x_i of P such that $B(x_i) \notin Q$, is in $nca_{\mathcal{T}}(B(u), B(v))$, and then we will prove Lemma 1. In this proof we will consider only the tree $\mathcal{T}$, so when we will talk about ancestor or descendant, it will always be in $\mathcal{T}$.

Let i_0 be the smallest index such that $B(x_{i_0}) \notin Q$. Note that $i_0 \neq 1$ and $i_0 \neq l$. Thus there exists $Y \in Q$ such that Y is a separator between x_{i_0} and v. Thus there exists $j > i$ such that $x_j \in Y$. Moreover by minimality of i_0, $B(x_{i_0-1}) \in Q$. Assume that $x_{i_0} \notin \mathit{nca}_{\mathcal{T}}(B(u), B(v))$. By Proposition 3, either $B(x_{i_0-1})$ is an ancestor of $B(x_{i_0})$, or $B(x_{i_0})$ is an ancestor of $B(x_{i_0-1})$. If $B(x_{i_0-1})$ is an ancestor of $B(x_{i_0})$, then $B(x_{i_0-1})$ is also an ancestor of Y, and by Rule 3 of Definition 1, $x_{i_0-1} \in Y$: a contradiction because the path $x_1, \ldots x_{i_0-1}, x_j, \ldots, x_l$ would be shorter than P which is a shortest path. Thus $B(x_{i_0})$ is an ancestor of $B(x_{i_0-1})$, thus $Y = \mathit{nca}_{\mathcal{T}}(B(u), B(v))$ and by Rule 3 of Definition 1, $x_{i_0} \in Y$.

Let i_1 be the largest index such that $B(x_{i_1}) \notin Q$, possibly $i_1 = i_0$. By replacing x_{i_0} by x_{i_1}, and, x_{i_0-1} by x_{i_1+1}, we obtain similarly that $x_{i_1} \in \mathit{nca}_{\mathcal{T}}(B(u), B(v))$.

Thus we have two nodes $x_{i_0}, x_{i_1} \in \mathit{nca}_{\mathcal{T}}(B(u), B(v))$. They are adjacent in $\mathcal{G}$, so there is no $x_{i'} \in P$ with $i_0 < i' < i_1$. □

Corollary 1. *Let u, v be two nodes of $\mathcal{G}$, $X = \mathit{nca}_{\mathcal{H}}(B(u), B(v))$, and let P_X be the path from u to v in the tree $\mathcal{S}_X$. Then $|\mathit{except}(P_X)| \leqslant 3$.*

Proof. Let u, v be two nodes of $\mathcal{G}$, $X = \mathit{nca}_{\mathcal{H}}(B(u), B(v))$, and let P_X be the path from u to v in the tree $\mathcal{S}_X$. Let P_{up} and P_{down}, the paths in $\mathcal{S}_X$ from u to r_X, respectively from r_X to v. Note that P_X is composed by a sub-path of P_{up} then by a sub-path of P_{down}. Thus $\mathit{except}(P_X) \subseteq \mathit{except}(P_{up}) \cup \mathit{except}(P_{down})$. By Lemma 1 $|\mathit{except}(P_{up})| \leqslant 2$ and $|\mathit{except}(P_{down})| \leqslant 2$.

Moreover, Proposition 2 states that X is on the path in $\mathcal{T}$ from $B(u)$ to $B(v)$, thus X is an ancestor in $\mathcal{T}$ either of $B(u)$, or of $B(v)$. If X is an ancestor of $B(u)$, then by Lemma 1 each node of $\mathit{except}(P_{up})$ is in $\mathit{nca}_{\mathcal{T}}(B(u), X) = X$. Moreover $r_X \in X$ thus r_X and w are adjacent in $\mathcal{G}$, therefore $|\mathit{except}(P_{up})| \leqslant 1$. Similarly if X is an ancestor of $B(v)$, then $|\mathit{except}(P_{down})| \leqslant 1$. Thus $|\mathit{except}(P_X)| \leqslant |\mathit{except}(P_{up})| + |\mathit{except}(P_{down})| \leqslant 3$. □

3 The Routing Scheme

Let us outline the routing scheme. Each node u of $\mathcal{G}$ contains in its address the information needed to route in all trees $\mathcal{S}_X$ such that X is an ancestor in $\mathcal{H}$ of $B(u)$ ($B(u)$ included). They are $O(\log n)$ such routing trees. To send a message from a source u to a destination v, we use a route through the tree $\mathcal{S}_X$ where $X = \mathit{nca}_{\mathcal{H}}(B(u), B(v))$. In fact, X is a separator between u and v in $\mathcal{G}$ (Proposition 2), and both u, v contain the information about routing in $\mathcal{S}_X$. So, the standard route is done along $\mathcal{S}_X$. However that makes harder the routing process is that, along the route between u and v in $\mathcal{S}_X$, one may traverse some nodes w such that $B(w)$ is not a descendant of X in the tree $\mathcal{H}$. As w knows only trees $\mathcal{S}_Y$ for all ancestors Y of $B(w)$ in $\mathcal{H}$, it turns out that w ignores the route in $\mathcal{S}_X$, whereas $\mathcal{S}_X$ spans w. Repeating recursively the same kind of routing from w to v may result of a very long detour from the distance between u to v. (Actually the result for the length of the routes would be a multiplicative stretch $s > 1$, and not a constant additive stretch as required.) Such a node w belongs to *except*(P_X) where P_X is the path from u to v in $\mathcal{S}_X$. As these nodes are at most 3 (Corollary 1), the solution consists to store in advance the right edge for all the nodes that are in *except*(P_X) in order to force them to follow the path in $\mathcal{S}_X$.

3.1 Description of the Labels

We assume that the outgoing edges of every node u of $\mathcal{G}$ are numbered arbitrarily by distinct integers, called *output port numbers*, and taken from $[1, \deg(u)]$. Our scheme associate to every node u of $\mathcal{G}$ two labels: its *address*, denoted by *address*(u), and a *local routing table*, denoted by *table*(u).

• The local routing table of u in $\mathcal{G}$, *table*(u), is set to *info*(u) (defined hereafter);

• The address of u in $\mathcal{G}$ is defined by *address*$(u) = \langle path\rangle\,(u)$, *id*$(u)$, *help*$(u)$, *info*$(u)$. Let h be the depth of $B(u)$ in $\mathcal{H}$, and $i \in \{0, \ldots, h\}$. Let X_i be the ancestor in $\mathcal{H}$ of $B(u)$ of depth i and let P_i be the path in S_{X_i} from u to r_{X_i}. The fields of *address*(u) are as follows:

<u>*path*</u>(u)*:* represents a sequence $(p_1, \ldots, p_h)$ of h integers. For every node X of $\mathcal{H}$, the edges between X and its children are labeled with distinct integers. *path*(u) is the sequence of edge labels encountered along the path from the root of $\mathcal{H}$ to $B(u)$. So, *path*(u) is an identifier of $B(u)$. If $B(u)$ is the root of $\mathcal{H}$, then *path*(u) is defined as the empty sequence. For convenience, *path*$(u)[j]$ denotes p_j, and *path*$(u)[1 \ldots j]$ denotes the sequence $(p_1, \ldots, p_j)$. Note that the path from the root of $\mathcal{H}$ to $nca_{\mathcal{H}}(B(u), B(v))$ is the longest common prefix between *path*(u) and *path*(v).

<u>*id*</u>(u)*:* is an integer taken from $[1, n]$, such that $id(u) \neq id(v)$ for all u, v of $\mathcal{G}$.

<u>*help*</u>(u)*:* is a table with $h+1$ entries. The i-th entry is $\langle |except(P_i)|, rescue(P_i)\rangle$. Where *rescue*$(P_i)$ is a table with $|except(P_i)|$ entries. Let w be the j-th node of *except*(P_i) (by Lemma 1, $j \leqslant 2$). The j-th entry of *rescue*(P_i) is:
$\langle id(w), down_{P_i}(w), up_{P_i}(w)\rangle$, where $down_{P_i}(w)$ and $up_{P_i}(w)$ are the output port numbers of the edges allowing to go in P_i from w to u, respectively from w to r_{X_i}

<u>*info*</u>(u)*:* is a table with $h + 1$ entries. The i-th entry of *info*(u) is $route_{X_i}(u)$, where $route_{X_i}(u)$ is a label depending on the tree $\mathcal{S}_{X_i}$ and on u such that the route from u to any node v in $\mathcal{S}_{X_i}$ can be determined from the labels $route_{X_i}(u)$ and $route_{X_i}(v)$ only. More precisely, for a suitable computable function f (so independent of the tree), $f(route_{X_i}(u), route_{X_i}(v))$, for every $v \neq u$, returns

the output port number of the first edge of the path from u to v in $\mathcal{S}_{X_i}$. An implementation of these labels is discussed in Lemma 4, in Paragraph 3.4.

3.2 The Routing Algorithm

Consider u, v two nodes of $\mathcal{G}$, u the sender and v the receiver. Procedure $\textsc{init}(u,v)$ is in charge of initializing the header attached to the message sent by u. This header, denoted by $H_{u,v}$, is $(h_X, \ell, (w_1,p_1), (w_2,p_2), (w_3,p_3))$ computed as follows:

Procedure $\textsc{init}(u,v)$:

1. Let X the nearest common ancestor in $\mathcal{H}$ between $B(u)$ and $B(v)$.
   Set h_X be the depth in $\mathcal{H}$ of X.
2. Set $\ell := \mathit{route}_X(v)$, i.e., $\mathit{info}(v)[h_X]$.
   Observe that this step just requires pointer assignment[3].
   Let P, P' be the paths in $\mathcal{S}_X$ from u to r_X, respectively from r_X to v.
3. For all nodes w of $\mathit{except}(P')$ (contained in $\mathit{rescue}(P')$, the second field of $\mathit{help}(u)[h_X]$), append $(\mathit{id}(w), \mathit{down}_{P'}(w))$ in $H_{u,v}$.
4. For all nodes w of $\mathit{except}(P) \setminus \mathit{except}(P')$, append $(\mathit{id}(w), \mathit{up}_P(w))$ in $H_{u,v}$.

Consider any node w of $\mathcal{G}$ that receives $H_{u,v} = (h_X, \ell, (w_1,p_1), (w_2,p_2), (w_3,p_3))$, the header computed from $\textsc{init}(u,v)$ (possibly, $w = u$). The output port number of the edge on which the message to v has to be sent from w is defined by Procedure $\textsc{send}(w, H_{u,v})$ described below. Observe that once $H_{u,v}$ is initialized by the sender, $H_{u,v}$ is never changed along the route.

Procedure $\textsc{send}(w, H_{u,v})$:

1. If $\mathit{id}(w) = w_i$, for $i = 1, 2$ or 3, then route on p_i.
2. Otherwise route in $\mathcal{S}_X$ using function $f(\mathit{route}_X(w), \ell)$.
   (Recall that $\ell = \mathit{route}_X(v)$. Moreover in this case, $w \notin \mathit{except}(P_i)$. Thus X is an ancestor in $\mathcal{H}$ of $B(w)$ and $\mathit{route}_X(w)$ is defined).

3.3 Correctness and Performances of the Routing Algorithm

We now give the correctness of the routing algorithm. Let $\rho(u,v)$ denote the length of the route produced by $\textsc{init}$ and $\textsc{send}$ from u to v. We denote by $d(u,v)$ the distance between u and v in $\mathcal{G}$. The correctness of our scheme is done proving that $\rho(u,v)$ is bounded. More precisely:

Lemma 2. *Let u, v be two nodes of $\mathcal{G}$, $\rho(u,v) \leqslant d(u,v) + 2$.*

[3] The copy of the header attached to the message before forwarding is not take in account in the time complexity of $\textsc{init}(u,v)$.

Proof. Let $P = x_1, \ldots, x_m$ be the path from $u = x_1$ to $v = x_m$ induced by the routing scheme. Clearly P is the path from u to v in $\mathcal{S}_X$, where X is the ball chosen by $\textsc{init}(u,v)$, i.e., $X = nca_{\mathcal{H}}(B(u), B(v))$. Thus $\rho(u,v) \leqslant d(u, r_X) + d(r_X, v)$.

Moreover, by Proposition 2, X is a separator between u and v in $\mathcal{G}$. Thus there exists $w \in X$ such that $d(u,w) + d(w,v) = d(u,v)$. As X is a clique, $d(u, r_X) \leqslant d(u,w) + 1$, and $d(r_X, v) \leqslant d(w,v) + 1$. This complete the proof. □

3.4 Implementation of the Scheme

We assume that the standard bitwise operations (like addition, xor, shift, etc.) on $O(\log n)$ bit words run in constant time.

Lemma 3. *For every u, path(u) and help(u) can be implemented by a binary string of $O(\log n)$ bits such that* $\textsc{init}(u,v)$ *runs in constant time.*

Proof. Step 1 of $\textsc{init}(u,v)$ needs h_X, the depth in $\mathcal{H}$ of $nca_{\mathcal{H}}(B(u), B(v))$. Although this problem cannot be solved for general trees with labels shorter than $\Theta(\log^2 n)$ bits [Pel00b], $O(\log n)$ bit labels suffice for $\mathcal{H}$ as is depth is bounded by $O(\log n)$. Assume $path(u) = (p_1, \ldots, p_h)$. To compact the labels, for every i, the integer $p_i \in \{1, 2, \ldots\}$ is chosen such that it labels the edge leading to the p_i-th heaviest child of the node identified by $path(u)[1 \ldots i-1]$. It is not difficult to check that such assignment satisfies: $\prod_{i=1}^{h} p_i \leqslant n$. Thus, using variable length encoding of each p_i, $path(u)$ can be stored on $O(\log n)$ bits. Using standard techniques, the length of the longest common prefix can be extracted in a constant number of bitwise operations (see [GKK$^+$00, pp.7-8] for a more precise implementation).

Step 2 of $\textsc{init}(u,v)$ is a pointer assignment. It assigns $info(u)[h_X]$ to ℓ, the second field of $H_{u,v}$. Steps 3 and 4 consist to extract $help(u)[h_X]$ and $help(v)[h_X]$, which are composed of a constant number of integers of $O(\log n)$ bits. Recall that by Corollary 1, $|except(P)| + |except(P')| \leqslant 3$. Then, tests and assignments on theses integers can be done in constant time. So, $\textsc{init}(u,v)$ runs in constant time. □

Lemma 4. *For every w, address(w), and table(w) can be implemented by a binary string of $O(\log^3 n / \log\log n)$ bits (polynomial-time constructible) such that* $\textsc{send}(w, H_{u,v})$ *runs in constant time. Moreover the length of $H_{u,v}$ is $O(\log^2 n / \log\log n)$ bits.*

Proof. Each node w of $\mathcal{G}$ stores $table(w)$, that is $info(w)$, and a finite set of algorithms (including the function f) representing a constant number of bits.

To implement the routing in the trees $\mathcal{S}_X$ we use the scheme presented in [FG01]. This scheme uses binary labels of length $O(\log^2 n / \log\log n)$, for an arbitrary port labeling of the tree. Moreover it takes a constant time to extract the output port number from two labels. Thus the length of $route_{X_i}(w)$ is $O(\log^2 n/\log\log n)$ bits for every trees $\mathcal{S}_{X_i}$ known by w. As $\mathcal{H}$ is of depth at most $\log n$, the length of $info(w)$, and thus of $table(w)$, is bounded by $O(\log^3 n/\log\log n)$ bits. The length of $address(w)$ is also bounded by $O(\log^3 n/\log\log n)$ bits.

Clearly, if $w = w_1, w_2$ or w_3, $\textsc{send}(w, H_{u,v})$ runs in constant time. Otherwise, let $\ell_1 = route_X(w)$ and $\ell_2 = route_X(v)$. By construction, ℓ_1 is $info(w)[h_X]$, and h_X, ℓ_2 are given by $H_{u,v}$. As computing $f(\ell_1, \ell_2)$ requires a constant time, cf. [FG01], it follows that $\textsc{send}(w, H_{u,v})$ runs in constant time.

Finally, the length of $H_{u,v}$ is $O(\log^2 n/\log\log n)$ bits since it contains only one tree routing label plus some constant terms on $O(\log n)$ bits. □

From all the previous lemmas it follows that:

Theorem 1. *There exists an additive* 2 *stretched loop-free routing scheme for chordal graphs using addresses and routing tables of size* $O(\log^3 n/\log\log n)$ *bits, and headers of size* $O(\log^2 n/\log\log n)$ *bits. Once computed by the sender, headers never change. Moreover, this scheme is polynomial-time constructible, and the routing function is computable in constant time for each node.*

References

[ABNLP89] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Compact distributed data structures for adaptive routing. In 21^{st} *Symposium on Theory of Computing (STOC)*, volume 2, pages 230–240, May 1989.

[ABNLP90] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Improved routing strategies with succinct tables. *Journal of Algorithms*, 11(3):307–341, September 1990.

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions. In 31^{th} *Symposium on Foundations of Computer Science (FOCS)*, pages 503–513. IEEE Computer Society Press, 1990.

[AP92] Baruch Awerbuch and David Peleg. Routing with polynomial communication-space trade-off. *SIAM Journal on Discrete Mathematics*, 5(2):151–162, May 1992.

[AR01] Stephen Alstrup and Theis Rauhe. Improved labeling scheme for ancestor queries. In 13^{th} *Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, January 2001. To appear.

[BCD99] Andreas Brandstädt, Victor Chepoi, and Feodor Dragan. Distance approximating trees for chordal and dually chordal graphs. *Journal of Algorithms*, 30:166–184, 1999.

[Die00] Reinhard Diestel. *Graph Theory (second edition)*, volume 173 of Graduate Texts in Mathematics. Springer, February 2000.

[Dou02] Yon Dourisboure. An additive stretched routing scheme for chordal graphs. In 28-*th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '02)*, June 2002. To appear.

[FG97] Pierre Fraigniaud and Cyril Gavoille. Universal routing schemes. *Journal of Distributed Computing*, 10:65–78, 1997.

[FG01] Pierre Fraigniaud and Cyril Gavoille. Routing in trees. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, 28th *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of Lecture Notes in Computer Science, pages 757–772. Springer, July 2001.

[FJ89] Greg N. Frederickson and Ravi Janardan. Efficient message routing in planar networks. *SIAM Journal on Computing*, 18(4):843–857, August 1989.

[FJ90] Greg N. Frederickson and Ravi Janardan. Space-efficient message routing in c-decomposable networks. *SIAM Journal on Computing*, 19(1):164–181, February 1990.

[Gav01] Cyril Gavoille. Routing in distributed networks: Overview and open problems. *ACM SIGACT News – Distributed Computing Column*, 32(1), March 2001. To appear.

[GG01] Cyril Gavoille and Marc Gengler. Space-efficiency of routing schemes of stretch factor three. *Journal of Parallel and Distributed Computing*, 61:679–687, 2001.

[GH99] Cyril Gavoille and Nicolas Hanusse. Compact routing tables for graphs of bounded genus. In Jiří Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, 26th *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of Lecture Notes in Computer Science, pages 351–360. Springer, July 1999.

[GKK$^+$00] Cyril Gavoille, Michal Katz, Nir A. Katz, Christophe Paul, and David Peleg. Approximate distance labeling schemes. Research Report RR-1250-00, LaBRI, University of Bordeaux, 351, cours de la Libération, 33405 Talence Cedex, France, December 2000.

[GP96] Cyril Gavoille and Stéphane Pérennès. Memory requirement for routing in distributed networks. In 15th *Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–133. ACM PRESS, May 1996.

[KM01] Haim Kaplan and Tova Milo. Short and simple labels for small distances and other functions. In 7th *International Workshop on Algorithms and Data Structures (WADS)*, volume 2125 of Lecture Notes in Computer Science, pages 32–40. Springer, August 2001.

[KMS02] Haim Kaplan, Tova Milo, and Ronen Shabo. A comparison of labeling schemes for ancestor queries. In 14th *Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, January 2002.

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays – Trees – Hypercubes*. Morgan Kaufmann, 1992.

[Lu02] Hsueh-I Lu. Improved comact routing tabels for planar networks via orderly spanning trees. In 8th *Annual International Computing & Combinatorics Conference (COCOON)*, volume Lectures Notes in Computer Science. Springer, August 2002.

[NN98] Lata Narayanan and Naomi Nishimura. Interval routing on k-trees. *Journal of Algorithms*, 26(2):325–369, February 1998.

[Pel00a] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications, 2000.

[Pel00b] David Peleg. Informative labeling schemes for graphs. In 25th *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 1893 of Lecture Notes in Computer Science, pages 579–588. Springer, August 2000.

[Pri97] Erich Prisner. Distance approximating spanning trees. In 14th *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1200 of Lecture Notes in Computer Science, pages 499–510. Springer, 1997.

[PS89] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[PU89] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, July 1989.

[RS86] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[Tho01] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In 42th *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, October 2001.

[TZ01a] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In 33rd *Annual ACM Symposium on Theory of Computing (STOC)*, pages 183–192, Hersonissos, Crete, Greece, July 2001.

[TZ01b] Mikkel Thorup and Uri Zwick. Compact routing schemes. In 13th *Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, Hersonissos, Crete, Greece, July 2001. ACM PRESS.

A Practical Multi-word Compare-and-Swap Operation

Timothy L. Harris, Keir Fraser, and Ian A. Pratt

University of Cambridge Computer Laboratory, Cambridge, UK
{tim.harris,keir.fraser,ian.pratt}@cl.cam.ac.uk

Abstract. Work on non-blocking data structures has proposed extending processor designs with a compare-and-swap primitive, CAS2, which acts on two arbitrary memory locations. Experience suggested that current operations, typically single-word compare-and-swap (CAS1), are not expressive enough to be used alone in an efficient manner. In this paper we build CAS2 from CAS1 and, in fact, build an arbitrary multi-word compare-and-swap (CASN). Our design requires only the primitives available on contemporary systems, reserves a small and constant amount of space in each word updated (either 0 or 2 bits) and permits non-overlapping updates to occur concurrently. This provides compelling evidence that current primitives are not only universal in the theoretical sense introduced by Herlihy, but are also universal in their use as foundations for practical algorithms. This provides a straightforward mechanism for deploying many of the interesting non-blocking data structures presented in the literature that have previously required CAS2.

1 Introduction

CASN is an operation for shared-memory systems that reads the contents of a series of locations, compares these against specified values and, if they all match, updates the locations with a further set of values. All this is performed atomically with respect to other CASN operations and specialized reads. The implementation of a non-blocking multi-word compare-and-swap operation has been the focus of many research papers [7,10,2,3,15,5]. As we will show none of these provides a solution that is practicable in terms of the operations it requires from the processor, its storage costs and the features it supplies.

This paper presents a new design that solves these problems. The solution is valuable because it finally allows many algorithms requiring CASN to be used in earnest (for example those from [11,5,4]). CASN is useful as a foundation for building concurrent data structures because it can update a set of locations between consistent states. Aside from its applicability, our solution is notable in that it considers the full implementation path of the algorithm. Previous work has often needed a series of abstractions to build strong primitives from those actually available – each layer adds costs, the sum of which places the algorithm beyond reasonable use.

We present our new design through a two-stage process: we develop a restricted form of CAS2 directly from CAS1 (Sect. 4) and we then show how to use that to implement CASN (Sect. 5). In Sect. 6 we discuss implementation problems such as memory management and the need for memory barrier operations. We evaluate the algorithm through experimental results on six processor families.

2 Background

Throughout this paper we assume a shared-memory model. We assume that an operation new allocates a fresh area of memory sufficient for a specified number of words. We take CAS1 as a primitive and assume initially that it – along with ordinary read and write operations – is implemented in a *linearizable* manner by the system (meaning that it appears to occur atomically at some point between its invocation and return). We assume that all memory accesses are of word-size and are to word-aligned addresses. As usual we define CAS1 as:

```
word_t CAS1(word_t *a, word_t o, word_t n)
{ old = *a;
  if (old == o) *a = n;
  return old;
}
```

We wish CASN to be *linearizable* so that it is easy to reason about its use. It should be *non-blocking*, meaning that some operation will complete if the system takes a large enough finite number of steps. This gives resilience against poor scheduler interactions (e.g. priority inversion). For scalability it is crucial that it is *disjoint-access-parallel*: operations on disjoint sets of locations should proceed in parallel. Finally, it should act on data structures with a natural and efficient representation. This means that reserving more than a few bits in each location is unreasonable. We would like to be able to use a built-in CAS1 operation for the case of a single-word update. It is usually necessary to use separate read and write operations on locations subject to update by CASN since disjoint-access-parallel designs place intermediate values in locations during their update.

2.1 Related Work

Herlihy's universal construction may form the basis of a CASN design, but it is not disjoint-access-parallel [7]. Neither is Greenwald's basic implementation using CAS2 [5], although he also shows how a further control word per word allows parallel updates. Israeli and Rappaport's design is disjoint-access-parallel [10], but each word must hold a processor 'ownership' field and the algorithm requires strong LL/SC operations. Those operations can be implemented over basic LL/SC or CAS1 by reserving further per-processor 'valid' bits in each word.

Anderson and Moir's wait-free CASN uses strong LL/SC single-word primitives [2]. It requires extensive auxiliary per-word structures. Moir subsequently developed a simpler *conditionally wait-free* design for CASN [15], meaning one

Table 1. `CASN` algorithms for a system with p processes, a machine word size of w bits, a maximum `CASN` width of n locations from a addresses, showing which algorithms are disjoint-access-parallel (*D-A parallel*) and which require support from the operating system kernel (*OS*)

	D-A parallel	Requires	Bits-per-word
[7]	No	`CAS1`	0
[5]	No	`CAS2`	0
[3]	No	`CAS1` + OS	0
[2]	Yes	Strong `LL/SC`	$p(w + l) + l$ where $l = \lg_2 p + \lg_2 a$
[15]	Yes	Strong `LL/SC`	$\lg_2 p + \lg_2 n$
[10]	Yes	Strong `LL/SC`	$\lg_2 p$
[3]	Yes	`CAS1` + OS	$1 + \lg_2 n + \lg_2 p$
[10]	Yes	`CAS1`	$p + \lg_2 p$
New	Yes	`CAS1`	0 or 2

that is not intrinsically wait-free but which, at key points after contention is detected, evaluates a user-supplied function to determine whether to retry.

Anderson *et al.* provide two further algorithms for priority-based schedulers [3]. One is only suitable for uniprocessors. The other is not disjoint-access-parallel. In their designs Anderson *et al.* use a restricted form of `CAS2` which is much the same as the `RDCSS` operation we define in Sect. 4. However, it requires priority-based scheduling and non-preemption guarantees for some code sequences.

Moir shows several ways to build strong `LL/SC` from `CAS1` or realistic `LL/SC` [14]. However, there are problems with each construction. The correctness of the first relies on sufficiently large counters reserved in each value not overflowing at certain points. The second design allows pointer-sized values to be stored by fragmenting them across words along with a header. This (at least) doubles the storage required. The third design provides single-word `LL/SC` operations without needing to avoid overflow based on an elaborate mechanism to control tag re-use – for example a single `SC` requires four operations on a tag-management queue. This design also requires a processor ID field to be reserved in every word, along with space for these bounded tags and a count field. None of these algorithms fits with our desire for a natural and efficient representation.

Table 1 summarizes the various existing `CASN` designs and contrasts them with our algorithm. For all except [2] and [15] the per-word overhead is reserved in each data location; in [2] and [15] it is separate.

3 Algorithmic Overview

As with most concurrent algorithms, the design of ours is rather intricate. We hope that a brief overview of the the algorithm's operation will aid readability. Central to it is the use of *descriptors.* These are data structures in which threads initiating some operation make available all of the information that others need to complete it – e.g. a CASN descriptor holds the addresses to be updated, the values expected to be found there, the new values to store and a status field indicating whether the CASN is still in progress.

A thread makes a descriptor active by placing a pointer to it into a location in shared memory. This is our non-blocking alternative to locking that location. Other threads seeing the descriptor pointer use the information in it to help the owning thread complete its operation and release the location. A CASN proceeds by placing pointers to its descriptor in each location being updated, checking that they hold the expected old values. If this succeeds for all the locations then each location is released, replacing the descriptor-pointers with the new values. If any location does not hold the requisite old value then the CASN is said to have failed and each location is restored to its old value. CASN therefore resembles an update made using two-phase locking, but employing descriptor pointers so that other threads accessing the locations do not block.

We decompose CASN into two layers. We first build a limited form of CAS2 (Sect. 4) that atomically introduces or removes descriptor-pointers conditional on a status field. From this we construct CASN (Sect. 5). Sect. 6 considers implementation issues and the management of the memory holding descriptors.

4 Double-Compare Single-Swap

We define RDCSS as a restricted form of CAS2 operating atomically as:

```
word_t RDCSS(word_t *a1, word_t o1, word_t *a2, word_t o2, word_t n2)
{ r = *a2;
  if ((r == o2) && (*a1 == o1)) *a2 = n2;
  return r;
}
```

This is *restricted* in that (*i*) only the location a2 can be subject to an update, (*ii*) the memory it acts on must be partitioned into a *control section* (within which a1 lies) and a *data section* (within which a2 lies), and (*iii*) the function returns the value from from a2 rather than an indication of success or failure. RDCSS may operate concurrently with (*i*) any access to the control section, (*ii*) reads from the data section using RDCSSRead, (*iii*) other invocations of RDCSS and (*iv*) updates to the data section using CAS1, subject to the constraint that such CAS1 may fail if an RDCSS operation is in progress on that location.

4.1 Design

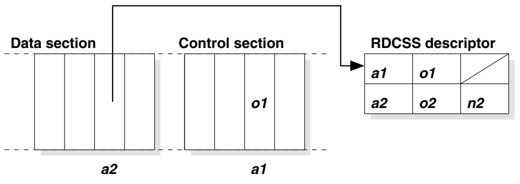
Figure 1 shows pseudo-code to implement RDCSS from CAS1. The *descriptor* passed to RDCSS contains five fields defining the proposed operation: the *control address* (a1), *expected value* (o1), *data address* (a2), *old value* (o2) and *new value* (n2). Descriptors are held outside the control and data sections and (aside from those introduced by RDCSS) values in the data section are distinct from pointers to descriptors. Each invocation uses a fresh descriptor, meaning one whose

```
word_t RDCSS (RDCSSDescriptor_t *d) {
  do {
    r = CAS1(d->a2, d->o2, d); /* C1 */
    if (IsDescriptor (r)) Complete(r); /* H1 */
  } while (IsDescriptor (r)); /* B1 */
  if (r == d->o2) Complete(d);
  return r;
}

word_t RDCSSRead (addr_t *addr) {
  do {
    r = *addr; /* R1 */
    if (IsDescriptor(r)) Complete(r); /* H2 */
  } while (IsDescriptor (r)); /* B2 */
 return r;
}

void Complete (RDCSSDescriptor_t *d) {
  v = *(d->a1); /* R2 */
  if (v==d->o1) CAS1(d->a2, d, d->n2); /* C2 */
    else CAS1(d->a2, d, d->o2); /* C3 */
}
```

Fig. 1. RDCSS pseudo-code implementation

address is (or acts as if it is) held only by the caller. A predicate `IsDescriptor` tests whether its parameter points to a descriptor – we discuss it in Sect. 6.

In outline `RDCSS` attempts a `CAS1` on the data address to change the old value into a pointer to the descriptor (`C1`). If successful, `Complete` finishes the operation: if the control address holds the expected value then the pointer is changed to the new value (`C2`), otherwise the old value is re-instated (`C3`). If a descriptor is found (`H1`, `H2`) then that `RDCSS` invocation is completed. A descriptor is 'active' when referenced from the data section, for example:



4.2 Correctness

We wish to establish that `RDCSS` and `RDCSSRead` provide linearizable non-blocking implementations. We proceeded by developing a model from the pseudo-code definitions and subjecting this to exhaustive tests using the Spin model checker [9]. Direct model checking is impracticable: the size of the shared memory, the number of active threads and the number of concurrent `RDCSS` invocations are unbounded. However, inspection of the algorithm lets us reduce the size of the state space to one which can be explored successfully:

– Each invocation of RDCSS can be considered separately. This surprising observation follows by examining the memory accesses. C1 is the only one to make a descriptor active and it succeeds at most once per descriptor (its return value causes loop B1 to terminate). Updates, C2 and C3 make descriptors inactive. Therefore each descriptor has at most one interval of time over which it is active, causing at most two updates – one to active it and one to deactivate it. Both updates are to the data address specified in the descriptor. Different RDCSS operations acting on the same location are thereby serialized by the order of their active periods, so we can consider them individually.
– We divide values in memory into equivalence classes. We classify the contents of the control address as either equal or not equal to the expected value. We need four classes for the data address: the old value, the new value, a pointer to the descriptor active on it and finally all other values.
– Although there may be an unbounded number of threads in the system, each is in one of a limited number of states defined by a point in the code of Fig. 1 and the values of local variables. We model the threads collectively as a set of pairs (p, m) where p represents a possible thread state and m is a boolean indicating whether at most one, or possibly more than one, thread is in that state. For example, this set initially contains one pair representing a single thread invoking RDCSS and multiple potential invocations of RDCSSRead.

We hypothesized that RDCSS can be linearized at the last execution of R2 for a descriptor that becomes active and otherwise at the last execution of C1. RDCSSRead would be linearized at its last execution of R1. From this abstraction we developed a Spin model in which global variables represent (i) the set of possible thread states (ii) the contents of the control and data addresses and (iii) the 'logical' contents of the data address, updated at the proposed linearization point of RDCSS and read at the proposed linearization point of RDCSSRead.

We model execution by defining a guarded statement for each thread state, enabled when that state is possible. Additional statements, always enabled, model operations that can operate concurrently with RDCSS– for example external updates to the value held at the control address. We used assertion statements to compare the logical and actual memory contents at the proposed linearization points. Spin accepts the resulting model without any assertion failures.

Showing non-blocking behaviour proceeds more directly: observe that each backward branch (B1, B2) is taken only if a descriptor-pointer was read from the data section and Complete invoked on that descriptor. Each descriptor-pointer is stored in the data section at most once (as above, at C1) and each invocation of Complete either removes the descriptor-pointer (if C2 or C3 succeeds) or observes it to have already been removed (if the attempted CAS1 fails). Therefore backward branches can only occur if system-wide progress has been made.

5 CASN Using RDCSS

We will now show how CASN can be implemented using RDCSS. As before a descriptor held in shared memory describes the operation. A *CASN-descriptor*

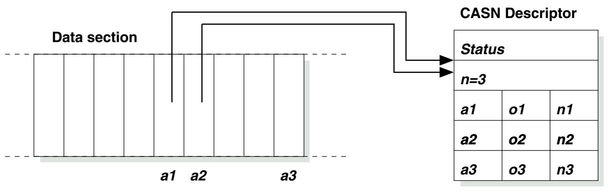
```
bool CASN (CASNDescriptor t *cd) {
  if (cd->status == UNDECIDED) { /* R4 */
phase 1: status = SUCCEEDED;
    for (i = 0; (i < cd->n) && (status == SUCCEEDED) ; i++) { /* L1 */
retry entry: entry = cd->entry[i];
            val = RDCSS (new RDCSSDescriptor t (&(cd->status), UNDECIDED,
                                                 entry->addr, entry->old, cd)); /* X1 */
            if (IsCASNDescriptor t (val)) {
              if (val != cd) {
                CASN (val); /* H3 */
                goto retry entry;
              }
            } else if (val != entry->old) status = FAILED;
          }
    CAS1 (&(cd->status), UNDECIDED, status); /* C4 */
  }
 phase 2: succeeded = (cd->status == SUCCEEDED);
   for (i = 0; i < cd->n; i ++)
     CAS1 (cd->entry[i].addr, cd,
         succeeded ? (cd->entry[i].new) : (cd->entry[i].old)); /* C5 */
   return succeeded;
}

word t CASNRead (addr t *addr) {
  do {
    r = RDCSSRead(addr); /* R5 */
    if (IsCASNDescriptor (r)) CASN (r); /* H4 */
  } while (IsCASNDescriptor (r)); /* B3 */
  return r;
}
```

Fig. 2. Two-phase CASN pseudo-code implementation using RDCSS at `X1`

contains a *status* field (holding `UNDECIDED`, `FAILED` or `SUCCEEDED`), a *count* (`n`) and then a series of `n` *update entries* each having a distinct *update address* (`a1`, . . .), an *old value* (`o1`, . . .) and a *new value* (`n1`, . . .).

The update addresses lie in the data section of memory and are held according to some total order agreed by all threads to guarantee non-blocking behaviour, e.g. sorted. The `CASN` descriptors themselves are held in the control section: the *status* field will be subject to comparison using `RDCSS`. As before, each invocation is made with a fresh descriptor. `CASN` may operate concurrently with (*i*) other invocations of `CASN` and (*ii*) reads from the data section using `CASNRead`.

Fig. 2 shows the two-phased pseudo-code for our `CASN` algorithm and for an associated `CASNRead` operation. The first phase attempts to introduce pointers from each update address to the descriptor. For example, after installing two such pointers the memory may be depicted:



If phase 1 encounters a pointer to another descriptor then it helps that operation before re-trying. At the end of phase 1, `C4` tries to set the status field to `SUCCEEDED` (if pointers were installed at each address) or `FAILED` (if some address did not contain the value expected). The second phase iterates over the update entries removing the pointers. A descriptor is *undecided* whenever its status field holds that value; otherwise it is *decided* and either *failed* or *succeeded*. The *logical contents* of a data location are (*i*) the value it holds if that is not a descriptor pointer, (*ii*) the old value for that location in an undecided or failed descriptor it points to, or (*iii*) the new value for that location in a succeeded descriptor.

`CASNRead` is structured in the same way as `RDCSSRead`: it retries the read operation until it does not encounter a descriptor pointer. Although we do not show them here, other kinds of read operation are also possible. One alternative is for `CASNRead` not to help other `CASN` invocations and instead to derive the logical contents of the location using the descriptor that it encounters. As Moir observed, the ability to read without helping can aid performance [15].

5.1 Correctness

We initially developed `CASN` in concert with a Spin model of its behaviour parameterized on the number of concurrent operations, the number of storage locations and the range of values that those locations could hold. The model maintained the actual contents of those locations (updated using our algorithm with `CAS1` as a primitive) and the logical contents (updated by an atomic step at the proposed linearization point). The largest configuration that could be checked exhaustively comprised 3 concurrent `CAS2` operations on up to 4 binary locations.

While invaluable in identifying problems with early designs, this approach also helped us develop our ideas of why the algorithm works in a general setting. In this section we show that `CASN` is linearizable, performing an atomic update to the logical contents of memory at the point the descriptor becomes decided.

Conceptually, the argument is simplified if you consider the memory locations referred to by a particular `CASN` descriptor and the updates that various threads make on those locations within the implementation of the `CASN` function. A descriptor's lifecycle can be split into a first *undecided* stage and a second *decided* stage, joined by `C4` which updates the descriptor's status field. We show that, aside from `C4`, all of the updates preserve the logical contents of memory and we then show that, when `C4` is executed, it atomically updates the logical contents of the locations on which the `CASN` is acting.

Firstly, we consider the descriptor lifecycle and the updates that can be made by threads operating on undecided descriptors:

Lemma 1. *Descriptor lifecycle is undecided → decided.* The only update to the status field is `C4` which specifies `UNDECIDED` as the expected old value and either `FAILED` or `SUCCEEDED` as the new value. The old and new values differ and so `C4` can succeed only once for each descriptor.

Lemma 2. *Threads operating on a descriptor in the undecided stage are in phase 1 of the algorithm.* To reach phase 2 a thread must either complete `C4` (the first to do so would make the descriptor *decided*) or observe the descriptor to be decided at `R4`. In each case a contradiction.

Lemma 3. *Threads operating on undecided descriptors preserve the logical contents of memory.* The only update made in phase 1 of the algorithm is `X1` which replaces the expected old value for a location with a pointer to a `CASN` descriptor specifying that same old value for that location. Hence the logical contents are preserved.

Secondly, we consider the linearization of `CASN` operations:

Lemma 4. *Linearization of failed* `CASN`. If `C4` sets the status `FAILED` then `X1` returned a non-descriptor value, not matching the expected old value. The time the unexpected value was read can be taken as the linearization point.

Lemma 5. *Linearization of successful* `CASN`. If `C4` sets the status `SUCCEEDED` then loop `L1` completed and so (*i*) for each update entry, either `X1` installed a descriptor-pointer or it found a pointer already in place, (*ii*) those values remain in place since no thread is yet in phase 2 (Lemma 2) and so (*iii*) `C4` changes the logical contents of all update addresses, forming the linearization point.

Thirdly, we consider the *decided* stage of a descriptor's lifecycle:

Lemma 6. *Threads with visible effects operating on decided descriptors are in phase 2 of the algorithm.* The only updates to shared storage outside phase 2 are `X1` and `C4`. Each tests the descriptor status for `UNDECIDED`.

Lemma 7. *Threads with visible effects operating on decided descriptors preserve the logical contents of memory.* The only update is `C5` and, if it succeeds, the computed value matches the logical contents of the location.

Lemma 8. *All descriptor-pointers are removed after one thread exits phase 2 of the algorithm.* During phase 2 a thread attempts `C5` for each update entry. Only `C5` changes a descriptor-pointer into a non-descriptor pointer: it will fail if (*i*) a descriptor-pointer was not installed at that location or (*ii*) another thread has already removed the descriptor-pointer by its own execution of `C5`.

Finally, a technical requirement of the restrictions `RDCSS` imposes (Sect. 4):

Lemma 9. `CAS1` *operations do not fail because of concurrent* `RDCSS`. `C4` acts on the control section so cannot encounter an `RDCSS` descriptor pointer. The only other consideration is between `C5` and `X1`. `C5` has a `CASN` descriptor pointer as its old value and `X1` has a non-descriptor value as its old value, so if `C5` fails because it encounters a pointer to an `RDCSS` descriptor then it would have failed anyway.

6 Implementation

The pseudo-code makes a number of assumptions about the underlying platform, and ignores four important problems: allocating and de-allocating descriptors, implementing the IsDescriptor predicates, the availability of CAS1 as an atomic primitive and the non-linearizability of the processor-supplied read/write and CAS1 primitives. We address those concerns in Sects 6.1–6.4.

6.1 Storage Management

For both RDCSS and CASN we assumed that fresh descriptors would be used on each invocation. We developed two techniques to reduce allocations. Firstly, we embed a group of RDCSS descriptors into each CASN descriptor to form a *combined descriptor*. Rather than holding the five RDCSS fields directly, these embedded descriptors contain a single constant reference to the enclosing CASN descriptor from which the RDCSS descriptor values are derived. Secondly, in each combined descriptor, we provide only one embedded RDCSS descriptor per thread. This still acts 'as if' fresh addresses are used because (*i*) the addresses operated on by a particular CASN are distinct from one another, and (*ii*) the RDCSS X1 will install a thread's embedded descriptor at most once at each address (if the RDCSS does install the pointer then either the loop advances to the next iteration or it terminates because the status field is no longer UNDECIDED).

We evaluated two ways of managing these combined descriptors. In the first we assumed garbage collection is already provided. In the second we introduced reference counting following Valois' CAS1-based design [17]. We used per-thread lists of free descriptors so that, without contention, a descriptor retains affinity for a particular thread. Manipulating reference counts and free lists formed around 10% of the execution time of an un-contended CASN. Although this scheme does not allow the storage that holds descriptors to be re-used for other non-reference-counted purposes, it is easy to imagine hybrids in which long-term shrinking uses garbage collection but short-term re-use employs counts. We are currently evaluating such combinations as well as Michael's SMR algorithm [13] and Herlihy *et al.*'s solution to the *Repeat Offender Problem* [8].

6.2 Descriptor Identification

The IsDescriptor and IsCASNDescriptor predicates must identify pointers to the descriptors used by RDCSS and CASN. If run-time type information is available then this could be exploited without further storage cost. Otherwise, the pointers themselves can be made distinct by non-zero low-order bits (as we did in previous work to indicate deleted items [6]). We need two bits to distinguish ordinary pointers, references to RDCSS descriptors and references to CASN descriptors.

We favour this second scheme because it is widely applicable and it avoids an additional memory access to obtain type information. An attractive hybrid scheme, which we have not yet evaluated, is to reserve a single bit to identify descriptor-pointers in general and then to use type information, or prescribed header values, to distinguish between the two kinds.

Table 2. CPU microseconds per successful CASN operation on a range of popular four-processor systems and CASN widths of 2, 4, 16 and 64 words

	IA-32				IA-64				Alpha				SPARC			
Type	2	4	16	64	2	4	16	64	2	4	16	64	2	4	16	64
HF	2.4	4.2	21	280	1.6	2.8	17	280	2.0	3.9	24	200	3.1	5.5	30	430
HF-RC	2.1	3.6	19	270	1.5	2.6	16	270	2.2	3.7	23	200	3.2	5.5	28	400
IR	4.0	6.3	26	340	3.4	4.4	19	300	4.5	6.4	31	490	5.4	8.7	44	570
MCS	4.8	7.2	22	84	5.6	8.2	24	92	7.1	7.4	17	63	10	16	61	250
MCS-FG	2.1	4.2	17	130	1.4	2.8	14	130	2.6	5.3	26	210	3.5	6.9	43	290

6.3 Atomic Hardware Primitives

Rather than implementing CAS1 directly, some processors provide the more expressive LL/SC (load-linked, store-conditional) operations. Unlike the *strong* LL/SC operations sometimes used in algorithms, these must form non-nesting pairs and SC can fail *spuriously* [7].

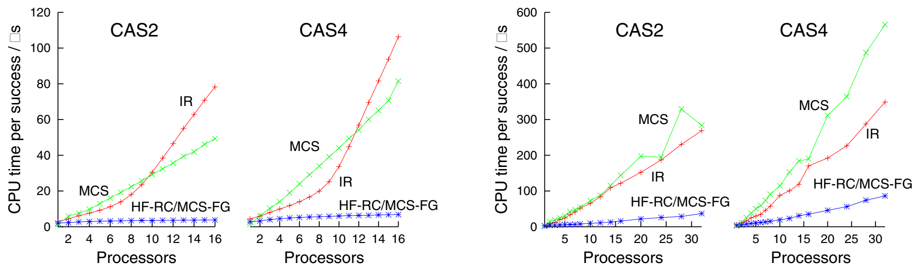
Where necessary we used a software version of CAS1, based on LL/SC, to generate the CASN results in this paper. Methods for building stronger primitives from LL/SC are well known: for example, the Alpha processor handbook shows how to use them to construct atomic single-word read-modify-write sequences such as CAS1. Such constructions, based on a simple loop that retries a LL/SC pair, are non-blocking under a guarantee that there are not infinitely many spurious failures during a single CAS1 operation.

6.4 Weak Memory Architectures

Finally, our pseudo-code assumes the sub-operations it uses are themselves linearizable. This is not true of modern systems – including all those used in Sect. 7. In general these systems provide cache-coherence, serializability of accesses to single words and total ordering between accesses from the same processor to the same location. Stronger ordering must be established using *barrier* instructions: operations before the barrier must commit before any later operation may be executed. Adve and Gharachorloo provide a tutorial on the subject [1].

7 Evaluation

Our benchmark is much the same as the *resource allocation* one used by Shavit and Touitou's which they argue is representative of highly concurrent queue and counter implementations [16]. A shared vector is initialized with distinct pointers. Each processor loops selecting a set of locations, reading their current values using CASNRead and attempting a CASN operation to permute the values between the locations. For a test using CASN operations of width n we divide the vector into n equal sized buckets and select one entry from each bucket. This benchmark enables a range of contention levels to be investigated by varying the



(a) 1–16 Power3-II processors (b) 1–32 MIPS processors

Fig. 3. CPU time per successful CASN operation for two large systems

concurrency, vector size, the width of CASN performed and whether padding is inserted between elements to place them on separate cache lines.

We implemented three non-blocking algorithms: ours assuming a garbage collector (HF) or using reference counting (HF-RC) and Israeli and Rappoport's CAS1-based design as the only practical alternative from Fig. 1 (IR). We also implemented two lock-based schemes using the queued spin-lock design of Mellor-Crummey and Scott [12], either with one lock to protect the entire vector (MCS) or with fine grain (i.e. per-entry) locks (MCS-FG).

Our measurements exclude initialisation. We start timing after all threads signal that they are executing and then run for two seconds. Results showing time per successful operation are calculated by dividing the total CPU time used (excluding initialisation) by the number of successful CASN operations. The CPU time and successful operations are summed across all threads. All results are presented to 2 significant figures. Although we do not analyse the costs of CASNRead operations in isolation, it is worth noting that a well-engineered implementation for any of the non-blocking algorithms adds only two operations to each read from a location that may be subject to CASN updates.

7.1 Small Systems

We ran our benchmark application using four threads on four-processor IA-32 Pentium-III, IA-64 Itanium, Alpha 21264 and SPARC Ultra-4 workgroup servers. We used a vector of 1024 elements without padding – in doing so we aim to produce a worst-case layout. The CPU requirements in μs per successful CASN are shown in Table 2 over a range of CASN widths on each system.

Our CASN algorithm performs universally better than the IR scheme. This is the case for every test we ran and follows intuition: the algorithms use the same helping strategy and, for an uncontended n way operation, HF performs $3n+1$ word-size CAS1 steps whereas IR performs $4n+4$ double-width steps. There is little difference in performance between HF and HF-RC: in low contention the cost of reference counting is balanced by the locality gained by re-use and as contention rises the main loop of the CASN dominates execution.

Table 3. Power3-II system (top) and MIPS R12000 system (bottom) : CPU microseconds per successful CASN operation with 16 threads *vs.* vector size and CASN width

	CAS2			CAS4			CAS8		
Type	256	1024	4096	256	1024	4096	256	1024	4096
HF-RC	4.7	3.8	3.6	11	6.8	5.9	39	17	12
IR	89	79	76	140	110	94	270	160	120
MCS	50	51	49	77	77	78	130	130	130
MCS-FG	4.2	3.6	3.6	11	7.6	6.9	35	18	14
DUMMY	2.8	2.7	2.7	4.8	4.4	4.3	9.4	8.0	7.7

	CAS2			CAS4			CAS8		
Type	256	1024	4096	256	1024	4096	256	1024	4096
HF-RC	21	16	15	55	33	28	180	99	58
IR	130	120	120	190	150	140	470	220	180
MCS	130	120	130	220	190	200	380	380	430
MCS-FG	18	14	12	38	29	24	115	69	54
DUMMY	14	11	12	26	23	21	62	44	40

Only the non-blocking algorithms experience CASN failures because the lock-based designs prevent updates between the old values being read and the CASN being attempted. On the 2-processor system the non-blocking algorithms exhibit indistinguishable success rates: 97-99% for widths of up to 8, 90% for 16, 70% for 32 and 50% for 64. On the 4-processor machines success rates of 90% and above are achieved for CASN widths of 2, 4 or 8.

7.2 Large Systems

We now examine larger systems: an IBM SP node of 16 Power3-II processors with uniform memory access and a ccNUMA Origin 2000 system with 64 MIPS R12000 processors. For these systems we inserted padding between vector elements to place each on its own cache line and eliminate *false sharing*. This improved the performance of all algorithms, particularly where contention was high. Furthermore, we maximized the performance of the MCS-FG algorithm by locating each vector element and its associated lock in the same cache line.

Fig. 3 shows how the CPU time per successful CASN varies with the number of processors used. In each case we examined CAS2 and CAS4 operating on a vector of 1024 pointers, corresponding to a minimum success rate of 92% on the Power3-II machine and 90% on the MIPS machine. We did not run experiments with HF because of its high per-processor memory demands in the absence of a garbage collector. In all graphs the lines for MCS-FG and HF-RC are coincident: we therefore present these results with suitably-labelled single lines.

Finally, we investigated the effects of varying the vector size. Table 3 shows μs per success on two 16-processor configurations. It is interesting to note the deleterious effect on performance caused by the increased contention occurring with smaller vector sizes, particularly for wider CASN. For example, on Power3-II the time per successful HF-RC CAS8 operation increases by 340% when the vector size reduces from 4096 to 256. With the larger vector 92% of operations succeed, but this drops to 52% for 256. The increased time per successful operation is due to the large amount of wasted work, but also to the heavy load placed on the machine's memory system as cache line ownership moves between CPUs.

Throughout all of our experiments we found that HF-RC outperforms the other non-blocking algorithm, IR, by a wide margin. Performance of HF-RC was closely comparable to that of the MCS-FG, the best lock-based algorithm. During these experiments we also recorded the ratio of minimum and maximum per-thread number of successful operations. Using this as a metric of fairness we found that HF-RC is at least as fair as MCS-FG.

7.3 Performance Bounds

We attempted to establish a best-case performance bound for CASN implementations on these systems. This was achieved by using a DUMMY function that performs a CAS1 on each of the N locations, but without any attempt to provide atomicity across the updates. For larger vector sizes (hence where contention is low) we found that the CPU time per operation used by DUMMY typically accounted for over 75% of that consumed by HF-RC. It is perhaps surprising that this simple operation takes such a large fraction of the time taken to complete the considerably more complex HF-RC and MCS-FG routines: For example, HF-RC requires three times as many CAS1 operations.

This discrepancy is because the cost of individual CAS1 operations vary considerably depending on whether the location's cache line is already held in an exclusive state in the local cache, or whether such a request must be issued to all other CPUs. When a CASN operation is started the locations that are to be updated are initially unlikely to be held locally since the vector is being actively shared with other CPUs. In contrast, the CAS1 operations that manipulate the CASN descriptor are likely to be local unless 'helping' has occurred.

We reason that any implementation of CASN will have to incur the cost of gaining exclusive ownership of the locations to be updated, and hence the performance of DUMMY provides a reasonable lower bound. From these results we conclude that substantial improvement on HF-RC is unlikely.

8 Conclusion

The results show that our algorithm achieves performance comparable with traditional blocking designs while maintaining the benefits of a non-blocking approach. By reserving only a small and constant amount of space (0 or 2 bits per location) we obtain key benefits over other non-blocking designs: those bits can often be held in storage that is otherwise unused in aligned pointer values, letting us build CASN from a single-word CAS1 operation and letting us use it on ordinary data structures. In contrast, Israeli and Rappoport's design requires per-processor reserved bits and further ownership information. Current processors provide a maximum of a 64-bit `CAS1` so if pointers are themselves 64-bits then this prevents the `IR` design from being used without substantial limitations on the data being held (or from being implementable at all much beyond the 32-processor results shown here). Moir's `LL/SC` constructions avoid per-processor reservations, but they are not amenable to use with natural pointer representations and their space reservations still grow with the level of concurrency [14]. Our work provides compelling evidence that `CAS1` is sufficient for practical implementations of concurrent non-blocking data structures, in contrast to other work that mandates `CAS2`.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.
2. J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proc. 14th PODC*, pp 184–193, Aug. 1995.
3. J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proc. 16th PODC*, pp 229–238, Aug. 1997.
4. D. L. Detlefs, C. H. Flood, A. T. Garthwaite, P. A. Martin, N. N. Shavit, and G. L. Steele Jr. Even better DCAS-based concurrent deques. In *Proc. 14th DISC*, LNCS 1914, pp 59–73, Oct. 2000.
5. M. Greenwald. *Non-blocking synchronization and system design.* PhD thesis, Stanford University, Aug. 1999.
6. T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proc. 15th DISC*, LNCS 2180, pp 300–314, Oct. 2001.
7. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM TOPLAS*, 15(5):745–770, Nov. 1993.
8. M. Herlihy, V. Luchangco, and M. Moir The repeat offender problem: a mechanism for supporting dynamic-sized, lock-free data structures. In *Proc. 16th DISC*, 2002.
9. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
10. A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. 13th PODC*, pp 151–160, Aug. 1994.
11. H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, Department of Computer Science, June 1991.
12. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, Feb. 1991.
13. M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proc. 21st PODC*, July 2002.
14. M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. 16th PODC*, pp 219–228, Aug. 1997.
15. M. Moir. Transparent support for wait-free transactions. In *Distributed Algorithms, 11th International Workshop*, LNCS 1320, pp 305–319, Sept. 1997.
16. N. N. Shavit and D. Touitou. Software transactional memory. In *Proc. 14th PODC*, pp 204–213, Aug. 1995.
17. J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th PODC* pp 214–222, Aug. 1995.

Failure Detection Sequencers: Necessary and Sufficient Information about Failures to Solve Predicate Detection

Felix C. Gärtner[1] and Stefan Pleisch[2]

[1] Department of Computer Science, Darmstadt University of Technology, D-64283 Darmstadt, Germany, felix@informatik.tu-darmstadt.de
[2] IBM Research, Zurich Research Laboratory CH-8803 Rüschlikon, Switzerland, spl@zurich.ibm.com

Abstract. This paper investigates the amount of information about failures needed to solve the predicate detection problem in asynchronous systems with crash failures. In particular, we show that predicate detection cannot be solved with traditional failure detectors, which are only functions of failures. In analogy to the definition of failure detectors, we define a *failure detection sequencer*, which can be regarded as a generalization of a failure detector. More specifically, our failure detection sequencer Σ outputs information about failures *and* about the final state of the crashed process. We show that Σ is necessary and sufficient to solve predicate detection. Moreover, Σ can be implemented in synchronous systems. Finally, we relate sequencers to perfect failure detectors and characterize the amount of knowledge about failures they additionally offer.

1 Introduction

Predicate detection in distributed settings is a well-understood problem and many techniques together with their detection semantics have been proposed [6]. Most of these techniques address predicate detection with the assumption that no faults occur in the system. However, it is desirable to also detect predicates which refer to the operational state of processes, e.g., predicates such as "$x_i = 1 \wedge crashed_i$", where $crashed_i$ is a predicate that is true iff (if and only if) process p_i has crashed. Since $x_i = 1$ might indicate the presence of a lock, the given predicate can be used to formalize special conditions such as "process p_i crashed while holding a lock", which is useful in the context of databases.

In the context of *crash* failures and the *consensus* problem, *failure detectors* have been devised to provide information about failures [3], but they offer "solely" information about failures. To detect general predicates such as the example predicate above, failure detection information needs to be combined with additional information about the internal state of a process. Indeed, while a failure detector may capture the predicate $crashed_i$, it gives no information about the value of x_i.

Ideally, a predicate detection algorithm never erroneously detects a predicate and does not miss any occurrence of the predicate in the underlying computation. As shown in [10], the quality of predicate detection critically depends on the quality of failure detection. This explains why work in [8,16,18] puts a restriction on the type of detectable predicates, or [9] weakens the semantics of predicate detection.

In previous work [10], we have investigated predicate detection in an asynchronous system with crash failures and found that it is impossible to solve predicate detection even with a very strong failure detector, the *perfect failure detector* [3]. In this paper, we show that predicate detection cannot be solved with *any* failure detector (as defined in [3]), no matter how strong it is. For example, consider a "real-time perfect" failure detector which makes no mistakes and flags the occurrence of a crash *immediately.* Even this failure detector is insufficient to solve predicate detection. The reason for this impossibility is that failure detectors are only functions of failures. Our proof is a generalization of previous impossibility proofs by the present authors [10] and by Charron-Bost, Guerraoui and Schiper [5]. We attempt to remedy the unpleasant situation caused by the result and (in analogy to the definition of failure detectors) define a *failure detection sequencer.* A failure detection sequencer is a generalization of a failure detector in that it conveys information that is a function of the failures *and* the current history of the system which is under observation. To solve predicate detection, we define a particular failure detection sequencer class Σ, that only gives one additional piece of information: for every crashed process it gives the latest state of the process before this one crashes. We show that Σ is necessary and sufficient to solve predicate detection and consequently is the "weakest failure detection sequencer" to solve predicate detection.

Although Σ is in a sense "stronger" than a perfect failure detector, it is still possible to implement Σ in synchronous systems. Moreover, we argue that using Σ it is possible to implement a *synchronizer* for asynchronous crash-affected systems which makes these systems equivalent to purely synchronous systems in terms of the solvability of *time-free* [5] problems. We finally argue that while perfect failure detectors can be viewed as capturing the synchrony of processes, failure detection sequencers in addition also capture the synchrony of communication.

After presenting the system model and defining the problem of predicate detection in Sections 2 and 3, we present our contributions in the following order: First, we show that it is impossible to achieve predicate detection with any failure detector in the sense of Chandra and Toueg [3] in Section 4. Section 5 introduces the failure detection sequencer abstraction and shows that a particular sequencer Σ is equivalent to predicate detection. In Section 6, we show how to implement Σ and then discuss the strength of Σ in Section 7. Finally, Section 8 concludes the paper. For the full proofs, the reader is referred to [11].

2 Model

We consider an asynchronous distributed system in which processes communicate via message passing. This means that no bounds on message transmission time nor on relative process speeds exist. Message delivery is reliable, i.e., a sent message is eventually delivered, no spurious messages are delivered, and messages are not altered. Processes can fail by crashing, i.e., they simply stop to execute steps. Crashed processes do not recover any more. Processes which do not crash are called *correct*.

2.1 Distributed Computations

A distributed system, called the *application system*, consists of a finite set Π of n processes $p_1, p_2, \ldots, p_n$ (called *application processes*). Each process p_i has a local state s_i (defined by the values assigned to its local variables) and performs atomic state transitions according to a local algorithm A. Such a state transition is also called an *event*. Sending and receiving a message also results in a state change. If a process p_i sends a message in state s_i which is received by process p_j resulting in state s_j, we say that s_i and s_j *correspond*.

We define a relation of *potential causality* (denoted "$\rightarrow$") [2] on local states as the transitive closure of the following two relations:

– $s \rightarrow s'$ if s and s' happen on the same process and s happens before s'.
– $s \rightarrow s'$ if s and s' happen on different processes and s and s' correspond.

A *local history of* p_i is an (infinite) sequence $s_1, s_2, \ldots$ of states. A *distributed computation* is defined as a set of local histories, one for every process. A *global state* of the computation is a vector $G = (s_1, s_2, \ldots, s_n)$ of local states, one for each process. Each local state identifies a point in the local history of a process and thus is equivalent to the set of all local states the process went through to reach its "current" local state. A global state G is *consistent* if the union of these sets (of all local states in G) is left-closed with respect to $\rightarrow$, i.e., if a state s is in this set and $s' \rightarrow s$, then s' must also be in this set. The set of all global states of a computation together with $\rightarrow$ define a lattice [14].

We assume the existence of a discrete global clock. Processes do not have access to this global clock; it is merely a fictional device to simplify presentation. Let $\mathcal{T}$ denote the range of output values of the global clock. For simplicity we think of $\mathcal{T}$ to be the set of natural numbers.

2.2 Failure Detectors

A *failure detector* is a device that can be queried at any time $t \in \mathcal{T}$ and outputs the set of processes that it suspects to have crashed at time t.

We adopt the formal definitions of failure detectors by Chandra and Toueg [3]. A *failure pattern* F is a mapping from $\mathcal{T}$ to the powerset of Π. The value of $F(t)$ specifies the set of application processes that have crashed until time $t \in \mathcal{T}$.

We define $\mathit{crashed}(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $\mathit{correct}(F) = \Pi \setminus \mathit{crashed}(F)$. A *failure detector history* H is a mapping from $\Pi \times \mathcal{T}$ to the powerset of Π. The value of $H(p,t)$ denotes the return value of the failure detector module for process p at time t, i.e., if p queries the failure detector at time t, $H(p,t)$ contains the set of processes suspected at that time.

A *failure detector* $\mathcal{D}$ maps a failure pattern F to a set of failure detector histories. The set of histories returned by the failure detector satisfy certain accuracy and completeness properties. A perfect failure detector satisfies *strong accuracy* and *strong completeness*:

– Strong accuracy: no process is suspected before it crashes. Formally:

$$\forall F . \forall H \in \mathcal{D}(F) . \forall t \in \mathcal{T} . \forall p, q \in \Pi \setminus F(t) . p \notin H(q,t)$$

– Strong completeness: a crashed process is eventually permanently suspected by every correct process. Formally:

$$\forall F . \forall H \in \mathcal{D}(F) . \exists t \in \mathcal{T} . \forall p \in \mathit{crashed}(F) . \forall q \in \mathit{correct}(F) . \forall t' \geq t . p \in H(q,t')$$

The set of all perfect failure detectors is denoted by $\mathcal{P}$. In the following, we will sometimes use the symbol $\mathcal{P}$ as a shorthand for any failure detector from $\mathcal{P}$.

2.3 Runs and Steps

Chandra and Toueg [3] define a computation (which they call a *run*) to be a tuple $R = (F, \mathcal{D}, I, S, T)$, where S is a sequence of algorithm steps and T is a sequence of increasing time values when these steps are taken. Steps are defined with respect to an algorithm which in turn is a collection of deterministic automata. We define a run in a slightly different but equivalent manner. Instead of S and T we use two functions: a *step function* S_s from $\mathcal{T}$ to the set of all algorithm steps, and a process function S_p from $\mathcal{T}$ to Π. Briefly spoken, $S_p(t)$ denotes the process which takes a step at time t and $S_s(t)$ identifies the step which was taken. Without loss of generality, we assume that at any instance of time at most one process takes a step. If no process takes a step at time t, both functions evaluate to $\perp$. A computation then is a tuple $R = (F, \mathcal{D}, I, S_s, S_p)$.

In predicate detection, which is defined in the following section, we wish to detect whether a predicate holds on the state of processes. We assume that the state resulting from an algorithm step contains enough information to infer the validity of the predicate. For instance, a predicate referring to the number of events that have occurred in the system requires that an event counter is part of the local state. In general, the state must allow to infer the events that have happened and are of interest for the predicate. We assume that there is at least a correspondence between the most recent step of a process and the state resulting from executing that step. In this paper, we use the terms *state* and *step* interchangeably.

3 Predicate Detection

To detect predicates in the application system (see Section 2.1), the application system is extended with a set Φ of m *monitor processes* $b_1, \ldots, b_m$. The sets Π and Φ together form the *observation system.*

While application processes may crash we assume, for simplicity, that monitor processes do not[1]. Crashes of application processes do not change the local state of the process[9]. However, the operational state of a process p_i is modeled by a virtual boolean variable $crashed_i$ on every monitor. The global state of the system together with the vector of *crashed* variables defines the *extended global state* of the system.

The task of the monitor processes is to observe the application processes and invoke a special primitive *detected* if the state of the system satisfies a certain predicate. A predicate ϕ is a boolean function on the extended global state of the application system. For example, the predicate $x_i = 2 \wedge crashed_i$ is true in a global state if the variable x_i of p_i equals 2 and p_i has crashed. We say that ϕ *holds* in a computation c iff there exists a consistent global state in c such that ϕ is true in that state.

In our version of predicate detection, monitors can observe multiple predicates simultaneously. More specifically, the predicate detection algorithm maintains a set S of currently active predicates. A special primitive $fork(\phi)$ can be used to add a predicate ϕ to this set. Whenever some $\phi \in S$ is found to hold in the computation, the predicate detection algorithm indicates this by pointing to ϕ, i.e., by calling $detected(\phi)$. Formally, detecting any $\phi \in S$ corresponds to detecting the disjunction of all such ϕ. This formulation of predicate detection has the important advantage of allowing us to increase the set of observed predicates at runtime. In other words, it does not matter when a predicate ϕ is added to S. Even if ϕ held "early" in the computation and $fork(\phi)$ is invoked very late (e.g., after hours), then still the algorithm must eventually invoke $detected(\phi)$ [2]. In this sense, our predicate detection concept is *adaptive* and thus slightly more general than other definitions of predicate detection (e.g., *perfect predicate detection* [10]). This is reflected in the following definition:

Definition 1 (predicate detection). *A predicate detection algorithm is a distributed algorithm running on the observation system with an input operation fork() and an output operation detected(). Using fork(ϕ) a new predicate can be added to an initially empty set S of predicates. The algorithm must satisfy the following properties:*

– *(Safety) If a monitor invokes detected(ϕ) then ϕ holds in the computation and $\phi \in S$.*

[1] Our results remain valid even if some monitors fail. The possibility results in Sect. 5 merely require that at least one monitor is correct.

[2] Later in Section 5.2 we show how this property can be implemented in asynchronous systems.

– *(Liveness) If $\phi \in S$ and ϕ holds in the computation, then eventually a monitor must invoke detected(ϕ).*

Our definition of predicate detection makes no reference to a specific implementation. Generally, one expects application processes to use causal broadcast [2] to consistently disseminate information about every local state change to all monitor processes. But this is not required by the specification. Furthermore, there is no indication how monitors keep track of the changes of *crashed* values of processes, i.e., we do not postulate the existence of a special type of failure detector in the specification. However, failure detection can be considered a special case of predicate detection on the extended state space where the predicate to be detected consists only of the *crashed* variables of processes. This highlights the close relationship between failure detection and predicate detection which is studied in the following sections.

Note that the meaning of "ϕ holds in the computation" corresponds to the detection modality *possibly*(ϕ) [7,13]. Detecting *possibly*(ϕ) involves constructing the entire computation lattice in the general case. The lattice represents all possible observations; hence, an observation is a path through the lattice. For simplicity we restrict our attention to *observer-independent predicates* [4]. For these types of predicates it is sufficient to construct a single observation, i.e., a single path through the lattice, to check whether ϕ holds in the observation. For example, stable predicates are observer-independent (a predicate is stable iff once it holds it holds forever). However, not all observer independent predicates are stable. For example, predicates which are local to a single process are observer-independent but may not be stable.

4 Impossibility of Predicate Detection in a Faulty Environment Using Failure Detectors

In this section we show that predicate detection cannot be solved with any failure detector in the sense of Chandra and Toueg [3]. This is because failure detectors are "functions of failures", i.e., the failure detector $\mathcal{D}$ is a function which maps a failure pattern F to some element of an arbitrary range $\mathcal{G}$. The proof is based on the assumption that apart from using failure detectors and (asynchronous) messages, no information can flow between processes. Messages sent by application processes to monitor processes for the sake of predicate detection are called *control messages.* The impossibility holds even if we assume that state changes on the application processes and the broadcast of control messages happen atomically.

Theorem 1. *It is impossible to solve predicate detection with any failure detector $\mathcal{D}$.*

Proof. The proof is by contradiction and thus assumes that there exists an algorithm A which solves predicate detection using some failure detector $\mathcal{D}$. Now consider a run $R_1 = (F, \mathcal{D}(F), I, S_s, S_p)$ in which p crashes without executing a single step and consider A detecting the predicate $\phi \equiv$"p crashed in initial state". Since A solves predicate detection, A will eventually detect ϕ (e.g., at time t_1). Now consider a run R_2 with the same failure pattern F, but different S_s and S_p where p executes a step s_1 before it crashes. Since A is correct and ϕ never holds, A will never detect ϕ. Since we assume that the only means of communication are control messages, A must receive the message m about the occurrence of s_1 in R_2 before t_1. If it would receive m later, A must act like in R_1 since it has no means to distinguish R_1 and R_2 (not even the failure detector can help here). But postulating that m is received before t_1 violates the asynchrony of communication, a contradiction.

In the next section we consider a way of circumventing the impossibility by extending the concept of a failure detector to a component which we call a *failure detection sequencer.*

5 Failure Detection Sequencers

The failure detector abstraction was introduced by Chandra and Toueg [3] to characterize different system models with respect to the solvability of problems in fault-tolerant computing. We take a similar approach and devise an oracle that encodes enough information to solve predicate detection in asynchronous systems with process crashes. As shown in the previous section, information about failures alone is not sufficient. Hence, our oracle also needs to provide information about the state of the process at the moment this process has crashed.

5.1 Definition

We now define a *failure detection sequencer Σ*, which consists of a set of passive modules, one for each monitor process. The sequencer can be queried by the monitor and returns an array of size n. The value at index i of the array is either $\perp$ or contains a predicate φ on the local state of process p_i. Informally spoken, the latter means that p_i has crashed and that its final state satisfied φ. The predicate φ may have different forms, e.g., indicate a unique sequence number of the step last performed by p_i. Let $\mathcal{A}$ denote the set of all possible array values, i.e., combinations of $\perp$ and local predicates, which can be returned by Σ. Formally, Σ is defined as follows:

A *sequencer history H_Σ* is a mapping from $\Phi \times \mathcal{T}$ to $\mathcal{A}$. The value of $H_\Sigma(m,t)$ indicates the return value of Σ at monitor b if it is queried at time t. If $H_\Sigma(m,t)[i] = s$, then b suspects p_i at time t to be in s ($s\ /\!\pm$). A *failure detection sequencer Σ* maps a failure pattern F, a step function S_s and a process function S_p to a set of sequencer histories.

Given a time t, the most recent step of a process p_i can be determined by inspecting S_s and S_p. If p_i has not executed any step, then the most recent step is denoted by ϵ. Formally, the *most recent step (mrs) of p_i at t given S_s and S_p* is s iff

$$mrs(p_i,t,S_s,S_p) : \exists t' \le t.(S_s(t') = s) \wedge (S_p(t') = p_i) \wedge (\forall t''.t' < t'' < t.S_p(t'') \neq p_i)$$

We require that the set of all possible sequencer histories H_Σ satisfies the following two properties:

– (Accuracy) No process is incorrectly suspected to be in state s. Formally:

$$\forall m.\forall p_i.\forall t.H_\Sigma(m,t)[i] = s \neq \perp \Rightarrow p_i \in F(t) \wedge (s = mrs(p_i, t, S_s, S_p))$$

– (Completeness) If p crashes, then eventually every monitor will permanently suspect p to be in some state. Formally:

$$\forall m.\forall p_i.\forall t.p \in F(t) \Rightarrow \exists t' \geq t.\forall t'' \geq t'.H_\Sigma(m,t'')[i] \neq \perp$$

Since the accuracy requirement has a conjunction in the consequent, it is possible to separate it into a step accuracy part and a crash accuracy part. Crash accuracy corresponds to *strong accuracy* of Chandra and Toueg [3] ("no process is suspected before it crashes"), while step accuracy would mean that a non-$\perp$ sequencer output for process p_i at time t always equals the state which p_i is in *at the same moment* (i.e., at time t). Clearly, this property has only trivial solutions (i.e., a solution which always outputs $\perp$) since asynchronous message passing does not allow instantaneous message delivery. However, the combination of step accuracy and crash accuracy makes sense again since crashes "freeze" the state of a process so that there is no danger of state change once the sequencer has suspected that process.

We have called the new device a "sequencer" because it allows to implement causal order on failure detection events. Indeed, using Σ it is possible to infer the state of a process at the moment it is suspected. This means that it is possible to know how many control messages are in transit. Hence, the "delivery" of the suspicion can be delayed until all causally preceding events have been delivered; Σ can be used to "sequence" crash notifications, as shown in the following section.

5.2 Equivalence to Predicate Detection

Now we investigate the power of failure detection sequencers and show that they are sufficient and necessary to solve predicate detection. First we consider sufficiency.

The idea of implementing predicate detection using Σ is to embed crash events consistently into the causal order $\rightarrow$ of events in a computation. For this purpose, the algorithm shown in Figure 1 uses *causal broadcast* [2] (using primitives *c-bcast* and *c-deliver*) to disseminate information about state changes to all monitors and to withhold issuing the crash occurrence when Σ suspects p_i after some state s until the state of p_i has indeed reached state s. This is done using a vector *def_crash*$[i]$ (for "deferred crash").

The adaptiveness (i.e., the ability to "restart" predicate detection via *fork*) of predicate detection is implemented by using a variable *history*, which stores the sequence of global states. Whenever a new predicate ϕ is issued using the *fork* command, the entire history is checked whether or not ϕ held in the past.

Theorem 2. *Predicate detection can be solved using Σ.*

Proof. Proving the safety property of predicate detection requires to show that every state constructed by the algorithm in Figure 1 is a consistent global state over the extended state space of the application. Similarly, the liveness property can be proven by showing that once ϕ holds in the application, eventually every monitor will construct a corresponding global state (this is where the completeness property of Σ is needed).

We now show that Σ is necessary to solve predicate detection. To do this we assume the existence of an abstract algorithm *PD* that solves predicate detection on a given computation. Then we give an algorithm that emulates the output vector of Σ using *PD*.

Similar to the predicate detection algorithm in Figure 1 we instruct application processes to send a control message to all monitors if a local event happens. These control messages are used to fork an increasing number of instances of *PD*. Initially, a single instance for the predicate "p_i crashed in initial state" is started for every process p_i. When the first control message (i, s) arrives, a new instance is *fork*ed for the predicate "p_i crashed in state s". This is done whenever a new control message arrives.

The *output* vector which simulates the output of Σ is initialized with $\perp$ values and only changed, if one of the instances of predicate detection terminates by issuing *detected*(ϕ). This indicates that a process crashed in some state. The algorithm reflects this by changing the corresponding entry in *output*. The change is permanent since the state in which a process crashes does not change anymore.

Theorem 3. *If predicate detection is solvable, then Σ can be implemented.*

Proof. The accuracy property of Σ follows directly from the safety property of the predicate detection algorithm. Exploiting the adaptiveness of predicate detection allows us to show the completeness property of Σ.

It is interesting to study the role of adaptiveness in the proof of Theorem 3. For example, consider a definition of predicate detection without adaptiveness, i.e., it is merely possible to start instances of *PD* at the beginning of the computation. Not knowing the way in which the computation will proceed, it is necessary to invoke an instance of predicate detection for *every* state a process may reach. Hence, non-adaptive predicate detection can be used to implement Σ as long as the state space of a process is finite. Adaptiveness allows to invoke instances of predicate detection "on demand". This means that — given infinite state space — while there is no bound on the number of calls to *fork*, the number of concurrent instances of predicate detection is always finite.

The following theorem is an immediate consequence of Theorems 2 and 3. It can be rephrased as showing that Σ is the "weakest failure detector" for solving predicate detection. The quotation marks are important, because from Theorem 1 we know that we should not call Σ a failure detector.

Theorem 4. *Solving predicate detection is equivalent to implementing Σ.*

```
On every application process $p_i$:
  ⟨whenever a state change from $s$ to $s'$ happens⟩ do
    $c\text{-}bcast$ $(i, s)$ to all monitors
On every monitor process $b_j$:
  variables:
    $state[1..n]$ of ⟨local state information⟩ init ⟨initial states of processes⟩
    $crashed[1..n]$ of boolean init false
    $def\_crashed[1..n]$ of $\{\perp\} \cup$ ⟨local state information⟩
    $history$ sequence of ⟨$(state, crashed)$⟩ init ⟨initial state⟩
    $S$ set of ⟨global predicates⟩ init $\emptyset$
  algorithm:
    do forever
      case ⟨next event⟩ of              {* three cases possible *}
      case 1: ⟨$(i, s)$ is c-delivered⟩
        $state[i] := s$
        $history := history \cdot (state, crashed)$
        if $\exists \phi \in S.\phi(state, crashed)$ then $detected(\phi)$ endif
        if $def\_crash[i] = state[i]$ then
          $crashed[i] :=$ true
          $history := history \cdot (state, crashed)$
          if $\exists \phi \in S.\phi(state, crashed)$ then $detected(\phi)$ endif
        endif
      case 2: ⟨$\Sigma$ suspects $p_i$ in $s$⟩
        if $state[i] = s$ then
          $crashed[i] :=$ true
          $history := history \cdot (state, crashed)$
          if $\exists \phi \in S.\phi(state, crashed)$ then $detected(\phi)$ endif
        else {* $state[i] \neq s$ *}
          $def\_crash[i] := s$
        endif
      case 3: ⟨$fork(\phi)$ is called⟩
        $S := S \cup \{\phi\}$
        if $\exists s_k \in history.\phi(s_k)$ then $detected(\phi)$ endif
      end {* case *}
    end {* do forever *}
```

Fig. 1. Solving predicate detection using Σ. The primitives *c-bcast* and *c-deliver* denote causal broadcast and causal message delivery, respectively. The operator $\cdot$ denotes concatenation of sequences. Furthermore, the choice of the case statement is supposed to happen in a fair manner (e.g., event handling is performed using first-come first-serve).

6 Implementing Σ

The sequencer Σ is a rather strong device and its strength makes it a highly desirable tool in crash-affected systems. Hence, the question naturally arises on how to implement Σ in "real" systems. First, consider *synchronous systems*, i.e., systems where bounds on message delivery delays and relative processing speeds exist. In synchronous systems, Σ can easily be implemented, for instance,

```
 1  On every application process $p_i$:
 2     ⟨whenever a state change $(s, s')$ happens⟩ **do**
 3        *c-bcast* $(i, s)$ **to all monitors**
 4  On every monitor process $b_j$:
 5     **variables:**
 6        $output[1..n]$ **of** $\{\perp\} \cup$ ⟨process state information⟩ **initially** $\perp$
 7     **algorithm:**
 8        **for all** $i \in \{1, \ldots, n\}$ **do begin**
 9           *fork*("$p_i$ crashed in initial state") **end**
10        **do forever**
11           ⟨wait until $(i, s)$ is c-delivered or $detected(\phi)$ is invoked⟩
12           **if** ⟨$(i, s)$ was c-delivered⟩ **then**
13              *fork*("$p_i$ crashed in state $s$")
14           **elsif** ⟨$detected(\phi)$ was called⟩ **then**
15              {* $\phi$ is "$p_i$ crashed in $s$" *}
16              $output[i] := s$
17           **endif**
18        **end** {* do forever *}
```

Fig. 2. Emulating Σ using a predicate detection algorithm. State changes and sending control messages on application processes is assumed to happen atomically. Event handling in line 11 is again performed in a fair manner, e.g., using first-come first-serve.

by the algorithm in Figure 3. This algorithm is a variant of the algorithm for implementing a perfect failure detector in synchronous systems presented by Tel [15]. With every local step, process p_i decrements a special timer variable r, one for every remote process. Upon message reception from process p_j ($j \neq i$), the timer is reset to the in-ital value δ, which is computed from the maximum message delivery delay and the difference in relative processing speed. If p_i fails to receive a message from p_j before the timer elapses, then p_j is suspected by p_i.

```
On every process $p_j$:
   **with** ⟨every step⟩ **FIFOsend** "alive in state $s$" *to all*
On every process $p_i$:
   **variables:**
      $D_i[1..n]$ **init** $(\perp, \ldots, \perp)$ {* sequencer output *}
      $r_i[1..n]$ **init** $(\delta, \ldots, \delta)$ {* timers *}
      $S_i[1..n]$ **init** ⟨initial states of $p_1, \ldots, p_n$⟩
   **algorithm:**
      **upon FIFOreceive** "alive in state $s$" from $p_j$ **do**
         ⟨reset timer $r_i[j]$ to $\delta$⟩
         $S_i[j] := s$
      **upon** ⟨expiry of timer $r_i[j]$⟩ **do**
         $D_i[j] := S_i[j]$
```

Fig. 3. Implementing Σ in synchronous systems. The value δ is a local timeout value computed from the global boundary on message delivery delay and relative processing speeds.

To see that the algorithm indeed implements Σ, we need to show that it satisfies the accuracy and completeness properties given in Section 5.1. The proof of the completeness property is the same as for perfect failure detectors. To see that the accuracy property is satisfied, consider the sequence of "alive" messages received by Σ. As these messages are sent and arrive in FIFO order[3], the failure detector also receives the correct sequence of state information. If p_j crashes, the final message received by p_i $(i \neq j)$ is also the final message which was sent by p_j. This implies that the state information given in that message is a true indication of the most recent step performed by p_j.

Theorem 5. *In a synchronous system the output of the algorithm in Figure 3 satisfies the accuracy and completeness conditions of Σ.*

Note that in general the entire state of the crashed process needs to be delivered by the failure detection sequencer. The efficiency of the implementation can be improved by taking into account the semantics of the predicate and the application (e.g., by delivering only references to certain predefined states).

Now consider a system without bounds on relative process speeds but bounded communication delays (i.e., asynchronous processes with *synchronous* communication). In such systems, Σ is implementable if any $\mathcal{D} \in \mathcal{P}$ is given. The algorithm is shown in Figure 4 and is similar to the one in Figure 3. Here the timing bound δ refers to the synchrony of the communication channels. Completeness is achieved through the completeness of $\mathcal{D}$ and the fact that the timer eventually runs out. Accuracy is satisfied because of the accuracy of $\mathcal{D}$, the FIFO property of messages (as above), and the fact that after expiry of the timer, no message can be in transit (bounded communication delays).

```
8   algorithm:
9     upon FIFOreceive "alive in state $s$" from $p_j$ do
10      $S_i[j] := s$
11    upon ⟨$\mathcal{D}$ suspects $p_j$⟩ do
12      ⟨reset timer $r_i[j]$ to $\delta$⟩
13    upon ⟨expiry of timer $r_i[j]$⟩ do
14      if ⟨$\mathcal{D}$ suspects $p_j$⟩ then
15        $D_i[j] := S_i[j]$
16      endif
```

Fig. 4. Implementing Σ using $\mathcal{D} \in \mathcal{P}$ and synchronous communication (lines 1 to 7 are the same as in Figure 3). The value δ is a local timeout value computed from the global boundary on message delivery delay.

[3] FIFO broadcasts are implementable in synchronous systems, as they can even be implemented in asynchronous systems [12].

Theorem 6. *In a system with asynchronous processes, synchronous communication, and any $\mathcal{D} \in \mathcal{P}$, the output of the algorithm in Figure 4 satisfies the accuracy and completeness conditions of Σ.*

We discuss the relationship between perfect failure detectors and Σ in more detail in the following section.

7 Discussion

We have shown that predicate detection cannot be solved with a perfect failure detector. However, it is solvable using failure detection sequencer Σ. In a sense this means that Σ is "stronger" than a perfect failure detector. Since both abstractions can be implemented in synchronous systems, a perfect failure detector seems to "loose" some information that a sequencer retains at its interface. In this context, two questions arise which we now discuss: (1) How can this difference in information be characterized, and (2) how much information (if any) does a sequencer loose compared to a fully synchronous system?

Regarding question (1), it seems that the synchrony of communication is the aspect which Σ (in contrast to perfect failure detectors) encapsulates. Consider for example an additional oracle Δ which can be queried whether or not the communication channel to a process p_j is empty. Both oracles, Δ and any $\mathcal{D} \in \mathcal{P}$, are incomparable, since they cannot emulate each other in asynchronous crash-affected systems. However, using Δ instead of the timeout mechanism in the algorithm of Figure 4 yields Σ. Hence, knowledge about the synchrony of communication channels is all that is needed to strengthen a perfect failure detector to Σ. Conversely, this information can be regarded as being "lost" at the interface of a perfect failure detector.

Regarding question (2), we now argue that Σ retains the full information present in synchronous systems. Using Σ, it is possible to implement a *synchronizer* [1] for asynchronous crash-affected systems. A synchronizer is a distributed algorithm that allows asynchronous processes to proceed in *rounds*. For this, the synchronizer generates a sequence of clock-pulses [1] at each process which separate the rounds. With every pulse, a process is allowed to send at most one message to one of its neighbors. The synchronizer ensures that all messages sent at the beginning of round r are received within round r. It also ensures that every correct process (i.e., a process that does not fail) participates in infinitely many rounds.

Since the failure detection sequencer makes it possible to identify the final message from a crashed process, it is possible to implement such a synchronizer just like in the fault-free case [15, p. 408]: At the beginning of round r, every surviving process sends exactly one message m_r to every other process (using reliable broadcast [12]). The application message which the process might send in round r is associated with this synchronizer message to form a single message. A process p_i waits until, for every other process p_j, either (a) m_r is received or (b) Σ suspects p_j. Note that in the latter case it is possible to distinguish the two cases where p_j crashed before or after sending the message m_r. (This distinction is not possible with a perfect failure detector.) Waiting for m_r is important in order to satisfy the specification of the synchronizer, as no other way exists to prevent application messages from round r to be received in round $r + 1$ or later.

The pulses generated by the synchronizer resemble a form of global logical time. Such a time is present in synchronous systems and so the synchronizer transforms the asynchronous system into a synchronous system, with the exception of global real time. In other words, time-free applications [5] perceive an asynchronous system augmented with Σ as equally strong as a synchronous system. Hence, Σ can be regarded as a form of failure detector which offers applications full synchrony without referring to a global clock.

As Σ is equivalent to a perfect failure detector which suspects a crashed process only if there are no messages in transit from that process, one can argue that Σ could be specified based on these properties. However, we feel that such a specification makes assumptions about the implementation of failure detectors, although most implementations will indeed be based on message reception (or the lack thereof). Our specification, in contrast, is only based on the state of the process and does not refer to a particular implementation.

8 Future Work

Many open issues for future work remain: For instance, can other protocols (like those used for solving *consensus*) exploit the additional power of failure detection sequencers to improve efficiency (e.g., in terms of message complexity)? Another interesting issue is whether other (possibly weaker) classes of failure detection sequencers are meaningful in asynchronous systems and offer more information than failure detectors. An obvious candidate would be an "eventually accurate" failure detection sequencer $\diamond\Sigma$. However, we conjecture that $\diamond\Sigma$ is equivalent to $\mathcal{P}$ with respect to the problems it allows to solve.

Acknowledgments. We wish to thank Rachid Guerraoui for his comments on an earlier version of this paper and the anonymous reviewers for their suggestions. The first author wishes to thank Ted Herman for many helpful discussions on the topic of failure detection.

References

1. B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, Oct. 1985.
2. K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb. 1995.
3. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.
4. B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan. 1995.

5. B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.
6. C. M. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.
7. R. Cooper and K. Marzullo. Consistent detection of global predicates. *ACM SIGPLAN Notices*, 26(12):167–174, Dec. 1991.
8. V. K. Garg and J. R. Mitchell. Distributed predicate detection in a faulty environment. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, 1998.
9. F. C. Gärtner and S. Kloppenburg. Consistent detection of global predicates under a weak fault assumption. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, pages 94–103, Nürnberg, Germany, Oct. 2000. IEEE Computer Society Press.
10. F. C. Gärtner and S. Pleisch. (Im)Possibilities of predicate detection in crash-affected systems. In *Proceedings of the 5th Workshop on Self-Stabilizing Systems (WSS2001)*, number 2194 in Lecture Notes in Computer Science, pages 98–113, Lisbon, Portugal, Oct. 2001. Springer-Verlag.
11. F. C. Gärtner and S. Pleisch. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. Research Report RZ 3438, IBM Research Laboratory, Zurich, 2002.
12. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.
13. K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG91)*, pages 254–272, 1991.
14. F. Mattern. Virtual time and global states of distributed systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, 1989. Elsevier Science Publishers. Reprinted on pages 123–133 in [17].
15. G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, second edition, 2000.
16. S. Venkatesan. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability*, 38(1):103–110, Apr. 1989.
17. Z. Yang and T. A. Marsland, editors. *Global States and Time in Distributed Systems.* IEEE Computer Society Press, 1994.
18. P. yu Li and B. McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC'93)*, pages 224–230, Nov. 1993.

Bounding Work *and* Communication in Robust Cooperative Computation*

Bogdan S. Chlebus[1,2], Leszek Gąsieniec[3],
Dariusz R. Kowalski[2,4], and Alex A. Shvartsman[5,6]

[1] Department of Computer Science and Engineering, University of Colorado at Denver, Campus Box 109, Denver, CO 80217-3364, USA.
[2] Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, Warszawa 02-097, Poland.
[3] Department of Computer Science, University of Liverpool, Chadwick Building, Peach Street, Liverpool L69 7ZF, UK.
[4] Département d'Informatique, Université du Québec à Hull, Hull, Québec J8X 3X7, Canada.
[5] Department of Computer Science and Engineering, University of Connecticut, 191 Auditorium Road, Unit 3155, Storrs, CT 06269, USA.
[6] Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square NE43-316, Cambridge, MA 02139, USA.

Abstract. We consider the *Do-All* problem: p failure-prone processors perform t similar and independent tasks. We assume that processors are synchronous, communicate by message passing, and are subject to crashes determined by an adaptive adversary restricted only by the upper bound f on the number of crashes. The performance of algorithms in this setting is normally measured in terms of *work* (total available processor steps) and *communication* (total number of point-to-point messages) complexity. We consider work and communication as comparable resources and we develop algorithms that have efficient *effort* defined as *work* + *communication*. We present a p-processor, t-task algorithm that has effort $\mathcal{O}(t+p^{1.77})$, against the unbounded adversary ($f < p$). This is the *first* algorithm that achieves *subquadratic* in p effort efficiency for unbounded adversary, or even for linearly-bounded adversary that crashes up to a constant fraction of the processors. We present another algorithm that has work $\mathcal{O}(t + p\log^2 p)$ against f-bounded adversaries such that $p - f = \Omega(p^b)$ for a constant b, $0 < b < 1$. We show how to achieve effort $\mathcal{O}(t + p\log^2 p)$ against a linearly-bounded adversary; this result is close to lower bound $\Omega(t + p\log p/\log\log p)$.

1 Introduction

We study algorithms for a distributed system in which tasks need to be done by failure-prone processors. We consider the abstract problem, called *Do-All*,

* The work of the third author was in part supported by the NSF-NATO Grant 0209588. The work of the fourth author was supported by a NSF Career Award and NSF Grants 9988304 and 0121277.

where the failure-prone processors communicate by message passing and where the tasks are similar, re-entrant, independent and idempotent. Such problems are encountered when a set of tasks (in particular tasks admitting "at least once" execution semantics) needs to be performed in a distributed system.

The *Do-All* problem in the synchronous message-passing setting was introduced by Dwork, Halpern and Waarts [12], whose algorithms' efficiency is measured in terms of *effort*, defined as the sum of *task-oriented work* and *communication.* Task-oriented work counts only the processing steps expended on performing tasks and it discounts any idling or waiting steps. Communication is measured as the message complexity. A more conservative measure of work is *available processor steps* [16] that accounts for all processing steps, including task-oriented work, idling and waiting. It is used in most solutions [4,10,14].

We also consider the synchronous failure-prone setting where t tasks need to be performed by p message-passing processors subject to up to f crash failures imposed by an adversary. The adversary is adaptive, meaning that the decisions about the failures of specific processors are made on-line. A processor can send a message to any subset of processors in one step. If a processor crashes during a step then any messages it sent are delivered to some arbitrary subset of the destinations. Crashed processors do not restart.

We measure algorithm performance using the *effort* approach following [12], but we use the *work* component of effort measured in terms of *available processor steps* following De Prisco, Mayer and Yung [10]. Thus we define *effort* to be $\mathcal{W} + \mathcal{M}$, where *work* $\mathcal{W}$ accounts for each step taken by the operational processors, includes idling, and where *communication* $\mathcal{M}$ is measured as the number of point-to-point messages. Prior research (except for [12]) did not consider effort, and instead focused on developing *Do-All* algorithms that are efficient in terms of work, and then dealt with communication efficiency as a secondary goal

Our results. Let t be the number of tasks and p the number of processors. We present a new way of structuring algorithms for the p-processor t-task *Do-All* problem that allows for both work *and* communication to be controlled in the presence of adaptive adversaries. We give a generic algorithm for performing work in the networks of crash-prone processors. This algorithm is parameterized for task-assignment rules and virtual communication graphs superimposed on the underlying network.

We call a deterministic algorithm *constructive* if all implementation details are specified. This is in contrast to algorithms that refer to combinatorial objects that are known to exist. Starting with a fully constructive algorithm, we trade constructiveness for better effort bounds. In some cases, either the communication patterns or the patterns of performing tasks are determined by referring to permutations with certain properties that are known to exist. In all cases the algorithms also solve *Do-All* when exposed to the unbounded adversary.

I. We present a deterministic constructive algorithm, called algorithm BAL, that uses a balancing task allocation policy (Section 5.1). This algorithm has work $\mathcal{W} = \mathcal{O}(t + p\sqrt{p+t}\ \log^{3/2} p)$ against adaptive f-bounded adversaries, for any $f < p$. We also show that the effort of this algorithm against linearly-bounded adversaries is $\mathcal{W}+\mathcal{M}=\mathcal{O}(t+p\sqrt{t+p}\ \log^{3/2}p)$. In this case each processor sends $\mathcal{O}(1)$ messages in each step.

II. We also develop a deterministic algorithm which is more efficient than algorithm BAL, but it is not constructive (Section 5.2). This algorithm, called algorithm DPAL, has work $\mathcal{W}=\mathcal{O}(t+p\log^2 p)$ against the adaptive f-bounded adversary, if $p-f=\Omega(p^b)$, for a constant b, $0<b<1$. When subjected to linearly-bounded adversaries, this algorithm has the best known effort of $\mathcal{W}+\mathcal{M}=\mathcal{O}(t+p\log^2 p)$. This effort of algorithm DPAL is close to the lower bound $\Omega(p\log p/\log\log p)$ on work (see [6]).

III. We ultimately develop a deterministic algorithm, called algorithm UAL, that is efficient against the unbounded adversary (Section 5.3). This algorithm is obtained by combining our algorithm DPAL with the algorithm of De Prisco, Mayer and Yung [10]. The effort of our algorithm UAL is $\mathcal{W}+\mathcal{M}=\mathcal{O}(t+p^{1.77})$. This is the only known algorithm with both work and communication subquadratic in p in the worst case.

Previous work. Dwork, Halpern and Waarts [12] were the first to study the *Do-All* problem; further research includes [4,6,7,10,14,15]. Paper [12] considers a variation of effort, defined as the sum of *task-oriented work* and *communication*. A single-coordinator algorithmic paradigm is common in prior work.

De Prisco, Mayer and Yung [10] were the first to use *available processor steps* as the work measure. They gave an algorithm with work $\mathcal{O}(t+(f+1)p)$ and communication $\mathcal{O}((f+1)p)$. Galil, Mayer and Yung [14] gave an algorithm with better communication cost of $\mathcal{O}(fp^{\epsilon}+\min\{f+1,\log p\}p)$, for any $\epsilon>0$. Chlebus, De Prisco and Shvartsman [4] developed algorithms based on aggressive coordination, where the number of coordinators grows exponentially following crashes. Their algorithms rely on *atomic* multicasts. One of their algorithms has work $\mathcal{O}((t+p\log p/\log\log p)\log f)$ and communication $\mathcal{O}(t+p\log p/\log\log p+fp)$; these bounds were improved in [15] for $f\le p/\log\log p$. Another algorithm in [4] incorporates restarted processors, and is the only known algorithm to do so efficiently, and it uses $\mathcal{O}((t+p\log p+f)\cdot\min\{\log p,\log f\})$ work and its message complexity is $\mathcal{O}(t+p\log p+fp)$. A summary of these results is given in Table 1.

Table 1 does not include work dealing with specialized adversaries and models of communication. Chlebus and Kowalski [6] studied *Do-All* with fail-stop failures in the presence of the weakly-adaptive linearly-bounded adversary. They developed a randomized algorithm with the expected effort $\mathcal{O}\big(t+p\big(1+\log^*\big(\frac{t\log p}{p}\big)\big)\big)$. This performance is provably better than that of any deterministic algorithm in such a setting, where work $\Omega(p\log p/\log\log p)$ has to be performed ([6,15]). Chlebus, Kowalski and Lingas [7] and Clementi, Monti and Silvestri [8] studied *Do-All* in the model where communication is performed over a multiple-access channel.

2 Model: Computation, Tasks, Failures, and Performance

We assume a system of p synchronous, message-passing processors with unique identifiers in the interval $[1..p]$. We refer to the processor with identifier v as

Table 1. Summary of previous results against the unbounded adversary.

Paper	Metrics	Complexity	Remarks
[12]	*Work (tasks)* *Messages*	$\mathcal{O}(t+p)$ $\mathcal{O}(p\sqrt{p})$	Task-oriented work (allows idling and waiting)
[10]	*Work* *Messages*	$\mathcal{O}(t+p^2)$ $\mathcal{O}(p^2)$	
[14]	*Work* *Messages*	$\mathcal{O}(t+p^2)$ $\mathcal{O}(fp^\epsilon + p\log p)$	
[4]	*Work* *Messages*	$\mathcal{O}(t\log f + p\frac{\log p \log f}{\log\log p})$ $\mathcal{O}(t + p\frac{\log p}{\log\log p} + fp)$	Using atomic broadcast for unbounded adversary, $f < p$
[4]	*Work* *Messages*	$\mathcal{O}((t\log f + p\log p\log f)$ $\mathcal{O}(t + p\log p + fp)$	Using atomic broadcast; results are restated here for less than p restarts

"processor v". The computation is structured in terms of synchronous constant-time steps. There are t tasks that need to be performed; these tasks have unique identifiers in the interval $[1..t]$. Each processor knows the values p and t. We place no restrictions on the relationship between p and t.

Communication. Processors communicate by message passing. We assume a complete network. A processor may send a message to any subset of processors during a step. A message sent in one step is delivered to its recipients by the start of the next step. We assume that no messages are lost or corrupted in transit, and we assume that the size of the messages is at most linear in $p + t$.

Failures. Processors fail by crashing. Non-faulty processors are called *operational*. We denote by f the maximum number of failures that may occur in any execution. We do not consider processor restarts (cf. [4]), and so f does not exceed p. Processors may crash during a multicast; in this case some arbitrary subset of the destinations may receive the message.

Tasks. Every processor knows all the tasks, and it can perform any task given its identifier. We assume that it takes exactly one step to perform a task. Tasks are *idempotent*, that is, they can be performed many times and concurrently. They are also *independent*, in the sense that any order of performance is allowed.

Adversaries. The occurrence of failures is governed by adversarial models. These models are characterized in quantitative and qualitative terms that respectively determine the magnitude of processor failures and the flexibility to select faulty processors by the adversary.

We call an adversary *size-bounded* if there is an upper bound on the number of faults; when an explicit bound f is given we call the adversary f*-bounded*. An adversary is called *linearly-bounded* if it is (cp)-bounded for some constant $0 < c < 1$. An adversary is called *unbounded* if at least one processor is infallible. An adversary is called *adaptive* if it decides on-line what processors to crash and when (possibly subject to the quantitative restrictions). All the adversaries we consider are adaptive, with a possible restriction of being size-bounded. Our goal is to design algorithms that are correct against the unbounded adversary and as efficient as possible against specific size-bounded adversaries.

Correctness. An algorithm solving the *Do-All* problem is considered to be correct if the following two conditions are satisfied:

1. Each task is performed by some processor eventually.
2. Each processor stops to work on the current instance of the problem eventually.

There are two ways in which a processor can satisfy requirement 2. A processor may voluntarily stop to work, we say that it *halts*, or it may be forced to do so by the adversary who crashes the processor. Processors may halt at different steps. Halted processors are considered operational. Halting is a property of algorithms and does not limit the adversary in choosing which f processors are prone to failures. An algorithm *terminates* at a given step if this is the first step by which each processor either halted or crashed.

Performance measures. We consider the following measures of performance: work complexity and communication complexity. *Work* $\mathcal{W}$ counts the total number of steps of processors, including idling, which is accrued till termination. That is, we count the *available processor steps* [16]. Halted processors continue contributing to work until the algorithm terminates. *Communication* $\mathcal{M}$ counts the total number of point-to-point messages sent. We define *effort* as $\mathcal{W} + \mathcal{M}$.

3 Communication and Graphs

Let $G = (V, E)$ be a graph, with V the set of nodes and E the set of edges. For an induced subgraph H ($H \subseteq V$) we define $N_G(H)$ to be the subset of V consisting of all the nodes in H and their neighbors in G. We also define G^k to be the k-th power of graph G, that is, $G^k = (V, E')$, where the edge $(u, v) \in E'$ iff there is a path between u and v in G of length at most k. The number of nodes in subgraph H is denoted by $|H|$. The specific families of graphs $G(p)$ that we use have a uniform upper bound on the degrees of nodes, and we denote by Δ the maximum node degree of each graph $G(p)$.

We interpret each processor as a node and each sender/receiver pair as an edge of a *communication graph* $G = G(p)$. The notation $G(p)$ denotes the initial topology of a communication graph on p processors. Each faulty or halted processor is removed causing the communication graph to evolve through a sequence of subgraphs until it vanishes altogether when the algorithm terminates.

Suppose we take snapshots of the communication graphs at certain timesteps s_i, let G_i be the subgraph of $G(p)$ induced by the sets V_i of processors still operational at step s_i. Sets V_i have these properties: $V_{i+1} \subseteq V_i$ and $|V_i| \geq p - f$. Note that progress is achieved if there is a least one "good" connected component H_i of G_i, which evolves suitably with time: (1) it is "large" to be able to perform many tasks concurrently, (2) "of small diameter" to have a small delay in interprocessor coordination, and finaly (3) $H_{i+1} \subseteq H_i$ to guarantee consistency of computation. We quantify the notion of being "good" as follows.

Let f be an upper bound on the number of failures. Let α, β and γ be positive constants. We say that a subgraph $H \subseteq G = G(p)$ is *compact* if $|H| \geq \alpha(p-f)$ and the diameter of H is not larger than $\beta \log p + \gamma$.

Graph $G(p)$ is said to satisfy Property $\mathcal{R}$ if there is a function P, which assigns subgraph $P(R) \subseteq G(p)$ to each subgraph $R \subseteq G(p)$ of size at least $p-f$, such that the following hold:

$\mathcal{R}.1$: $P(R) \subseteq R$. $\mathcal{R}.3$: The diameter of $P(R)$ is at most $\beta \log p + \gamma$.
$\mathcal{R}.2$: $|P(R)| \geq \alpha |R|$. $\mathcal{R}.4$: If $R_1 \subseteq R_2$ then $P(R_1) \subseteq P(R_2)$.

Suppose $\langle G_i \rangle$ is a sequence of subgraphs of $G(p)$ such that $G_{i+1} \subseteq G_i$ and $|G_i| \geq p-f$, for $i \geq 1$. We may use Property $\mathcal{R}$ to obtain a sequence of compact subgraphs $\langle H_i \rangle$ in $G(p)$, such that $H_{i+1} \subseteq H_i$. To this end it is sufficient to define $H_i = P(G_i) \subseteq G_i$.

We use the family $L(n)$ of regular expander graphs introduced by Lubotzky, Phillips and Sarnak [17]. These graphs are defined and can be constructed for each number n of the form $q(q^2-1)/2$, where q is a prime integer congruent to 1 modulo 4. The node degree Δ_0 can be any number such that $\Delta_0 - 1$ is a prime congruent to 1 modulo 4 and a quadratic nonresidue modulo q. It follows, from the properties of the distribution of prime numbers (see e.g. [9]), that Δ_0 can be selected to be a constant independent n and q such that $n = q(q^2-1)/2 = \Theta(p)$. Since for each p there is a number n with such properties which is $\Theta(p)$, we let each processor simulate $\mathcal{O}(1)$ nodes, and we henceforth assume that p is as required so that $L(p)$ can be constructed.

Upfal [20] showed that there is a function P' such that if R is a subgraph of $L(p)$ of size at least $\frac{71}{72} \cdot p$ then subgraph $P'(R)$ of R has size at least $|R|/6$ and diameter at most $30 \log_2 p$. (These constants in the case of a linear-size subgraphs can be improved, see [2,13].) We show that a similar fact holds for sublinear-size subgraphs as well if a better expansion of the underlying communication graph is available. To this end we use power $L(p)^k$, for sufficiently large k. Such graphs are also constructive and of bounded degree. Magnifying expansion by taking a power of an expander was already suggested by Margulis [18] and used for instance by Diks and Pelc [11] in the context of broadcasting and fault diagnosis.

Let $\lambda > 1$ be a constant witht he property that the inequality $N_{L(p)}(R) \geq \lambda |R|$ holds if R is a subgraph of $L(p)$ satisfying $|R| < 71p/72$. Such λ exists because there is a constant $c > 0$ for which the inequality $|N(R)| \geq (1 + c(1 - |R|/p))|R|$ holds for an arbitrary set R of nodes (see [1]).

Lemma 1. *For every $f < p$ there exists a positive integer ℓ such that graph $L(p)^\ell$ has* Property $\mathcal{R}$ *with $\alpha = 1/7$, $\beta = 30$ and $\gamma = 2$.*

Proof. If $p - f \geq 71p/72$ then we may take $\ell = 1$ and rely on the construction in [20]. Suppose that $f > p/72$. Take as k the smallest positive integer satisfying $(p-f)\lambda^k \geq 71p/72$, and let $\ell = 2k+1$. Let R be a subgraph of graph $L(p)^\ell$ of size at least $p-f$. Since $|N_{L(p)^k}(R)| \geq 71p/72$ holds, we have $|P'(N_{L(p)^k}(R))| \geq |N_{L(p)^k}(R)|/6$. Let A denote the subgraph $P'(N_{L(p)^k}(R)) \cap R$ and B the subgraph $P'(N_{L(p)^k}(R)) \setminus R$. Define $P(R)$ to be the subgraph $P(R) = N_{L(p)^k}(A \cup B) \cap R$.

Note that $P(R)$ is defined as a set of vertices in graph $L(p)^\ell$ and is treated as a subgraph of $L(p)^\ell$ induced by these vertices.

One can see that ($\mathcal{R}$.1) holds by the definitions of $P(R)$.

To prove ($\mathcal{R}$.2) we estimate $|P(R)|$. Suppose, to the contrary, that $|R \setminus P(R)| \geq 6|R|/7$. Then $|N_{L(p)^k}(R \setminus P(R))| \geq 6(p-f)\lambda^k/7 \geq 6{\cdot}71p/(7{\cdot}72) > 5p/6$, which implies that $N_{L(p)^k}(R \setminus P(R)) \cap N_{L(p)^k}(P'(N_{L(p)^k}(R))) \neq \emptyset$. This contradicts the fact that $R \setminus P(R)$ and $P(R)$ are disjoint.

Next we prove ($\mathcal{R}$.3). Let v_1 and v_2 be two vertices in subgraph $P(R)$ of $L(p)^\ell$. Consider two vertices v_1 and v_2 from $P(R)$. Suppose that they are from $P'(N_{L(p)^k}(R))$. There is a path between node v_1 and node v_2 in subgraph $P'(N_{L(p)^k}(R))$ of $L(p)$ of length at most $30 \log p$. Let $v_1 = w_1, w_2, \ldots, w_m = v_2$ be a sequence of consecutive nodes on this path. We show that there is a path between node v_1 and node v_2 of the same length in subgraph $P(R)$ of $L(p)^\ell$, details are omitted. The case when either v_1 or v_2 are not in $P'(N_{L(p)^k}(R))$ is straightforward, as both of them have neighbors in $P'(N_{L(p)^k}(R))$ in subgraph $L(n)^k$, by the inclusion $P(R) \subseteq N_{L(p)^k}(P'(N_{L(p)^k}(R)))$. It follows that the distance between them in the subgraph $P(R)$ of $L(p)^\ell$ is at most $2 + 30 \log p$.

To prove ($\mathcal{R}$.4) let us consider $R_1 \subseteq R_2$, both subgraphs of sizes at least $p - f$. For any subsets of nodes A and B, let A_1 and B_1 denote the subgraphs induced by A and B, respectively, in subgraph R_1. Similarly, let A_2 and B_2 denote the subgraphs induced by A and B, respectively, in subgraph R_2. It is clear that $N_{L(p)^k}(R_1) \subseteq N_{L(p)^k}(R_2)$. If $X \subseteq Y$ are sets of nodes such that $|X| \geq 71p/72$ then $P'(X) \subseteq P'(Y)$, see [20]. It follows that inclusions $A_1 \subseteq A_2$ and $B_1 \subseteq B_2 \cup (A_2 \setminus A_1)$ hold. We obtain the final inclusion $P(R_1) \subseteq P(R_2)$, which completes the proof. □

Lemma 1 has been used recently by Chlebus and Kowalski [5] in their work on fault-tolerant gossiping and consensus algorithms.

4 Generic Algorithm

A processor that has not halted nor crashed is called *active*. We say a processor is *busy* at a certain step if its local knowledge does not imply that all tasks have been done. Our algorithms have the property that if some tasks are still outstanding then all the operational processors are both active and busy: they all iterate the main phase of the algorithm, with the goal of having all the tasks performed.

Recall that a subgraph of the communication graph is compact if its size is at least $(n-t)/7$ and its diameter is at most $30 \log p + 2$, see Lemma 1. Let the *neighborhood* of a processor v be the subgraph containing each operational processor whose distance from v in the communication graph is at most $30 \log p + 2$. A processor is said to be *compact* if its neighborhood is such. These notions are dynamic, since the status operational/faulty of a processor my change over time, so the status of being compact may be lost at some point of computation.
Local state. Each processor v maintains a local state consisting of three ordered lists and additional variables. The list $\texttt{Tasks}_v$ contains the tasks assumed to

be outstanding, the list $\texttt{Processors}_v$ contains the identifiers of the processors assumed to be active, and list $\texttt{Busy}_v$ contains the identifiers of the processors assumed to be busy performing tasks. These lists are initially ordered by the identifiers of the respective items. The position of an item in a list is its *rank* in the list. Items may be removed from lists, which affects the ranks of the remaining items. Each item u on list $\texttt{Processors}_v$ has an associated *distance variable* that stores the distance from processor v to u in the local view of the communication graph.

Each processor v maintains a counter (of phases) in its variable $\texttt{Counter}_v$. Additionally, processor v maintains variable $\texttt{Done}_v$, which remains false until v learns that all tasks have been performed. Variable $\texttt{Selected\_Processor}_v$ is used to remember a processor. Variable $\texttt{Phase}_v$ contains the name of the current phase performed by the processor; this name is either `Main` or `Closing`.

Messages. A message sent by processor v includes its identifier and the following components of its state: $\texttt{Tasks}_v$, $\texttt{Processors}_v$, and $\texttt{Busy}_v$. Additionally, the signal `stop` may be included if needed.

Handling lists. When in the course of a computation an item needs to be selected from a list, then this is done using some `Selection_Rule`. In Section 5 we define selection rules to instantiate specific algorithms.

Each active processor v updates the lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$ and $\texttt{Busy}_v$ after receiving messages from its neighbors in the current communication graph. If an item is not on the copy of a list received from some neighbor then it is removed from the corresponding local list. If no message is received from a neighboring processor u then u is removed from the list $\texttt{Processors}_v$. If signal `stop` is received from processor u or if processor u was chosen as the $\texttt{Selected\_Processor}_v$ then u is removed from the list $\texttt{Busy}_v$.

Distances among processors. Processors estimate distances among them in the communication graph using their local views of the graph. The distances may change during the computation due to failures. In each phase the distance variables attached to the items in the lists `Processors` are updated: for each processor v, the minimum of the distance variables attached to v on all the received processors lists is taken and incremented by one.

We use abbreviation $g(p)$ for function $30 \log_2 p + 2$. During a phase each processor v estimates the size of its neighborhood. This is done by comparing the distance variables on the updated $\texttt{Processors}_v$ list with $g(p)$: if the value of the distance variable is not greater than $g(p)$ then the corresponding processor is included in the neighborhood of v. If the estimated size of the neighborhood of processor v is at least $(p - f)/7$ then v is said *to believe to be compact.*

Structure of the generic algorithm. The algorithm consists of the initialization and a loop that iterates through one of the two phases.

The phases are of two kinds: Main Phase and Closing Phase. Each phase consists of three stages: (1) receiving messages, (2) local computation, and (3) multicasting messages. The goal of the Main Phase is to have all the tasks completed. The Main Phase is given in Figure 1.

```
STAGE 1: Receive messages.
STAGE 2: Perform local computations:
  a. Update the lists $\texttt{Tasks}_v$, $\texttt{Processors}_v$ and $\texttt{Busy}_v$.
  b. Update the distance variables, attached to the processors in the list $\texttt{Processors}_v$,
     and estimate the size of the neighborhood.
  c. If some processor is in list $\texttt{Processors}_v$ and not in list $\texttt{Busy}_v$ then set variable
     $\texttt{Done}_v$ to true.
  d. If list $\texttt{Tasks}_v$ is nonempty then select a task from it by applying Selection_Rule,
     remove the task from list $\texttt{Tasks}_v$, perform the task, and increment $\texttt{Counter}_v$.
  e. If list $\texttt{Tasks}_v$ is empty then set $\texttt{Done}_v$ to true and switch $\texttt{Phase}_v$ to Closing.
  f. If either ($\texttt{Done}_v$ is true and $v$ does not believe to be compact) or ($\texttt{Counter}_v > t$)
     then halt.
STAGE 3: Multicast messages to these neighbors in the communication graph that are
     still in the list $\texttt{Processors}_v$.
```

Fig. 1. Generic algorithm: Main Phase. Code for processor v.

Once a processor detects an empty list Tasks, it switches to the Closing Phase. Now the goal is to have all the active processors informed that the tasks have already been completed. This is accomplished in a similar manner: informing a processor is treated as a "task." To perform a "task" of this kind, an item from the list Busy is selected, then removed from the list, and signal stop is sent to it. In the Closing Phase the list Tasks is no longer needed.

Lemma 2. *The generic algorithm is correct against the unbounded adversary.*

Lemma 3. *If an algorithm uses only the graph $L(p)^\ell$ for communication, where ℓ is minimal such that Lemma 1 holds, and if its work complexity is $\mathcal{W}$, then its communication cost is equal to $\mathcal{M} = \mathcal{O}(\mathcal{W} \cdot \big(p/(p-f)\big)^{2\log_\lambda \Delta_0})$.*

5 Specific Algorithms

In this section we instantiate several algorithms obtained from the generic algorithm of Section 4 by specifying the following: (1) the patterns used to select tasks to be performed at a step, and (2) the messaging patterns using communication graphs. The performance of each algorithm is analyzed against specific adaptive size-bounded adversaries. We bound the maximum number of failures using parameter f. This number in turn determines the exponent ℓ of the power of the graph $L(p)$ we need for communication in order to establish the specific performance results.

For the purpose of analysis of the performance of specific instantiations of the generic algorithm, we partition the computation into disjoint epochs: each *epoch* $\mathcal{E}_i$ is defined to consist of $g(p) = 30\log_2 p + 2$ consecutive phases. An epoch has sufficient duration for all processors in a compact subgraph to communicate.

Throughout the proofs, the notation G_i means the subgraph of communication graph $G(p)$ induced by the vertices that are active during the first phase of epoch $\mathcal{E}_i$. Note that $G_{i+1} \subseteq G_i$ for $i \geq 1$. We let H_i stand for a compact subgraph of G_i in G, such that $H_{i+1} \subseteq H_i$ for $i \geq 1$. We resort to Property $\mathcal{R}$ to define H_i.

In every epoch $\mathcal{E}_i$ there is a compact node in G_i. This fact, in terms of communication, means that in every step there are $\Omega(p-f)$ operational processors which can communicate with each other during the next $g(p)$ phases, provided all of them are still active. If a processor halts then its list `Busy` is empty and this information needs $g(p)$ phases to proliferate. It follows that there are $\Omega(p-f)$ processors, which stay active till the end of the computation, possibly with the exception of the last $g(p)$ steps, and can communicate with each other with the delay of at most $g(p)$ phases.

A specific deterministic algorithm and a given adversary determines graphs G_i induced by active nodes during the first phase of epoch $\mathcal{E}_i$. If the adversary crashes processor nodes depending on the random decisions to select items from their lists then graphs G_i depend on the execution. In this case graphs H_i also depend on the execution.

Let $T_{v,i}$ be the set of elements in $\texttt{Tasks}_v$ during the first phase of $\mathcal{E}_i$. We use the following notations: U_i stands for $\bigcup_{v \in H_i} T_{v,i}$, S_i for $\bigcap_{v \in H_i} T_{v,i}$, and $u_i = |U_i|$, $s_i = |S_i|$.

5.1 Balancing Algorithm BAL

Algorithm BAL is instantiated from the generic algorithm by defining the rule `Load_Balance` used to select items by load balancing. Let q be the number of elements in the list at hand, then processor v selects the item of rank $r(v)$, where the number $r(v)$ is uniquely determined by the inequalities

$$p(r(v)-1)/q < v < p(r(v)+1)/q \ . \tag{1}$$

We now analyse the performance of the algorithm.

Lemma 4. *If $u_i \geq 11p^2 g(p)$ then $u_i - u_{i+1} \geq |H_{i+1}| g(p)$.*

Lemma 5. *Let $u = u_{i_0}$. For $i > i_0$, if during epoch $\mathcal{E}_i$ all the processors in H_{i+1} are in the Main Phase and $u_i > \sqrt{\frac{u}{\log u}}$, then $u_i - u_{i+1}$ is either at least $\min\left\{\frac{1}{14p}\sqrt{\frac{u}{\log u}}\ , 1\right\} \cdot |H_{i+1}|$ or at least $u_i \cdot \sqrt{\frac{\log u}{u}}$.*

Theorem 1. *The Do-All problem can be solved by a constructive deterministic algorithm with work $\mathcal{W} = \mathcal{O}\big(t + p\sqrt{p+t}\ \log^{3/2} p\big)$ against adaptive f-bounded adversaries, for any number of faults $f < p$.*

Proof. If the number of active processors in H_i decreases by more than twice in $\mathcal{E}_i$ then $\mathcal{E}_i$ is called *stormy*, otherwise it is *peaceful*. Since the total work accrued during stormy epochs is $\mathcal{O}(p \log p)$, we may assume in the rest of the proof that all the epochs are peaceful. Let H be a set of processors such that all its elements remain active during at least one full (peaceful) epoch, starting from some $\mathcal{E}_m$, where $H \subseteq H_m$, and $|H| \geq |H_m|/2$.

Set H may survive for a number of epochs, possibly till the termination. We can partition all the numbers of epochs into five disjoint intervals I_j, $1 \leq j \leq 5$, in such a way that if $a \in I_{j_1}$ and $b \in I_{j_2}$, for $j_1 < j_2$, then $a < b$. The partitioning is as follows. Interval I_1 contains those i for which $u_i \geq 11p^2 g(p)$. Interval I_2 contains those i for which both $u_i < 11p^2 g(p)$ and $s_i \geq 1$ hold. Hence if $i \in I_2$ then no local knowledge of a processor in H implies that all the tasks have been done. Interval I_3 contains just a single number i of the first epoch $\mathcal{E}_i$ following the epochs with numbers in I_2. During $\mathcal{E}_i$ all the processors in H learn that all the tasks have been performed. Number i is in interval I_4 if H is still compact during the first phase of epoch $\mathcal{E}_i$. Finally interval I_5 contains the remaining numbers of epochs.

We estimate the sizes of intervals I_j. By Lemma 4, during epochs in I_1 all processors in H perform different tasks, hence $|I_1| = \mathcal{O}(t/(|H|g(p)))$. During I_2 all processors from H are in the Main Phase. Let i_0 be the minimal element in I_2. Let $x = \sqrt{u/\log u}$, where $u = u_{i_0}$. We further partition I_2 into $I_2 = I_2^1 \cup I_2^2 \cup I_2^3$ as follows. Number $i \in I_2^1$ iff $u_i \leq x$. Let the remaining numbers of epochs in I_2 satisfy the following: if $i \in I_2^2$ then $u_i - u_{i+1} \geq u_i/x$, and if $i \in I_2^3$ then

$$u_i - u_{i+1} \geq \min\left\{\frac{x|H|}{14p}, |H|\right\} .$$

Lemma 5 guarantees that such a partition of I_2 is possible. We estimate the sizes of components of I_2: $|I_2^1| = \mathcal{O}(x/g(p)) = \mathcal{O}(\sqrt{t/\log t}/g(p))$. If $i \in I_2^2$ then $u_{i+1} \leq u_i(1 - 1/x)$ so the size of I_2^2 is $\mathcal{O}(\sqrt{u \log u}) = \mathcal{O}(\sqrt{t \log t})$, because $u(1-1/x)^{x \log u} = \mathcal{O}(1)$. The size of I_2^3 is

$$\begin{aligned}&\mathcal{O}(\max(up/(|H|x), u/|H|)) = \mathcal{O}(\sqrt{u \log u} \cdot p/|H| + u/|H|)\\ &= \mathcal{O}((\sqrt{t \log t} + \sqrt{ug(p)}) \cdot p/|H|) = \mathcal{O}((\sqrt{t(g(p) + \log t)}) \cdot p/|H|)\end{aligned}$$

because $u < 11p^2 g(p)$. It follows that $|I_2| = \mathcal{O}((\sqrt{t(g(p) + \log t)}) \cdot p/|H|)$.

The cases of I_4 and I_5 are omitted. The whole work contributed by H is $\mathcal{O}\big(|H|g(p) \sum_{j=1}^{5} |I_j|\big)$, which yields the bound

$$\mathcal{O}\left(t + p\sqrt{(p+t)(\log p + \log t)} \log p\right) \tag{2}$$

by our estimates. This bound was proved under the assumption that the contributing processors are in the same set H. In reality, set H may evolve through sets that get smaller at certain epochs. Then, as the next set H takes over, we are still within the bound (2) because it does not depend on the size of H. □

Corollary 1. *The Do-All problem can be solved by a constructive deterministic algorithm with effort* $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p\sqrt{t+p}\log^{3/2} p)$ *against adaptive linearly-bounded adversaries.*

Proof. The algorithm in Theorem 1 uses a constant-degree network for communication, and each processor sends $\mathcal{O}(1)$ messages in each step, by Lemma 3. □

5.2 Permutation Algorithms RPAL and DPAL

We now explore another selection paradigm where processors choose items from lists in the order given by their private permutations. We consider randomized algorithm RPAL and a scheme of deterministic algorithms DPAL parametrized by families of permutations.

The new selection rule `Select_According_To_Permutations` operates as follows. Suppose each processor v has two private permutations, π_1 over $[1..p]$ and π_2 over $[1..11p^2 g(p)]$. Processor v starts by permuting the list $\texttt{Busy}_v$ according to π_1. If it needs to select an item from $\texttt{Busy}_v$ then it takes the first one in the list. Tasks are selected the same way as in rule `Load_Balance` as long as the size of $\texttt{Tasks}_v$ is greater than $11p^2 g(p)$. As soon as exactly $11p^2 g(p)$ items remain in $\texttt{Tasks}_v$, processor v permutes the list $\texttt{Tasks}_v$ according to π_2. Then, at any time when processor v needs to select an item from $\texttt{Tasks}_v$, it simply takes the first one in the list.

To specify the permutations required by the selection rule, we consider two alternatives. The randomized algorithm RPAL starts by each processor selecting two permutations at random, independently among all the processors. The deterministic algorithm DPAL equips each processor with a fixed pair of permutations. Algorithm DPAL is more efficient than the deterministic algorithm BAL, however the implementation of BAL is immediate, whereas algorithm DPAL is not constructive. Once the permutation is selected in an execution of algorithm RPAL, the future behavior of the adversary is determined by a *failure pattern* F consisting of a subset R of processors of size f, and for each processor $v \in R$ the time step during which it fails. Let H denote a compact subgraph $H \subseteq G$ of size $\Omega(p - f)$. We let $\mathcal{F}_H$ denote the set of all the failure patterns such that processors in H do not crash.

Lemma 6. *Suppose the adversary can select only the failure patterns from* $\mathcal{F}_H$*. If* $u_i \geq a$ *then the inequality* $u_i - u_{i+1} \geq (1/4)\min\{a, |H|g(p)\}$ *holds with the probability at least* $1 - \exp(-\Omega(|H|\log p))$.

Lemma 7. *If the adversary can select only the failure patterns from* $\mathcal{F}_H$ *then the number of epochs in an execution of* RPAL *is* $\mathcal{O}((t+p)/(|H|g(p)) + \log p)$ *with the probability at least* $1 - \exp(-\Omega(|H|\log p))$.

Theorem 2. *There is a deterministic algorithm that solves the Do-All problem with work* $\mathcal{W} = \mathcal{O}\big(t + p\log^2 p\big)$ *against the adaptive* f*-bounded adversary, if* $p - f = \Omega(p^b)$*, for a constant* $b > 0$.

Proof. Consider an execution of algorithm RPAL. Let $H_0 = G, H_1, \ldots, H_m$, for $m = \lceil \log \frac{p}{p-f} \rceil$, be a sequence of subgraphs such that $H_{j+1} \subseteq H_j$ and $|H_{j+1}| = |H_j|/2$ hold, and additionally, at every step of computation of algorithm RPAL the set H of operational processors satisfies the condition $H_j \subseteq H \subseteq H_{j-1}$, for some j where $1 \leq j \leq m$. Such a subsequence H_i depends on the random permutations chosen randomly at the start of RPAL.

Consider a set H, such that $H_j \subseteq H \subseteq H_{j-1}$ for some $j = 1, \ldots, m$, and H remains operational after phase l, while after phase $l-1$ the set of operational processors included set H_{j-1}. The sets $H_{j+1}, \ldots, H_m$ cannot be determined by step l. For a fixed set H we can estimate the number of possible components of size $|H_j| \geq |H|/2$ as follows:

$$\binom{|H|}{|H_j|} \leq \binom{|H|}{|H|/2} \leq 2^{|H| \cdot h(1/2)} \leq 2^{|H_{j-1}| \cdot h(1/2)} ,$$

where $h(x)$ is the binary entropy function. The fraction of families of permutations that result in only $\mathcal{O}((t+p)/(|H|g(p)) + \log p) = \mathcal{O}((t+p)/(|H_j|g(p)) + \log p)$ epochs of algorithm RPAL, while it is is executed, is at least

$$1 - 2^{|H_{j-1}|h(1/2)} \exp(-\Omega(|H|g(p))) = 1 - \exp(-\Omega(|H|g(p)))$$

by Lemma 7. Hence the part of computation during which the set of operational processors includes H_j and is included in H_{j-1} takes at most $\mathcal{O}((t+p)/(|H_j|g(p)) + \log p)$ epochs, for a fraction of at least $1 - \exp(-\Omega(|H|g(p)))$ of the families of initial permutations. It follows that a fraction of the number of families of permutations that satisfy the claim of the theorem is at least

$$1 - \sum_{j=1}^{m} \exp(-b|H_j|g(p)) \geq 1 - \sum_{j=1}^{m} \exp(-b(p-f)g(p)) \geq 1 - \exp(-c(p-f)g(p))$$

for some constants $b, c > 0$. Algorithm DPAL using one of such families performs work at most $\max_{j \leq m} \{d\big((t+p)/(|H_j|g(p)) + \log p\big) \cdot |H_{j-1}|g(p)\} = \mathcal{O}(t + p\log^2 p)$ for some constant $d > 0$, by an argument similar to that used in the proof of Theorem 1. □

Corollary 2. *There is a deterministic algorithm that solves Do-All with effort* $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p\log^2 p)$ *against adaptive linearly-bounded adversaries.*

5.3 Hybrid Algorithm UAL

We now present algorithm UAL that is efficient against the unbounded adversary. The algorithms consists of two parts: *Part 1* and *Part 2*. *Part 1* is based on the permutation algorithm DPAL that completes its job if the number of faults f is at most some f_0. It uses the graph $L(p)^\ell$ as its communication graph, where ℓ is set to some ℓ_0. The original t tasks are partitioned into chunks of size Δ, where Δ is the node degree of graph $L(p)^\ell$. Then each of the chunks is treated as a single task by DPAL, in the sense that a communication is performed only after the chunk is completed. This guarantees that the communication of *Part 1* is the same as its work. The *Part 1* algorithm is performed for some T_0 steps. After that all the processors that have not halted yet, if any, run the *Part 2* algorithm. *Part 2* starts by all the active processors sending messages to each other to learn who is alive and what tasks are still outstanding. The processors reset the lists of processors and tasks accordingly, containing, say, at most p_1 processors and t_1 tasks. Then algorithm DMY of De Prisco, Mayer and Yung [10], is run. The work of algorithm DMY is $\mathcal{O}(t_1 + (f_1+1)p_1)$ and its communication is $\mathcal{O}((f_1+1)p_1)$, where f_1 is the number of faults that occur during this phase.

The parameters f_0, ℓ_0 and T_0 are selected so as to make our estimates of work of *Part 1* and *Part 2* approximately equal. Let us fix $a > 0$ such that also the inequality

$$a < 1 - \frac{2\log_\lambda \Delta_0}{1 + 2\log_\lambda \Delta_0}$$

holds. We set the parameters of algorithm UAL as follows:

$$f_0 = p - p^{1-a} \ , \qquad \ell_0 = 2\log_\lambda \frac{p}{p - f_0} + 1 \qquad \text{and} \qquad T_0 = \frac{t + \Delta_0^{\ell_0} p \log^2 p}{p - f_0} \ . \quad (3)$$

The work and communication of the resulting algorithm UAL is $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p^{2-a})$, as stated in Theorem 3.

Theorem 3. *There is a deterministic algorithm that solves the Do-All problem with effort $\mathcal{W} + \mathcal{M} = \mathcal{O}(t + p^{2-a})$ against the unbounded adversary, where $a >$ 0.23 is a constant.*

Proof. The effort of *Part 1* is $\mathcal{O}(t + \Delta p \log^2 p)$ and that of *Part 2* is $\mathcal{O}(t + p(p-f))$, where $f < p$ is a variable. Equating these two bounds we obtain

$$\Delta \log^2 p = p - f \ . \quad (4)$$

As in the proofs of Lemmas 1 and 3, the node degree of the power of graph $L(p)$ is $\Delta = \Delta_0^k$, where $k = 2\log_\lambda \frac{p}{p-f} + 1$, provided that $k+1$ is not bigger than the girth of $L(p)$, that is, the length of the shortest cycle. It was shown in [17] that this girth is at least $\frac{1}{2}\log_{\Delta_0} p$. The inequality

$$2\log_\lambda \frac{p}{p - f_0} + 1 < \frac{1}{2}\log_{\Delta_0} p$$

can be checked by inspection, so we can proceed.

Since $\log_\lambda \frac{p}{p-f} = \log_{\Delta_0} \frac{p}{p-f} \cdot \log_\lambda \Delta_0$, we can convert equation (4) into an equivalent form

$$\Delta_0 \Big(\frac{p}{p-f}\Big)^{2\log_\lambda \Delta_0} = \frac{p-f}{\log^2 p} \ ,$$

which is then equivalent to $\varDelta_0 p^{2\log_\lambda \varDelta_0} \log^2 p = (p-f)^{1+2\log_\lambda \varDelta_0}$. Solving for f we obtain $f_0 = p - p^{1-a}$, where $a = \frac{1}{1+2\log_\lambda \varDelta_0} + o(1)$. The effort of algorithm UAL is

$$\mathcal{O}(t + \varDelta p \log^2 p) = \mathcal{O}(t + p(p - f_0)) = \mathcal{O}(t + p^{2-a}) \ .$$

The estimates in [3,19,20] imply that with $\varDelta_0 = 70$ the constant λ is at least as large as 7/2. If the constant a and parameters f_0, ℓ_0 and T_0 in equations (3) are set with 7/2 as the value of λ, then one can verify that $a > 0.23$. □

6 Discussion

We presented the first solution for *Do-All* that has both work *and* communication *subquadratic* in p in the worst case. There remains a substantial gap between our upper bound and the best known lower bound on work of $\varOmega(t+p\log p/\log\log p)$. Narrowing this gap, in particular, improving the lower bound, is challenging.

References

1. N. Alon, Eigenvalues and expanders, *Combinatorica*, 6 (1986) 83–96.
2. N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, Scalable secure storage when half the system is faulty, in *Proc., 27th Int. Colloquium on Automata, Languages and Programming*, 2000, Springer LNCS 1853, pp. 577–587.
3. N. Alon, and V.D. Milman, λ_1, isoperimetric inequalities for graphs, and superconcentrators, *J. Combin. Theory Ser. B*, 38 (1985) 73–88.
4. B.S. Chlebus, R. De Prisco, and A.A. Shvartsman, Performing tasks on synchronous restartable message-passing processors, *Distributed Computing*, 14 (2001) 49–64.
5. B.S. Chlebus, and D.R. Kowalski, Gossiping to reach consensus, in *Proc., 14th ACM Symp. on Parallel Algorithms and Architectures,* 2002, Winnipeg, Manitoba, Canada, to appear.
6. B.S. Chlebus, and D.R. Kowalski, Randomization helps to perform independent tasks reliably, submitted. A preliminary version appeared as: Randomization helps to perform tasks on processors prone to failures, in *Proc. 13th Int. Symposium on Distributed Computing*, 1999, Springer LNCS 1693, pp. 284–296.
7. B.S. Chlebus, D.R. Kowalski, and A. Lingas, The do-all problem in broadcast networks, in *Proc., 20th ACM Symp. on Principles of Distributed Computing,* 2001, Newport, Rhode Island, pp. 117–126.
8. A. Clementi, A. Monti, and R. Silvestri, Optimal f-reliable protocols for the do-all problem on single-hop wireless networks, in *Proc., 13th Int. Symposium on Algorithms and Computation,* 2002, to appear.
9. H. Davenport, "Multiplicative Number Theory," 2nd ed. revised by H.L. Montgomery, Springer Verlag, Berlin, 1980.
10. R. De Prisco, A. Mayer, and M. Yung, Time-optimal message-efficient work performance in the presence of faults, in *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 1994, pp. 161–172.
11. K. Diks, and A. Pelc, Optimal adaptive broadcasting with a bounded fraction of faulty nodes, *Algorithmica*, 28 (2000) 37–50.

12. C. Dwork, J. Halpern, and O. Waarts, Performing work efficiently in the presence of faults, *SIAM J. on Computing*, 27 (1998) 1457–1491.
13. A. Fiat and J. Saia, Censorship resistant peer-to-peer content addressable networks, in *Proc., 13th ACM-SIAM Symp. on Discrete Algorithms*, 2002.
14. Z. Galil, A. Mayer, and M. Yung, Resolving message complexity of byzantine agreement and beyond, in *Proc. 36th IEEE Symp. on Foundations of Computer Science*, 1995, pp. 724–733.
15. C. Georgiou, A. Russell, and A.A. Shvartsman, The complexity of synchronous iterative do-all with crashes, in *Proc., 15th Int. Symposium on Distributed Computing,* 2001, Springer LNCS 2180, pp. 151–165.
16. P.C. Kanellakis, and A.A. Shvartsman, Efficient parallel algorithms can be made robust, *Distributed Computing*, 5 (1992) 201–217.
17. A. Lubotzky, R. Phillips, and P. Sarnak, Ramanujan graphs, *Combinatorica*, 8 (1988) 261–277.
18. G.A. Margulis, Explicit constructions of concentrators, *Prob. Per. Infor.* 9 (1973) 71–80; English transl. in *Problems of Information Transmission*, (1975) 325–332.
19. R.M. Tanner, Explicit concentrators from generalized N-gons, *SIAM J. Alg. Disc. Methods*, 5 (1984) 287–293.
20. E. Upfal, Tolerating a linear number of faults in networks of bounded degree, *Information and Computation*, 115 (1994) 312–320.

Minimal Byzantine Storage

Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin

Department of Computer Science
University of Texas at Austin
Austin, Texas 78712 {jpmartin, lorenzo, dahlin}@cs.utexas.edu

Abstract. Byzantine fault-tolerant storage systems can provide high availability in hazardous environments, but the redundant servers they require increase software development and hardware costs. In order to minimize the number of servers required to implement fault-tolerant storage services, we develop a new algorithm that uses a "Listeners" pattern of network communication to detect and resolve ordering ambiguities created by concurrent accesses to the system. Our protocol requires $3f + 1$ servers to tolerate up to f Byzantine faults—f fewer than the $4f + 1$ required by existing protocols for non-self-verifying data. In addition, SBQ-L provides atomic consistency semantics, which is stronger than the regular or pseudo-atomic semantics provided by these existing protocols. We show that this protocol is optimal in the number of servers—any protocol that provides safe semantics or stronger requires at least $3f + 1$ servers to tolerate f Byzantine faults in an asynchronous system. Finally, we examine a non-confirmable writes variation of the SBQ-L protocol where a client cannot determine when its writes complete. We show that SBQ-L with non–confirmable writes provides regular semantics with $2f + 1$ servers and that this number of servers is minimal.

1 Introduction

Byzantine storage services are useful for systems that need to provide high availability. These services guarantee data integrity and availability in the presence of arbitrary (*Byzantine*) failures. A common way to design such a system is to build a *quorum system*. A quorum system stores a shared variable at a set of servers and performs read and write operations at some subset of servers (a *quorum*). Quorum protocols define an intersection property for the quorums which, in addition to the rest of the protocol description, ensures that each read has access to the current value of the variable. Byzantine quorum systems enforce the intersection property necessary for their consistency semantics in the presence of Byzantine failures.

The number of servers in a Byzantine storage system is a crucial metric since server failures must be independent. Therefore, to reduce the correlation of software failures, each server should use a different software implementation [18]. The first advantage of reducing the number of servers necessary for a service is the reduction in hardware costs. However, as hardware costs get cheaper in comparison to software and maintenance costs, the most important benefit of reducing the number of different servers is the corresponding reduction in development and maintenance costs. Furthermore, for large software systems (e.g. NFS, DNS, JVM) a fixed number of implementations may be available, but it is expensive or otherwise infeasible to create additional implementations. In such a situation, a new protocol requiring fewer servers may enable replication techniques where they were not previously applicable.

To minimize the number of servers, we present a new protocol called Small Byzantine Quorums with Listeners (SBQ-L). The protocol uses a "Listeners" pattern of communication to detect and resolve ordering ambiguities when reads and writes simultaneously access a shared variable.[1] Whereas existing algorithms use a fixed number of communication rounds, servers and readers using SBQ-L exchange additional messages when writes are concurrent with reads. This communication pattern allows the reader to monitor the evolution of the global state instead of relying on a snapshot. As a result, SBQ-L provides strong consistency semantics using fewer servers. In particular, Table 1 shows that SBQ-L provides atomic semantics [9] for generic data using as few as $3f + 1$ servers to tolerate f faults, instead of the $4f + 1$ servers that were previously required to provide even the weaker regular [11] or partial-atomic [17] semantics.

We show that SBQ-L is optimal with respect to the number of servers required to provide a safe shared variable in the common model of asynchronous reliable authenticated channels [1,3,11,12,13]. In particular, we show that any protocol that tolerates f Byzantine failures and that provides safe or stronger semantics must use at least $3f + 1$ servers. Since SBQ-L can provide atomic semantics with $3f + 1$ servers, it is optimal with respect to this critical metric.

We apply the SBQ-L protocol and our lower bound analysis to compare protocols for generic data to these for *self-verifying data* (data that cannot be undetectably altered, e.g. that are digitally signed). We find that, surprisingly, SBQ-L performs equally well with generic or self-verifying data. Existing protocols require more servers for generic data (second column of Table 1). Our lower bound of $3f + 1$ servers applies regardless of whether data is generic or self-verifying. Therefore our SBQ-L protocol, already optimal for generic data, cannot be improved by using self-verifying data. This analysis suggests that the distinction between these two classes of protocols is not as fundamental as previous results imply.

We also examine the distinction between protocols with *confirmable* writes and those with *non-confirmable* writes. Consistency semantics are defined in terms of conditions that must hold when reads and writes complete; however the specification for when a write completes is left out of the definition. The traditional approach defines the completion of the write as the instant when the writer completes its protocol. We call these protocols confirmable. If instead write completion is defined in a way that cannot be locally determined by the writer, but writes are still guaranteed to eventually complete, we say that the resulting protocol is non-confirmable.

[1] We call this communication model "Listeners" because of its similarity with the Listeners object-oriented pattern introduced by Gamma et. al. [8].

The bottom two lines of Table 1 indicate that the SBQ-L protocol can be modified to be non-confirmable. In that configuration, it can provide regular semantics for generic data using only $2f+1$ servers instead of the $3f+1$ required in prior work [15]. We again show that our protocol is optimal by proving that $2f+1$ servers are required to provide even safe semantics for non-confirmable writes. The existence of the SBQ-L protocol shows that this bound is tight. This result shows that the distinction between confirmable and non-confirmable protocols is fundamental.

Table 1. Required number of servers and semantics for various protocols for Byzantine distributed shared memory. New results and improvements over previous protocols are shown in bold

Required Semantics	Existing Protocols	SBQ-L	Safe Semantics
conf., generic	$4f+1$, safe [11,12][2],[15][1] $4f+1$, partial-atomic [17][2]	**$3f+1$, atomic**[2]	**$\geq 3f+1$ servers**
conf., self-verifying	$3f+1$, regular [11],[15][1]; $3f+1$, atomic [12],[5][1,2]	$3f+1$, atomic[2]	
non-conf., generic	$3f+1$, safe [15]	**$2f+1$, regular**[2]	**$\geq 2f+1$ servers**
non-conf., self-verifying	$2f+1$, regular [15]	$2f+1$, regular[2]	

(1) Does not require reliable channels. (2) Tolerates faulty clients.

Like other quorum protocols, SBQ-L guarantees correctness by ensuring that reads and writes intersect in a sufficient number of servers. SBQ-L differs from many traditional quorum protocols in that in a minimal-server threshold configuration, clients send messages to all servers on read and write operations.[2] Most existing quorum protocols access a subset of servers on each operation for two reasons: to tolerate server faults and reduce load. Note that SBQ-L's fault tolerance and load properties are similar to those of existing protocols. In particular, SBQ-L can tolerate f faults, including f non-responsive servers. Although in its minimal-server configuration it sends read and write requests to all $3f+1$ servers, this number is no higher than the $3f+1$ (out of $4f+1$) servers contacted by most existing protocols. We note that the fact that SBQ-L contacts a large fraction of servers on each operation is a direct consequence of the minimality of the number of servers.

The rest of this paper is organized as follows. Section 2 presents our model and assumptions and reviews the different semantics that distributed shared memory can provide, Section 3 presents the SBQ-L protocol, and Section 4 proves bounds on the number of servers required to implement these semantics. In Section 5, we explore practical considerations, including how to tolerate faulty clients, the trade-offs between bandwidth and concurrency, and how to avoid live-lock or memory problems during concurrent execution. Section 6 discusses related work and we conclude in the last section.

[2] As described in Section 3, it is possible to use more servers than the minimum and in this case only a subset of the servers is touched for every read.

2 Preliminaries

2.1 Model

We assume a system model commonly adopted by previous work in which quorum systems are used to tolerate Byzantine faults [1,3,11,12,13]. In particular, our model consists of an arbitrary number of clients and a set U of data servers such that the number $n = |U|$ of servers is fixed. A *quorum system* $Q \subseteq 2^U$ is a non-empty set of subsets of U, each of which is called a *quorum*.

Servers can be either *correct* or *faulty*. A correct server follows its specification; a faulty server can arbitrarily deviate from its specification. Following Malkhi and Reiter [11], we define a *fail-prone system* $\mathcal{B} \subseteq 2^U$ as a non-empty set of subsets of U, none of which is contained in another, such that some $B \in \mathcal{B}$ contains all faulty servers. Fail-prone systems can be used to describe the common *f-threshold* assumption that up to a threshold f of servers fail (in which case $\mathcal{B}$ contains all sets of f servers), but they can also describe more general situations, such as when some computers are known to be more likely to fail than others.

The set of clients of the service is disjoint from U and clients communicate with servers over point-to-point channels that are authenticated, reliable, and asynchronous. We discuss the implications of assuming reliable communication under a Byzantine failure model in detail in our previous work [15]. Initially, we restrict our attention to server failures and assume that clients are correct. We relax this assumption in Section 5.1.

2.2 Consistency Semantics

Consistency semantics define system behavior in the presence of concurrency. Lamport [9] defines the three semantics for distributed shared memory listed below. His original definitions exclude concurrent writes, so we present extended definitions that include these [17].

Using a global clock, we assign a time to the *start* and *end* (or completion) of each operation. We say that an operation A *happens before* another operation B if A ends before B starts. We then require that all operations be totally ordered using a relation $\dashrightarrow$ (*serialized order*) that is consistent with the partial order of the *happens before* relation. In this total order, we call write w the *latest completed write* relative to some read r if $w \dashrightarrow r$ and there is no other write w' such that $w \dashrightarrow w' \wedge w' \dashrightarrow r$. We say that two operations A and B are *concurrent* if neither A happens before B nor B happens before A. The semantics below hold if there exists some relation $\dashrightarrow$ that satisfies the requirements.

- *safe* semantics guarantee that a read r that is not concurrent with any write returns the value of the latest completed write relative to r. A read concurrent with a write can return any value.
- *regular* semantics provide safe semantics and guarantee that if a read r is concurrent with one or more writes, then it returns either the latest completed write relative to r or one of the values being written concurrently with r.

– *atomic* semantics provide regular semantics and guarantee that the sequence of values read by any given client is consistent with the global serialization order ($\rightarrow$).

The above definitions do not specify when the write completes. The choice is left to the specific protocol. In all cases, the completion of a write is a well-defined event. We will begin by considering only protocols in which the writer can determine when its write has completed (confirmable protocols). We later relax this requirement in Section 3.2 and show that the resulting protocols with non-confirmable writes require fewer servers.

```
W1  Write(D) {
W2          send (QUERY_TS) to all servers
W3          receive answer (TS, $ts$) from server $isvr$ set $ts[isvr] := ts$
W4          wait until the $ts[\,]$ array contains $q_w$ answers.
W5          $max\_ts := max\{ts[\,]\}$
W6          $ts := min\{t \in T_c : max\_ts < t \wedge last\_ts < t\}$
            // $ts \in T_c$ is larger than all answers and previous timestamp
W7          $last\_ts := ts$
W8          send (STORE, $D, ts$) to all servers.
W9          receive answer (ACK,$ts$) from server $i$
W10         wait until $q_w$ servers have sent an ACK message
W11 }

R1  (D,ts) = Read() {
R2          send (READ) to $q_r$ servers.
R3          loop {
R4                  receive answer (VALUE,$D, ts$) from server $s$
                     // (possibly more than one answer per server)
R5                  if $ts > latest[s].ts$ then $latest[s] := (D, ts)$
R6                  if $s \notin S$: // we call this event an "entrance"
R7                          $S := S \cup \{s\}$
R8                          $T :=$ the $f + 1$ largest timestamps in $latest[\,]$
R9                          for all $isvr$, for all $jtime \notin T$, delete $answer[isvr, jtime]$
R10                         for all $isvr$,
R11                                 if $latest[isvr].ts \in T$
                                     then $answer[isvr, latest[isvr].ts] := latest[isvr]$
R12                 if $ts \in T$ then $answer[s, ts] := (D, ts)$
R13         } until $\exists D, ts, W :: |W| \geq q_w \wedge (\forall i : i \in W : answer[i, ts] = (D, ts))$
            // i.e., loop until $q_w$ servers agree on a (D,ts) value
R14         send (READ_COMPLETE) to all servers
R15         return $(D, ts)$
R16 }
```

Fig. 1. *Confirmable SBQ-L client protocol for the f-threshold error model*

variable	initial value	notes
q_w	$\left\lceil \frac{n+f+1}{2} \right\rceil$	Size of the write quorum
q_r	$\left\lceil \frac{n+3f+1}{2} \right\rceil$	Size of the read quorum
T_c	Set of timestamps for client c	The sets used by different clients are disjoint
$last\_ts$	0	Largest timestamp written by a particular server
$latest[\,]$	$\emptyset$	A vector storing the largest timestamp received from each server and the associated data
$answer[\,][\,]$	$\emptyset$	Sparse matrix storing at most $f+1$ data and timestamps received from each server
S	$\emptyset$	The set of servers from which the reader has received an answer

Fig. 2. Client variables

3 The SBQ-L Protocol

Figure 1 presents the f-threshold[3] SBQ-L confirmable client protocol for generic data. The initial values of the protocol's variables are shown in Figure 2.

In lines W1 through W6, the Write() function queries a quorum of servers in order to determine the new timestamp. The writer then sends its timestamped data to all servers at line W8 and waits for acknowledgments at lines W9 and W10. The Read() function queries an access set [4] of servers in line R2 and waits for messages in lines R3 to R13. An unusual feature of this protocol is that servers send more than one reply if writes are in progress. For each read in progress, a reader maintains a matrix of the different answers and timestamps from the servers (`answers[][]`). The read decides on a value at line R13 once the reader can determine that a quorum of servers vouch for the same data item and timestamp, and a notification is sent to the servers at line R14 to indicate the completion of the read. A naïve implementation of this technique could result in the client's memory usage being unbounded; instead, the protocol only retains at most $f+1$ answers from each server.

This protocol differs from previous Byzantine quorum system (BQS) protocols because of the communication pattern it uses to ensure that a reader receives a sufficient number of *sound* and *timely* values. A reader receives different values from different servers for two reasons. First, a server may be faulty and supply incorrect or old values to a client. Second, correct servers may receive concurrent read and write requests and process them in different orders.

Traditional quorum systems use a fixed number of rounds of messages but communicate with quorums that are large enough to guarantee that intersections of read and write quorums contain enough answers for the reader to identify a value that meets the consistency guarantee of the system. Rather than using extra servers to disambiguate concurrency, SBQ-L uses extra rounds of messages when servers and clients detect writes concurrent with reads. Intuitively, other protocols take a "snapshot" of the situation. The SBQ-L protocol looks at the evolution of the situation in time: it views a "movie".

[3] We describe the more general quorum SBQ-L protocols in the extended technical report [14].

SBQ-L's approach uses more messages than some other protocols. Other than the single additional READ_COMPLETE message sent to each server at line R14, however, additional messages are only sent when writes are concurrent with a read.

Figure 1 shows the protocol for clients. Servers follow simpler rules: they only store a single timestamped data version, replacing it whenever they receive a STORE message with a newer timestamp. When receiving a read request, they send the contents of this storage. Servers in SBQ-L differ from previous protocols in what we call the Listeners communication pattern: after sending the first message, the server keeps a list of clients who have a read in progress. Later, if they receive a STORE message, then in addition to the normal processing they echo the contents of the store message to the "listening" readers – including messages with a timestamp that is not as recent as the data's current one but more recent than the data's timestamp at the start of the read. This listening process continues until the server receives a READ_COMPLETE message from the client indicating that the read has completed.

This protocol requires a minimum of $3f+1$ servers and provides atomic semantics with confirmable writes. We prove its correctness in the next section. Theorem 2 of Section 4 shows that $3f+1$ is the minimal number of servers for confirmable protocols. In Section 5.1 we show how to adapt this protocol for faulty clients.

3.1 Correctness

Traditional quorum protocols abstract away the notion of group communication and only concern themselves with contacting groups of responsive servers. Instead, our protocol specifies to which group of servers the messages should be sent and waits for acknowledgments from some quorum of servers within this access group. The read protocol relies on the acknowledged messages for safety, but it also potentially relies on the messages that are still in transit, for liveness. Because the channels are reliable, we know that these messages will eventually reach their destination.

Theorem 1. *The confirmable f-threshold SBQ-L protocol provides atomic semantics.*

Lemma 1 (Atomicity). *The confirmable threshold SBQ-L satisfies atomic semantics, assuming it is live.*

The SBQ-L protocol guarantees atomic semantics, in which the writes are ordered according to their timestamps. To prove this, we show that (1) after a write for a given timestamp ts_1 completes, no read can return a value with an earlier timestamp and (2) after a client c reads a timestamp ts_1, no later read can return a value with an earlier timestamp.

(1) Suppose a write for timestamp ts_1 has completed; then $\lceil\frac{n+f+1}{2}\rceil$ servers have acknowledged the write. At least $\lceil\frac{n-f+1}{2}\rceil$ of these are correct. In the worst

case, all the remaining servers can return the same stale or wrong reply to later reads. However there are only $\lceil \frac{n+f-1}{2} \rceil$ of them so they cannot form a quorum.

(2) Suppose that at some global time t_1, some client c reads timestamp ts_1. That means that $\lceil \frac{n+f+1}{2} \rceil$ servers returned a value indicating that this timestamp has been written, and again at least $\lceil \frac{n-f+1}{2} \rceil$ of these are correct: the remaining servers are too few to form a quorum. □

Lemma 2 (Liveness). *All functions of the confirmable threshold SBQ-L eventually terminate.*

For space reasons, we refer the reader to our extended technical report [14] for the proof of this lemma.

3.2 Non-confirmable Protocol

If a protocol defines the write completion predicate so that completion can be determined locally by a writer and all writes eventually complete, we call the protocol *confirmable*. This definition is intuitive and therefore implicitly assumed in most previous work. These protocols typically implement their `Write()` function so that it only returns after the write operation has completed.

If instead a protocol's write completion predicate depends on the global state in such a way that completion cannot be determined by a client although all writes still eventually complete, then we call the protocol *non-confirmable*. Non-confirmable protocols cannot provide blocking writes. The SBQ protocol [15], for example, is non-confirmable: writes complete when a quorum of correct servers have finished processing the write. This completion event is well-defined but clients cannot determine when it happens because they lack the knowledge of which servers are faulty.

The confirmable SBQ-L protocol of Section 3 requires at least $3f+1$ servers. This number can be reduced to $2f+1$ if the protocol is modified to become non-confirmable. The non-confirmable protocol is presented and proven correct in the extended technical report [14].

4 Bounds

In this section, we prove lower bounds on the number of servers required to implement minimal consistency semantics (safe semantics) in confirmable protocols. The bound is $3f+1$ and applies to any fault-tolerant storage protocol because the proof makes no assumption about how the protocol behaves. This lower bound not only applies to quorum protocols such as SBQ-L, but also to any other fault-tolerant storage protocol, even randomized ones. Also, the bounds hold whether or not data are self-verifying. Since the SBQ-L protocol of the previous section meets this bound, we know it is tight.

In previous work, protocols using self-verifying data often require f fewer servers than otherwise [11,15]. It was not known until now whether self-verifying data make a fundamental difference or if protocols using only generic (i.e. non-self-verifying) data could be made to perform as well. Although we show that self-verifying data has no impact on the minimal number of servers, they may be useful for other properties such as the number of messages exchanged or the ability to restrict access to the shared variables.

4.1 Confirmable Safe Semantics

Theorem 2. *In the authenticated asynchronous model with Byzantine failures and reliable channels, no live confirmable protocol can satisfy the safe semantics for distributed shared memory using* $3f$ *servers.*

To prove this impossibility we show that under these assumptions any protocol must violate either safety or liveness. If a protocol always relies on $2f+1$ or more servers for all read operations, it is not live. But if a live protocol ever relies on $2f$ or fewer servers to service a read request, it is not safe because it could violate safe semantics. We use the definition below to formalize the intuition that any such protocol will have to rely on at least one faulty server.

Definition 1. *A message* m *is* influenced by *a server* s *iff the sending of* m *causally depends [10] on some message sent by* s*.*

Definition 2. *A* reachable quiet system state *is a state that can be reached by running the protocol with the specified fault model and in which no read or write is in progress.*

Lemma 3. *For all live confirmable write protocols using* $3f$ *servers, for all sets* S *of* $2f$ *servers, for all reachable quiet system states, there exists at least one execution in which a write is only influenced by servers in a set* S' *such that* $S' \subseteq S$*.*

By contradiction: suppose that from some reachable quiet system state all possible executions for some writer are influenced by more than $2f$ servers. If the f faulty servers crash before the write then the writer can only receive messages that are influenced by the remaining $2f$ servers and the confirmable write execution will not complete. □

Note that this lemma can easily be extended to the read protocol.

Lemma 4. *For all live read protocols using* $3f$ *servers, for all sets* S *of* $2f$ *servers, for all reachable quiet system states, there exists at least one execution in which a read is only influenced by servers in a set* S' *such that* $S' \subseteq S$*.*

Thus, if there are $3f$ servers, all read and write operations must at some point depend on $2f$ or fewer servers in order to be live. We now show that if we assume a protocol to be live it cannot be safe by showing that there is always some case where the read operation fails.

Lemma 5. *Consider a live read protocol using $3f$ servers. There exist executions for which this protocol does not satisfy safe semantics.*

Informally, this read protocol sometimes decides on a value after consulting only with $2f$ servers. We prove that this protocol is not safe by constructing a scenario in which safe semantics are violated.

Because the protocol is live, for each write operation there exists at least one execution e_w that is influenced by $2f$ or fewer servers (by Lemma 3). Without loss of generality, we number the influencing servers 0 to $2f-1$. Immediately before the write e_w, the servers have states $a_0 \ldots a_{3f-1}$ ("state A") and immediately afterwards they have states $b_0 \ldots b_{2f-1}, a_{2f} \ldots a_{3f-1}$ ("state B"). Further suppose that the shared variable had value "A" before the write and has value "B" after the write. If the system is in state A then all reads should return the value A; in particular this holds for the reads that influence fewer than $2f+1$ servers. Consider such a read whose execution we call e. Execution e receives messages that are influenced by servers f to $3f-1$ and returns a value for the read based on messages that are influenced by $2f$ or fewer servers; in this case, it returns A. Lemma 4 guarantees that execution e exists.

Now consider what happens if execution e were to occur when the system is in state B. Suppose also that servers f to $2f-1$ are faulty and behave as if their states were $a_f \ldots a_{2f-1}$. This is possible because they have been in these states before. In this situation, states A and B are indistinguishable for execution e and therefore the read will return A even though the correct answer is B.

$\square$

The last two lemmas show that in the conditions given, no read protocol can be live and safe. $\square$

4.2 Non-confirmable Safe Semantics

For non-confirmable protocols, the minimum number of servers for safe semantics is $2f+1$ instead of $3f+1$ for confirmable protocols. We refer the reader to the extended technical report [14] for the proof.

Theorem 3. *In the reliable authenticated asynchronous model with Byzantine failures, no live protocol can satisfy the safe semantics for distributed shared memory using $2f$ servers.*

5 Practical Considerations

In the next subsections we show how to handle faulty clients, quantify the number of additional messages, experimentally measure the effect of additional messages, discuss the protocol latency, and show an upper bound on memory usage.

5.1 Faulty Clients

The protocols in the previous two sections are susceptible to faulty clients in two ways: (1) faulty clients can choose not to follow the write protocol and prevent future reads from terminating or (2) faulty clients can violate the read protocol to waste server resources. We extend the protocol to address these issues below.

Liveness. Faulty writers can prevent future read attempts from terminating by making sure that no quorum of servers has the same value (a *poisonous write*), for example by sending a different value to each server. All reads will then fail because they cannot gather a quorum of identical answers.

Poisonous writes can be prevented if clients sign their writes and servers propagate among themselves the write messages they receive. This modification ensures that the servers will reach a consistent state, it is described in more detail in the extended technical report [14].

Resource Exhaustion. A faulty reader can neglect to notify the servers that the read has completed and force the server to continue that read operation forever. The cause of the problem is that readers can cause a potentially unbounded amount of work at the servers (the processing of a nonterminating read request) at the cost of only constant work (a single faulty read request).

This attack can be rendered impractical by removing the imbalance in the protocol, forcing the readers to contact the servers periodically. The resulting protocol is always safe and relies on good network behavior for liveness. It is described in more detail in the extended technical report [14].

5.2 Additional Messages

SBQ-L's write operation requires $3n$ messages in the non-confirmable case and $4n$ messages in the confirmable case, where n is the number of servers, regardless of concurrency. This communication is identical to previous results: the non-confirmable SBQ protocol [15] uses $3n$ messages and the confirmable MR protocol [11] requires $4n$ messages.

The behavior of the SBQ-L read operation depends on the number of concurrent writes. Other protocols (both SBQ and MR) exchange a maximum of $2n$ messages for each read. SBQ-L requires up to $3n$ messages when there is no concurrency. In particular, step R14 adds a new round of messages. Additional messages are exchanged when there is concurrency because the servers echo all concurrent write messages to the reader. If c writes are concurrent with a particular read then that read will use $3n + cn$ messages.

Even in the case of concurrency, the additional messages do not impact latency as severely as one may fear because most of them are asynchronous and unidirectional. The SBQ-L protocol will not wait for $3n+cn$ message roundtrips.

In order to experimentally test the overhead of the extra messages used to deal with concurrency in SBQ-L, we construct and test a simple prototype. These experiments are described in detail in the technical report [14]. We find that increasing concurrency has a measurable but modest effect on the read latency.

5.3 Maximum Throughput

A goal of a BQS architecture is to support a high throughput for a low system cost. Two factors affect system cost: (1) the number of different servers n and (2) the required power of these servers, dictated by the load factor and the desired throughput. The *load factor* [16] is defined as "the minimal access probability of the busiest server, minimizing over the strategies", where the strategy is the algorithm used to pick a quorum.

SBQ-L has a load factor of $\frac{1}{2n}(n + \lceil\frac{n+2f+1}{2}\rceil)$ if only non-confirmable writes are supported and $\frac{1}{2n}(n + \lceil\frac{n+3f+1}{2}\rceil)$ if confirmable writes are also supported, assuming that reads and writes occur with equal frequency. Other protocols [5, 11,13] have a better asymptotic load factor, but either have a higher load factor for small values of n or cannot function using as few servers as SBQ-L.

A detailed comparison of cost to meet throughput goals depends on hardware costs, software costs and throughput goals and is outside of the scope of this paper. In general, when adding a server is expensive compared to buying a faster server, protocols such as SBQ-L that limit n may be economically attractive even if they increase the load factor.

5.4 Live Lock

The behavior of the protocol under heavy load must be described precisely to ensure the protocol remains live. In SBQ-L, writes cannot starve but reads can, if an infinite number of writes are in progress and if the servers always choose to serve the writes before sending the echo messages.

When serving a write request while a read is in progress, servers queue an echo message. The liveness of both readers and writers is guaranteed if we require servers to send these echoes before processing the next write request. A read will therefore eventually receive the necessary echoes to complete even if an arbitrary number of writes are concurrent with the read.

Another related concern is that of latency: can reads become arbitrarily slow? In the asynchronous model, there is no bound on the duration of reads. However, if we assume that writes never last longer than w units of time and that there are c concurrent writes, then in the worst case (taking failures into account) reads will be delayed by no more than $min(cw, nw)$. This result follows because in the worst case, f servers are faulty and return very high timestamps so that only one row of `answer[][]` contains answers from correct servers. Also, in the worst case each entrance (line R6) occurs just before the monitored write can be read. The second term is due to the fact that there are at most n entrances.

5.5 Buffer Memory

In SBQ-L, readers maintain a buffer in memory during each read operation (the `answer[][]` sparse matrix). While other protocols only need to identify a majority and as such require n units of memory, the SBQ-L protocol maintains a short history of the values written at each server. As a result, the read operation in SBQ-L requires up to $n(f+1)$ units of memory: the set T contains at most $f+1$ elements (line 8) and the `answer[][]` matrix therefore never contains more than n columns and $f+1$ rows (lines 9 and 12). In a system storing more than one shared variable, if multiple variables are read in parallel then each individual read requires its own buffer of size $n(f+1)$.

6 Related Work

Although both Byzantine failures [7] and quorums systems [6] have been studied for a long time, interest in quorum systems for Byzantine failures is relatively recent. The subject was first explored by Malkhi and Reiter [11,12]. They reduced the number of servers involved in communication [13], but not the total number of servers; their work exclusively covers confirmable systems.

In previous work we introduced non-confirmable protocols that require $3f+1$ servers ($2f+1$ for self-verifying data) [15]. In the present paper we expand on that work and reduce the bound to $2f+1$ for generic data and provide regular semantics instead of safe by using Listeners. We also prove lower bounds on the number of servers for these semantics and meet them.

Bazzi [3] explored Byzantine quorums in a synchronous environment with reliable channels. In that context it is possible to require fewer servers ($f+1$ for self-verifying data, $2f+1$ otherwise). This result is not directly comparable to ours since it uses a different model. We leave as future work the application of the Listeners idea of SBQ-L to the synchronous network model.

Bazzi [4] defines *non-blocking quorum system* as a quorum system in which the writer does not need to identify a live quorum but instead sends a message to a quorum of servers without concerning himself with whether these servers are responsive or not. According to this definition, all the protocols presented here use non-blocking quorum systems.

Several papers [4,13,16] study the load of Byzantine quorum systems, a measure of how increasing the number of servers influences the amount of work each individual server has to perform. A key conclusion of this previous work is that the lower bound for the load factor of quorum systems is $O(\frac{1}{\sqrt{n}})$. Our work instead focuses on reducing the number of servers necessary to tolerate a given fault threshold (or fail-prone system).

Phalanx [12] builds shared data abstractions and provides a locking service, both of which can tolerate Byzantine failure of servers or clients. It requires confirmable semantics in order to implement locks. Phalanx can handle faulty clients while providing safe semantics using $4f+1$ servers.

Castro and Liskov [5] present a replication algorithm that requires $3f+1$ servers and, unlike most of the work presented above, can tolerate unreliable network links and faulty clients. Their protocol uses cryptography to produce self-verifying data and provides linearizability and confirmable semantics. It is fast in the common case. Our work shows that confirmable semantics cannot be provided using fewer servers. Instead, we show a non-confirmable protocol with $2f+1$ servers. In the case of non-confirmable semantics, however, it is necessary to assume reliable links.

Attiya, Bar-Noy and Dolev [2] implement an atomic single-writer multi-reader register over asynchronous network, while restricting themselves to crash failures only. Their failure model and writer count are different from ours. When implementing finite-size timestamp, their protocol uses several rounds. The similarity stops there, however, because they make no assumption of network reliability and therefore cannot leverage unacknowledged messages the way the Listeners protocol does.

7 Conclusion

We present two protocols for shared variables, one that provides atomic semantics with non-confirmable writes using $2f+1$ servers and the other that provides atomic semantics with confirmable writes using $3f+1$ servers. In the reliable asynchronous communication model when not assuming self-verifying data, our protocols reduce the number of servers needed by previous protocols by f. Additionally, they improve the semantics for the non-confirmable case. Our protocols are strongly inspired by quorum systems but use an original communication pattern, the Listeners. The protocols can be adapted to either the f-threshold or the fail-prone error model.

The more theoretical contribution of this paper is the proof of a tight bound on the number of servers. We show that $3f+1$ servers are necessary to provide confirmable semantics and $2f+1$ servers are required otherwise.

Several protocols [5,11,12,15,18] use digital signatures (or MAC) to reduce the number of servers. It is therefore surprising that we were able to meet the minimum number of servers without using cryptography. Instead, our protocols send one additional message to all servers and other additional messages that only occur if concurrent writes are in progress.

Since our protocols for confirmable and non-confirmable semantics are nearly identical, it is possible to use both systems simultaneously. The server side of the protocols are the same, therefore the servers do not need to be aware of the model used. Instead, the clients can agree on whether to use confirmable or non-confirmable semantics on a per-variable basis. The clients that choose non-confirmable semantics can tolerate more failures: this property is unique to the SBQ-L protocol.

Acknowledgments. The authors thank Jian Yin and Mike Kistler for several interesting conversations and Alison Smith and Maria Jump for helpful comments on the paper's presentation.

References

1. L. Alvisi, D. Malkhi, E. Pierce, and R. Wright. Dynamic Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.
2. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM (JACM) Volume 42*, pages 124–142, 1995.
3. R. A. Bazzi. Synchronous Byzantine quorum systems. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 259–266, 1997.
4. R. A. Bazzi. Access cost for asynchronous Byzantine quorum systems. *Distributed Computing Journal volume 14, Issue 1*, pages 41–48, January 2001.
5. M. Castro and NB. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), New Orleans, USA*, pages 173–186, February 1999.
6. S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR) Volume 17, Issue 3*, pages 341–370, September 1985.
7. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Technical Report MIT/LCS/TR-282, 1982.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, October 1994. ISBN 0-201-63361-2.
9. L. Lamport. On interprocess communications. *Distributed Computing*, pages 77–101, 1986.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
11. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pages 203–213, 1998.
12. D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana, USA*, Oct 1998.
13. D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. In *Proceedings 16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 249–257, August 1997.
14. J-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. Technical Report TR-02-38, University of Texas at Austin, Department of Computer Sciences, August 2002.
15. J-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–383, June 2002.
16. M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.
17. E. Pierce and L. Alvisi. A recipe for atomic semantics for Byzantine quorum systems. Technical report, University of Texas at Austin, Department of Computer Sciences, May 2000.
18. R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

Wait-Free n-Set Consensus When Inputs Are Restricted

Hagit Attiya and Zvi Avidor

Department of Computer Science
The Technion, Haifa 32000, Israel
{hagit,azvi}@technion.ac.il

Abstract. The k-set consensus problem requires processes to decide on at most k of their input values. The problem can be solved using only read / write operations in the presence of f crash failures if and only if $f < k$. One way to subvert this impossibility result is to restrict the set of possible assignments of input values to processes. This paper presents a characterization of the input restrictions that allow a wait-free solution of n-set consensus in a system with $n + 1$ processes, using only read and write operations.

1 Introduction

The *k-set consensus* problem [1] requires nonfaulty processes to decide on at most k different values, so that every decision value is some process's input. The k-set consensus problem is solvable in an asynchronous system subject to f crash failures using read/write operations if and only if $f < k$. An algorithm[1] was presented in [1] while the lower bound was proved in three independent papers [2,3,4]. This is a rather disappointing result, having that usually the number of failures can not be controlled, and if failures are possible, then any number of failures can occur.

Several approaches were suggested to handle the impossibility of solving k-set consensus, and its important special case (with $k = 1$), the *consensus* problem. This includes usage of failure detectors [5], randomized methods [6,7] and approximate agreement [8].

An alternative approach is to restrict the set of possible input configurations as a way of solving unsolvable problems. For example, if only one input configuration is possible, then a a simple wait-free protocol solves the consensus problem: all processes decide on an arbitrary input value of this configuration without any communication. A natural question is whether restricting the set of possible input configurations allows to solve certain problems. Such a restriction can capture several practical situations where the input values of processes are related, e.g., after they have executed a lower-level coordination algorithm. This

[1] The algorithm is described for message passing systems, but can be easily adapted to shared memory systems.

can happen when subsets of processes share other means of communication like highly-reliable channels.

This paper characterizes which restrictions of the input configurations allow to solve n-set consensus, with a wait-free read / write algorithm in a system of $n+1$ processes. For input sets in which the problem is solvable, we provide a simple, single-round protocol. The paper also presents an algorithm for *deciding* whether a set of input configurations allows to solve consensus, using our characterization.

Our claims and proofs use the language of algebraic topology, although their nature is strictly combinatorial. The algebraic topology language was previously used to describe combinatorial claims, for example, by Attiya and Rajsbaum [9].

Taubendeld et al. [10,11] introduce the idea of restricting the set of inputs as mean to solve agreement problems. They presented possibility and impossibility results in a shared memory system and a hierarchy of problems that can be solved with up to f failures.

Friedman et al. [12] uses coding theory to classify the set of possible input configurations allowing to solve agreement problems in the presence of f failures.

Mostefaoui et al. [13] presented a condition-based approach for solving the consensus problem. For every $f > 1$, they define a set of sets of initial configurations $\mathbb{C}_f$ such that there is a read / write consensus algorithm tolerating f failures for input configurations in a set C, if and only if $C \in \mathbb{C}_f$. Later [14] they presented, for any k and $f(\geq k)$, a set of sets $\mathcal{S}_{f,k}$. They showed that an f-fault tolerant algorithm for k-set consensus assuming input configurations from a set C exists only if $C \in \mathcal{S}_{f,k}$. A matching impossibility result was presented only for the wait-free case.

It should be noticed that their definition of *solving* is different then ours. In their definition, all input configurations are possible but nonfaulty processes are required to terminate only if the initial configuration satisfies the condition. When the condition does not hold, nonfaulty processes may never decide, yet if they decide it must be on “correct” values. For example, they show [14, Theorem 3] that wait-free k-set consensus is solvable, under their definition, when the condition C prohibits input configurations with more than k values. This restriction means the input values already represent previous agreement. The non-trivial part of their protocol is to discover that no more than k input values are present. Our paper does not restrict the number of values in an input configuration: it is possible to solve k-set consensus even in configurations with $n+1$ values, provided that other input configurations are restricted.

2 Preliminaries

2.1 The Computation Model

We consider a shared-memory asynchronous system, where $n+1$ processes communicate by reading and writing variables in shared memory. The memory supports *immediate atomic snapshots* [15], which can be wait-free implemented from single-writer multi-reader variables. An immediate snapshot object provides an array $\boldsymbol{V}$. Process p_i writes to $\boldsymbol{V}[i]$ in a $\mathsf{WriteRead}_i(v)$ operation, which also returns a snapshot of $\boldsymbol{V}$. This operation can be modelled as a write operation, immediately[2] followed by an atomic snapshot operation, reading all entries of the shared array. In the protocol we provide for the n-set consensus we only use one-shot immediate snapshot, allowing WriteRead to be performed only once. This is equivalent to the *participating set* problem, where each process i writes its id into the shared memory and returns a set S_i of id's, satisfying the following conditions:

1. *Self-Containment*: $i \in S_i$
2. *Atomic Snapshot*: For all i, j, either $S_i \subseteq S_j$ or $S_j \subseteq S_i$.
3. *Immediacy:* For all i, j, if $i \in S_j$ then $S_i \subseteq S_j$.

2.2 k-Set Consensus

In a *decision problem*, each process starts with an input value, and after performing a *protocol*, it outputs a *decision value.*

The *k-set consensus* decision problem requires that nonfaulty processes decide on at most k different values. Formally, a protocol solves k-set consensus in the presence of f faults if the following conditions hold:

Validity: The decision value is some process's input value.
Termination: If less than f processes fail, then every nonfaulty process decides on some value.
Agreement: There are at most k different decision values.

2.3 Basic Notions of Algebraic Topology

We rely on the terminology of algebraic topology [16], cast in combinatorial terms.

An *n-simplex* is a set of size $n+1$; the elements of the set are called *vertices.* Simplexes are usually accompanied by a superscript, stating their *rank* (number of vertices minus one). Given an n-simplex S^n and a vertex v, $face_v(S^n)$ represents $S^n \setminus \{v\}$. A set of simplexes, closed under containment is a *simplicial complex*, or in short, a *complex.* If a complex is a result of taking a set of n-simplexes, and closing it by containment, it is sometimes referred to as an n-complex.

Note that for a complex K, $\bigcup K$ is the set of vertices of the complex.

Definition 1. *Given an n-simplex S^n and a vertex $v \notin S^n$, $v * S^n$ is the $(n+1)$-simplex containing S^n and v.*

A *configuration* C is a set of process-value pairs, one for each process, capturing the states of all processes in the system. It corresponds in a natural manner to a simplex $\{\langle p_i, \alpha_i \rangle\}_{i=0}^{n}$.

[2] This term is being used intentionally, since WriteRead operation is *not* an atomic operation. See [15] for more details.

Two functions retrieve information from a vertex of such a simplex S^n: $In(\langle p, \alpha\rangle) = \alpha$ is the value, and $Proc(\langle p, \alpha\rangle) = p$ is the process identifier. These functions naturally extend to full simplexes, $In(S^n) = \{\alpha \mid \langle p, \alpha\rangle \in S^n\}$ and $Proc(S^n) = \{p \mid \langle p, \alpha\rangle \in S^n\}$.

A set of input configurations $\mathcal{I}$ implies a *complex*, in a natural manner. In an *input* configuration, the values associated with each process are input values from the input domain $\mathcal{V}$. The resulting *input complex* is denoted $K(\mathcal{I})$.

In some of the previous papers using algebraic topology to analyze distributed algorithms (e.g. [3]), the term *output complex* (represented by $\mathcal{O}$) represents a complex whose contained simplices model the legal termination configurations. This means that each vertex in $\mathcal{O}$ is a pair of values: process id and a decision value. In this paper we are only interested in the set of decision values, decided by the nonfaulty processes, thus we define the *output complex* as a complex whose contained simplices represents legal sets of decided values. Since we're analyzing the k-set consensus protocol, each simplex in $\mathcal{O}$ is a set of at most k different values from $\mathcal{V}$.

To map between two complexes, one should conserve the structure of the complex, that is, a simplex must be mapped to a simplex. Such a map is usually defined between the sets of vertices of two complexes.

Definition 2. *A* simplicial map *between two simplicial complexes K and L is a map $\delta : \bigcup K \to \bigcup L$, such that $\delta(S) \in L$ for any $S \in K$.*

3 A Characterization for Wait-Free n-Set Consensus

The characterization of input configurations that allow to solve wait-free n-set consensus is based on the notion of processes' *knowledge*. This section defines this knowledge and casts it in terms of algebraic topology, allowing to state our characterization theorem. The theorem is proved later, in Section 5.

3.1 What Processes Know?

In an immediate-atomic-snapshot memory, a process knows only information contained in the sequence of views it obtains using $\mathsf{WriteRead}_i$ operations. The following definition captures this notion formally. Two executions σ_1 and σ_2 are *indistinguishable* to process p, denoted $\sigma_1 \overset{p}{\sim} \sigma_2$, if the sequence of views from p's WriteRead operations in σ_1 is equal to the sequence of views from p's WriteRead operations in σ_2. For a set of processes $\mathcal{P}$, say that σ_1 and σ_2 are indistinguishable to $\mathcal{P}$, denoted $\sigma_1 \overset{\mathcal{P}}{\sim} \sigma_2$, if $\sigma_1 \overset{p}{\sim} \sigma_2$ for every process $p \in \mathcal{P}$.

To provide better intuition about this term, we first define $\mathcal{P}'$-solo executions:

Definition 3. *For a set of processes $\mathcal{P}'$, a $\mathcal{P}'$-solo execution is an execution where only processes in $\mathcal{P}'$ perform steps.*

Assume the processes are initialized with input values from some configuration I. Consider an arbitrary $\mathcal{P}'$-solo execution of a subset of processes $\mathcal{P}' \subsetneq \mathcal{P}$. Processes in $\mathcal{P}'$ directly learn about each other's input values. When all input combinations are possible, they will know nothing about the input values of processes in $\mathcal{P} \setminus \mathcal{P}'$. On the other hand, if the possible initial configurations are restricted, processes in $\mathcal{P}'$ have some knowledge about the input values of $\mathcal{P} \setminus \mathcal{P}'$, without communicating with them at all.

For example, consider three processes p_0, p_1 and p_2. Assume that there are two possible input configurations: in I_1 each p_i starts with i, and in I_2, p_0 start with 0, p_1 with 3 and p_2 with 4.

When solving 2-set consensus, there is an execution in which p_0 decides 0. Let σ_i be the set of p_0-solo executions, starting from I_i, $i = 1, 2$. Since σ_1 and σ_2 are indistinguishable to p_0, it decides on the same value d in both configurations. Therefore, p_0 decides 0 in both σ_1 and σ_2, since otherwise, the validity condition is violated in either σ_1 or σ_2. That is, when a process has no additional information about the input values of other processes, it must decide on its own value in a solo execution.

In contrast, if only I_1 is possible, then p_0 may decide on a value other than its own input, e.g., p_1's input value, even without communicating with p_1.

3.2 Stating Knowledge in Topology Terms

Consider a sub-simplex T^m of an input simplex S_0^n. In a Proc(T^m)-solo execution, the processes in T^m may decide only on input values they know to be allowed for all possible input configurations. The validity condition of k-set consensus requires these values to be in *all* input configurations extending T^m (as proved below in Lemma 4). This motivates the next definition.

Definition 4. *Given a sub-simplex T^m of an n-simplex $S_0^n \in K(\mathcal{I})$, the* knowledge *of T^m is the set*

$$KN(T^m) = \bigcap_{S^n \in K(\mathcal{I}), S^n \supseteq T^m} In(S^n).$$

The simplex $S^n \supseteq T^m$ represents an allowed input configuration (possibly different than S_0^n) that extends T^m. Knowledge increases when the set of processes increase, as shown in the next simple lemma:

Lemma 1. *For any two simplexes T^{m_1}, S^{m_2}, if $T^{m_1} \subseteq S^{m_2}$, then $KN(T^{m_1}) \subseteq KN(S^{m_2})$.*

Proof.

$$KN(T^{m_1}) = \bigcap_{T^{m_1} \subseteq J^n} In(J^n) \subseteq \bigcap_{S^{m_2} \subseteq J^n} In(J^n) = KN(S^{m_2})$$

□

Our main theorem can now be stated.

Theorem 1. *There is a wait-free protocol solving n-set consensus for a set of input configurations $\mathcal{I}$ if and only if for every $S^n \in K(\mathcal{I})$, either $|In(S^n)| < n+1$ or there is a simplex $T^{n-1} \subset S^n$ such that $KN(T^{n-1}) = In(S^n)$.*

Intuitively, in order to solve n-set consensus for some set of input configurations, we need to consider two types of input configurations. For input configurations with fewer than $n+1$ input values to begin with, the protocol is trivial: just let each process decide upon it's input value. For input configurations with $n+1$ input values, knowledge is used. If a subset of the processes (T^{n-1}) knows about the input value of one of the remaining processes, the processes do not have to wait for this process to write its input value, and can decide by themselves. This abnormality allow to create such a wait-free protocol.

4 Subdivisions and Solvability

Our characterization theorem refers only to the input complex (spanned by the allowed input configurations), and does not postulate the existence or inexistence of *subdivisions*, as done for example, in [3]. However, to prove the theorem, we need to study the subdivision of the input complex which is implied by running a protocol. In this section, we define a specific kind of subdivision, the *standard chromatic subdivision*, and explain how it relates to protocols.

4.1 The Standard Chromatic Subdivision

There are several geometric and algebraic definitions of the standard chromatic subdivision [9,15,3]. We employ the pure combinatorial definition, first introduced by Herlihy and Shavit in [3].

Definition 5. *The* standard chromatic subdivision *of a simplex $S^n = \{x_0, x_1, \dots, x_n\}$ is the simplicial complex $\chi(S^n)$ containing all simplexes $\{\langle p_0, A_0\rangle, \langle p_1, A_1\rangle, \dots, \langle p_n, A_n\rangle\}$, where A_i are subsets of S^n such that the following conditions hold:*

1. Self-Containment*: $p_i \in Proc(A_i)$*
2. Atomic Snapshot*: if $j \in Proc(A_i)$ then $A_j \subseteq A_i$*
3. Immediacy*: for each $j, i \in Proc(S^n)$ either $A_j \subseteq A_i$ or $A_i \subseteq A_j$.*

The base *of S^n is the simplex $base(S^n) = \{\langle p_0, S^n\rangle, \langle p_1, S^n\rangle, \dots, \langle p_n, S^n\rangle\}$.*

Lemma 2. *Any $s^n \in \chi(S^n)$ can be written as $x * t^{n-1}$, where $x \in base(S^n)$ and $t^{n-1} \in \chi(S^n)$*

Proof. Consider a simplex $s^n = \{\langle p_0, A_0\rangle, \langle p_1, A_1\rangle, \dots, \langle p_n, A_n\rangle\} \in \chi(S^n)$. By the Immediacy there is a maximal set among A_i s, w.l.o.g. let it be A_n. By the Self-Containment we have that $A_n = S^n$, which implies the lemma. □

Lemma 3. *If $s^m \in \chi(T^m)$ and $T^m \subsetneq S^n$, then $s^m \cap base(S^n) = \emptyset$.*

Proof. By definition, $A_i \subseteq T^m$ for any $\langle p_i, A_i\rangle \in s^m$. Thus, there is no $\langle p_i, A_i\rangle \in s^m$ such that $A_i = S^n$. □

The standard chromatic subdivision can be applied repeatedly; $\chi^R(\mathcal{I})$ denotes the complex $\mathcal{I}$ subdivided R times.

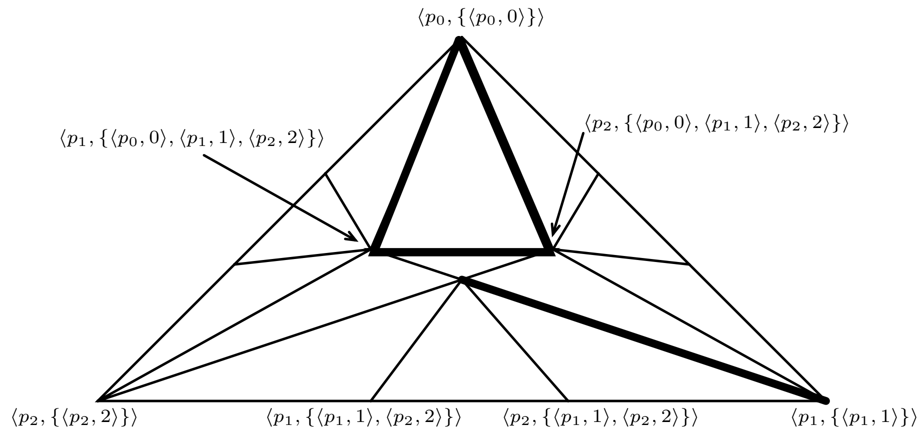

$\langle p_0, \{\langle p_0, 0\rangle\}\rangle$
$\langle p_1, \{\langle p_0, 0\rangle, \langle p_1, 1\rangle, \langle p_2, 2\rangle\}\rangle$
$\langle p_2, \{\langle p_0, 0\rangle, \langle p_1, 1\rangle, \langle p_2, 2\rangle\}\rangle$
$\langle p_2, \{\langle p_2, 2\rangle\}\rangle$
$\langle p_1, \{\langle p_1, 1\rangle, \langle p_2, 2\rangle\}\rangle$
$\langle p_2, \{\langle p_1, 1\rangle, \langle p_2, 2\rangle\}\rangle$
$\langle p_1, \{\langle p_1, 1\rangle\}\rangle$

Fig. 1. Graphical representation of execution

4.2 From a Standard Chromatic Subdivision to a Protocol

The standard chromatic subdivision corresponds in a natural manner to executions in which a process performs a WriteRead operation on an immediate snapshot object. The view returned by the WriteRead operation contains the input values of some of the processes. To provide an intuition, let us examine the one-shot immediate snapshot memory model, which as mentioned, is equivalent to the participating set problem. Notice the correspondence between the definition of the participating set problem and the definition of the standard chromatic subdivision.

For example, consider a system with three processes p_0, p_1, and p_2 starting with input values 0, 1 and 2. The following execution is possible:
1. p_0 calls WriteRead with input 0.
2. p_0 returns from WriteRead with value $\{\langle p_0, 0\rangle\}$
3. p_1 calls WriteRead with input 1
4. p_2 calls WriteRead with input 2
5. p_1 returns from WriteRead with value $\{\langle p_0, 0\rangle, \langle p_1, 1\rangle, \langle p_2, 2\rangle\})$
6. p_2 returns from WriteRead with value $\{\langle p_0, 0\rangle, \langle p_1, 1\rangle, \langle p_2, 2\rangle\})$

This execution is represented by the following simplex $s^n \in \chi(S^n)$, where $S^n = \{\langle p_0, 0\rangle, \langle p_1, 1\rangle, \langle p_2, 2\rangle\}$:

$$s^n = \left\{\left\langle p_0, \{\langle p_0, 0\rangle\}\right\rangle, \langle p_1, \{\langle p_0, 0\rangle, \langle p_1, 1\rangle, \langle p_2, 2\rangle\}\rangle, \left\langle p_2, \{\langle p_0, 0\rangle, \langle p_1, 1\rangle, \langle p_2, 2\rangle\}\right\rangle\right\}$$

Figure 1 describes $\chi(S^n)$. The vertices of the original simplex S^n, represents the solo-executions. E.g. the vertex $\langle p_0, 0\rangle \in S^n$, transformed into $\langle p_0, \{\langle p_0, 0\rangle\}\rangle \in \chi(S^n)$. The vertices of $base(S^n)$ are in the center. Each represents the set of executions where p_i saw the inputs of all the processes in the result from the WriteRead operation. The simplex s^n which represents the execution discussed above is represented by the emphasized triangle.

Definition 6. *A* k-coloring *is a simplicial map from an n-complex to a k-complex*[3]*.*

The topological function extracting the input values of the processes that some processes saw after executing for several stages is the *carrier*:

Definition 7. *The* carrier *is recursively defined:*

$$carrier(s^m) = \begin{cases} s^m & s^m \in K(\mathcal{I}) \\ carrier(\bigcup_{\langle p_i, A_i \rangle \in s^m} A_i) & s^m \in \chi^R(K(\mathcal{I})),\ R > 0 \end{cases}$$

For example, take $s^2 = \left\{ \left\langle p_0, \{\langle p_0, \alpha\rangle, \langle p_1, \beta\rangle, \langle p_2, \gamma\rangle\} \right\rangle, \left\langle p_1, \{\langle p_1, \beta\rangle\} \right\rangle \right\} \in \chi(\{\langle p_0, \alpha\rangle, \langle p_1, \beta\rangle, \langle p_2, \gamma\rangle\})$ (the emphasized line in Figure 1). Then,

$$carrier(s^2, K(\mathcal{I})) = \{\langle p_0, \alpha\rangle, \langle p_1, \beta\rangle, \langle p_2, \gamma\rangle\} \ .$$

That is, p_0 and p_1 together saw the input values of all processes; in fact, p_0 alone saw all input values.

Theorem 2. *If there is a k-coloring $\delta : \chi(K(\mathcal{I})) \to \mathcal{O}$ such that $\delta(v) \in KN(carrier(v, K(\mathcal{I})))$ for every $v \in \chi(K(\mathcal{I}))$, then there is a k-set consensus algorithm for input configurations in $\mathcal{I}$.*

Proof. In our k-set consensus algorithm, we use a single-shot immediate snapshot memory. When starting with initial configuration represented by the simplex S^n, each process p_i performs a WriteRead operation (with its input value). The returned value $A_i \subseteq S^n$ indicates a vertex $v = \langle p_i, A_i \rangle \in \chi(K(\mathcal{I}))$, the standard chromatic subdivision of the input complex. The process decides on $\delta(v)$, the color assigned to this vertex by δ.

Termination is straightforward, since each process performs exactly one WriteRead operation and decides. *Agreement* is also obvious, since δ is a simplicial map to a k-complex. It remains to prove *validity*.

Let I be an input configuration, and let $S^n \in K(\mathcal{I})$ be the corresponding simplex. In an execution starting from I, a process p decides on $\delta(v) \in KN(carrier(v, K(\mathcal{I})))$ where $v \in \chi(S^n)$ $(Proc(v) = p)$. *Validity* is proved by showing that $KN(carrier(v, K(\mathcal{I}))) \subseteq In(S^n)$.

By Definition 7, $carrier(v, K(\mathcal{I})) \subseteq S^n$. By Lemma 1, $KN(carrier(v, K(\mathcal{I}))) \subseteq KN(S^n) = In(S^n)$, which concludes the proof. □

[3] Note that the the *Output Complex* $\mathcal{O}$ contains only the decision values

4.3 From a Protocol to Repeated Standard Chromatic Subdivision

The repeated standard chromatic subdivision also have an algorithmic interpretation. Namely, if a protocol exists, then the repeated standard chromatic subdivision of the input complex can be (simplicially) mapped to the output complex, in a way that conforms with the k-set consensus decision problem. To define the adherence to the problem, a map $\Delta : K(\mathcal{I}) \to \mathcal{O}$ is introduced. For any $S^n \in K(\mathcal{I})$, representing an initial configuration, Δ maps S^n to the simplexes of the allowed output configurations.

The following lemma shows that processes can decide only on values that they know. This relates the notion of knowledge to allowed output values for k-set consensus:

Lemma 4. *If there is a protocol for k-set consensus, then there exists an R and a k-coloring $\delta : \chi^R(K(\mathcal{I})) \to \mathcal{O}$, such that for any $v \in \chi^R(K(\mathcal{I}))$, $\delta(v) \in KN(carrier(v, K(\mathcal{I})))$.*

Proof. It was shown by Herlihy and Shavit [3] that if there is a protocol for k-set consensus, then there exists an R and a k-coloring $\delta : \chi^R(K(\mathcal{I})) \to \mathcal{O}$. Consider some vertex $v \in \chi^R(K(\mathcal{I}))$ and denote $S^m = carrier(v, K(\mathcal{I}))$. The results of Herlihy and Shavit shows that $\delta(v)$ is the set of possible decision values of $Proc(v)$ in the $Proc(S^m)$-solo executions with inputs $In(S^m)$.

Pick an arbitrary input configuration $I^n \supseteq S^m$ and consider the solo execution of $Proc(S^m)$ from I^n. Since only processes in $Proc(S^m)$ are allowed to perform steps, $Proc(v)$ decides on a value from $\delta(v)$. The Validity property implies that $\delta(v) \in In(I^n)$. Since I^n is arbitrary, this implies that $\delta(v) \in KN(S^m)$, as needed. □

5 Proof of the Characterization Theorem

5.1 Sufficient Condition

This section presents a constructive proof of the sufficient condition. we show a coloring of $\chi(K(\mathcal{I}))$ when the condition holds. By Theorem 2 this coloring of $\chi(K(\mathcal{I}))$ implies a "single-round" protocol for solving k-set consensus.

Theorem 3. *Assume for every $S^n \in K(\mathcal{I})$, either $|In(S^n)| < n + 1$, or there is a sub-simplex $T^{n-1} \subset S^n$ such that $KN(T^{n-1}) = In(S^n)$. Then there is an n-coloring δ of $\chi(K(\mathcal{I}))$ such that for every $v \in \chi(K(\mathcal{I}))$, $\delta(v) \in KN(carrier(v, K(\mathcal{I})))$.*

Proof. We first show how to color a simplex with less than $n+1$ input values, and then handle simplexes with $n + 1$ input values. We also show that the coloring of adjacent simplexes in $\chi(K(\mathcal{I}))$ agrees on their intersection.

Consider a simplex $S^n \in \chi(K(\mathcal{I}))$.

If $In(S^n) < n + 1$, then define $\delta_{S^n,\emptyset}(x)$ to be the input value of $Proc(x)$ in S^n, for any $x \in \chi(S^n)$. Clearly, $\delta_{S^n,\emptyset}(S^n)$ contains less than $n + 1$ colors.

If $In(S^n) = n+1$, then let $T_0^{n-1} \subset S^n$ be a simplex such that $KN(T_0^{n-1}) = S^n$. Let v be the remaining vertex $S^n \setminus T_0^{n-1}$ (that is, $T_0^{n-1} = face_v(S^n)$) and assume $v = \langle v_p, v_c \rangle$. Then, define

$$\delta_{S^n, T_0^{n-1}}(x) = \begin{cases} v_c & x \in base(S^n) \text{ or } x \in base(T_0^{n-1}) \\ Proc(x)\text{'s input value in } S^n & otherwise \end{cases}$$

Notice that any vertex of the form $\langle v_p, A_\alpha \rangle \in \chi(S^n)$ is mapped to v_c.

Claim. $\left|\delta_{S^n, T_0^{n-1}}(s^n)\right| \le n$, for every $s^n \in \chi(S^n)$.

Proof. Assume by way of contradiction that $\left|\delta_{S^n, T_0^{n-1}}(s^n)\right| = n+1$ for some simplex $s^n \in \chi(S^n)$. Lemma 2 implies that s^n can be written as $w * t^{n-1}$, where $w \in base(S^n)$. Since $\delta_{S^n, T_0^{n-1}}(w) = v_c$, we have that the $(n-1)$-simplex t^{n-1} is colored with n colors and $v_c \notin \delta_{S^n, T_0^{n-1}}(t^{n-1})$. Since all the vertices of $base(S^n)$ are colored with v_c, we have that $t^{n-1} \cap base(S^n) = \emptyset$. Thus, $t^{n-1} \in \chi(U^{n-1})$ for some $U^{n-1} \subset S^n$. There are two possibilities, both leading to a contradiction:
Case 1: $U^{n-1} = T_0^{n-1}$. Using Lemma 2 again implies that t^{n-1} contains one vertex (at least) from $base(T_0^{n-1})$, while $\delta_{S^n, T_0^{n-1}}(base(T_0^{n-1})) = v_c$.
Case 2: $U^{n-1} \neq T_0^{n-1}$. Thus, $U^{n-1} \neq face_v(S^n)$, hence $v_p \in Proc(U^{n-1})$. Since $t^{n-1} \in \chi(U^{n-1})$, $v_p \in Proc(t^{n-1})$, thus $v_c \in \delta_{S^n, T_0^{n-1}}(t^{n-1}) \subseteq \delta_{S^n, T_0^{n-1}}(s^n)$. □

To create δ from the separate maps, we pick for each simplex S^n with $n+1$ different input values a single sub-simplex $T_0^{n-1} \subseteq S^n$, such that $KN(T_0^{n-1}) = In(S^n)$.

$$\delta(v) = \begin{cases} \delta_{S^n, T_0^{n-1}}(v) & v \in S^n \wedge (In(S^n) = n+1) \\ \delta_{S^n, \emptyset}(v) & v \in S^n \wedge (In(S^n) < n+1) \end{cases}$$

To show that δ is a coloring of $\chi(K(\mathcal{I}))$, we now prove that adjacent simplexes agree on the coloring of their intersection.

Claim. Consider two simplexes S_1^n and S_2^n such that $S_1^n \bigcap S_2^n = T^m \neq \emptyset$. Then for any $v \in \chi(T^m)$: $\delta_{S_1^n, x_i}(v) = \delta_{S_2^n, x_i}(v)$, where $x_i = \emptyset$ or $x_i = T_i^{n-1} \subset S_i^n$.

Proof. Let $T^m = S_1^n \bigcap S_2^n$. There are two cases to consider:
Case 1: $m < n-1$. Lemma 3 implies that for every $i = 1, 2$:

1. $\chi(T^m) \cap base(S_i^n) = \emptyset$
2. For any $U_i^{n-1} \subset S_i^n : \chi(T^m) \cap base(U_i^{n-1}) = \emptyset$

From (2) we have that for $i = 1, 2$, $T^m \cap base(x_i) = \emptyset$. Which with (1), implies that for any $v \in T^m$ $\delta_{S_0^n, x_0}(v) = \delta_{S_1^n, x_1}(v) = In(v)$.

Case 2: $m = n-1$. Since $S_1^n \neq S_2^n$, there are vertices $\langle p_{i_0}, \alpha_1 \rangle \in S_1^n \setminus T^m$ and $\langle p_{i_0}, \alpha_2 \rangle \in S_2^n \setminus T^m$ such that $\alpha_1 \neq \alpha_2$. This implies that

$$KN(T^m) = \bigcap_{S^n \supseteq T^m} In(S^n) \subseteq \Big(In(S_1^n) \bigcap In(S_2^n)\Big) = In(T^m)$$

Thus, T^m is not the sub-simplex $T_i^{n-1} \subset S_i^n$ which is used to define $\delta_{S^n, T_i^{n-1}}$. This implies that again, $\delta_{S_0^n, x_0}(v) = \delta_{S_1^n, x_1}(v) = In(v)$. □

The last claim implies that δ is a well defined n-coloring. It remains to prove that given a vertex $v \in \chi(K(\mathcal{I}))$, $\delta(v) \in KN(carrier(v, K(\mathcal{I})))$. If $\delta(v) = Proc(v)$'s input value, which is in $carrier(v, K(\mathcal{I}))$, then the claim holds. Otherwise, the following must hold:

1. $v \in \chi(S^n)$, such that $In(S^n) = n + 1$.
2. A subsimplex $T_0^{n-1} \subsetneq S^n$ was chosen such that $KN(T_0^{n-1}) = In(S^n)$
3. $v \in base(S^n)$ or $v \in base(T_0^{n-1})$.

If $v \in base(S^n)$ then $carrier(v, K(\mathcal{I})) = S^n$, and if $v \in base(T_0^{n-1})$ then $carrier(v, K(\mathcal{I})) = T_0^{n-1}$. In either case, $KN(carrier(v, K(\mathcal{I}))) = In(S^n)$. Since $\delta(v) \in In(S^n)$, the proof follows. □

Theorem 2 implies that there is a protocol solving n-set consensus if inputs are restricted to $\mathcal{I}$.

5.2 Necessary Condition

The necessity of the condition in Theorem 3 relies on Sperner's Lemma:
Lemma 5 (Sperner's Lemma). *Assume there is a simplex $S^n \in K(\mathcal{I})$ such that $In(S^n) = n + 1$ and for any $T^m \subset S^n$, $KN(T^m) = In(T^m)$. Then for any R and any n-coloring δ of $\chi^R(K(\mathcal{I}))$, such that for any $v \in \chi^R(K(\mathcal{I}))$, $\delta(v) \in KN(carrier(v, K(\mathcal{I})))$, there is a simplex $s^n \in \chi^R(S^n)$, such that $|\delta(s^n)| = n+1$.*

Sperner's lemma requires that $KN(T^m) = In(T^m)$ for any $T^m \subseteq S^n$, while the immediate antecedent of our necessary condition states that $KN(T^{n-1}) = In(S^n)$ for every $T^{n-1} \subseteq S^n$. The following lemma proves that that if a set of $m \leq n$ processes know about the input value of some process outside the set, then some set of n processes know all the inputs.

Lemma 6. *Given a simplex S^n such that $|In(S^n)| = n + 1$, and a simplex $T^m \subset S^n$ such that $In(T^m) \subsetneq KN(T^m)$, then there exists a simplex $T^{n-1} \subsetneq S^n$ such that $T^m \subseteq T^{n-1}$ and $KN(T^{n-1}) = In(S^n)$*

Proof. Consider some value $v \in KN(T^m) \setminus In(T^m)$; $\langle p, \alpha \rangle \in S^n$ for some process p. Define $T^{n-1} = face_{\langle p,\alpha \rangle}(S^n)$, that is, $T^{n-1} = S^n \setminus \{\langle p, \alpha \rangle\}$. Clearly, $KN(T^{n-1}) \subseteq In(S^n)$.

Since $v \notin In(T^m)$, it follows that $T^m \subseteq S^n \setminus \{\langle p, \alpha \rangle\} = T^{n-1}$. From Lemma 1 it follows that $KN(T^m) \subseteq KN(T^{n-1})$, therefore, $v \in KN(T^{n-1})$. Since $v \notin In(T^{n-1})$, $In(S^n) \subseteq KN(T^{n-1})$, implying that $KN(T^{n-1}) = In(S^n)$. □

The necessity of our condition follows from the next lemma and Lemma 4.

Lemma 7. *If there exists a simplex $S^n \in K(\mathcal{I})$ such that $|In(S^n)| = n + 1$ and for any $T^{n-1} \subset S^n$, $KN(T^{n-1}) = In(T^{n-1})$. Then for any R, there is no n-coloring of $\chi^R(K(\mathcal{I}))$, such that for any $v \in \chi^R(K(\mathcal{I}))$, $\delta(v) \in KN(carrier(v, K(\mathcal{I})))$.*

Proof. By Lemma 6, $KN(T^m) = In(T^m)$ for any $T^m \subset S^n$, and the claim follows from Sperner's lemma. □

6 Deciding Whether a Condition Is Solvable

We outline an algorithm for deciding whether a specific input restriction allows to solve k-set consensus. The step complexity of the algorithm is $O(|\mathcal{I}|n^{O(1)})$, where $|\mathcal{I}|$ is the number of input configurations.

The main data structure of the algorithm has a value for each $(n-1)$-face of any $S^n \in \mathcal{I}$. For a face T^{n-1}, this value is the input value of a vertex "opposite" to T^{n-1} in S^n $(In(S^n \setminus T^{n-1}))$, if this value is unique; otherwise, it is ∞.

In its first stage, the algorithm goes over all simplexes in $K(\mathcal{I})$ and consider their $(n-1)$-faces. For an $(n-1)$-face T^{n-1} of a simplex $S^n \in K(\mathcal{I})$, let v be the input value of the vertex opposite T^{n-1} in S^n, namely, $In(S^n \setminus T^{n-1})$. If no value is associated with T^{n-1}, the algorithm stores v in the appropriate place. If a value $v' \neq v$ is associated with T^{n-1}, the algorithm stores ∞ in the appropriate place.

In its second stage, the algorithm goes over all simplexes in $K(\mathcal{I})$. The algorithm checks for each simplex $S^n \in K(\mathcal{I})$ whether one of the following conditions hold: (1) S^n does not have $n+1$ different input values, namely, $In(S^n) \leq n$, or (2) for some face T^{n-1} of S^n, the array contains a non-∞ value. $\mathcal{I}$ allows to solve k-set consensus if and only if either (1) or (2) holds for every n-simplex in $K(\mathcal{I})$. Said otherwise, k-set consensus cannot be solved on $\mathcal{I}$ if all the $n-1$-faces of some $S^n \in K(\mathcal{I})$, with more that $n+1$ different input values, have ∞ in their entries.

Each stage of the algorithm requires going over all the simplexes of $\mathcal{I}$, explaining the $O(\mathcal{I})$ term of its step complexity. The exact step complexity of each iteration in a stage, namely, the degree of the poly-n term, depends on the specific implementation of the data structure holding a value for all $n-1$-faces of $K(\mathcal{I})$.

7 Discussion

This paper proves a necessary and sufficient condition on sets of input configurations I, allowing wait-free solution to the n-set consensus problem, using a read-write memory. It is still necessary to provide provide similar conditions for the existence of f-fault tolerant k-set consensus algorithm, for arbitrary k and f.

References

1. Chaudhuri, S.: More choices allow more faults: Set consensus problems in totally asynchronous systems. Information and Computation **103** (1993) 132–158
2. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for t-resilient asynchronous computations. In: Proceedings of the 25th ACM Symposium on Theory of Computing. (1993) 91–100
3. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM **46** (1999) 858–923

4. Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM Journal on Computing **29** (2000) 1449–1483
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
6. Aumann, Y.: Efficient asynchronous consensus with the weak adversary scheduler. In: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing, ACM Press (1997) 209–218
7. Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols. In: Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing. (1983) 27–30
8. Dolev, D., Lynch, N.A., Pinter, S.S., Stark, E.W., Weihl, W.E.: Reaching approximate agreement in the presence of faults. Journal of the ACM **33** (1986) 499–516
9. Attiya, H., Rajsbaum, S.: The combinatorial structure of wait-free solvable tasks. In: Proceedings of the 10th International Workshop on Distributed Algorithms. Number 1151 in Lecture Notes in Computer Science, Springer-Verlag (1996) 321–343 Also Technical Report #CS0924, Department of Computer Science, Technion, December 1997.
10. Taubenfeld, G., Katz, S., Moran, S.: Impossibility results in the presense of multiple faulty porcesses. In: Information and Computation. (1994) 113(2):173–198
11. Taubenfeld, G., Moran, S.: Possibility and impossibility results in a shared memory environment. Acta Informatica **33** (1996) 1–20
12. Friedman, R., Mostefaoui, A., Rajsbaum, S., Raynal, M.: Distributed agreement and its relation with error-correcting codes. (In: Proc. 16th Symposium on Distributed Computing (DISC ’02), these proceedings)
13. Mostefaoui, A., Rajsbaum, S., Raynal, M.: Conditions on input vectors for consensus solvability in asynchronous distributed systems. In: Proceedings of the thirty-third annual ACM symposium on Theory of computing, ACM Press (2001) 153–162
14. Mostefaoui, A., Rajsbaum, S., Raynal, M., Roy, M.: Condition-based protocols for set agreement problems. (In: Proc. 16th Symposium on Distributed Computing (DISC ’02), these proceedings)
15. Borowsky, E., Gafni, E.: Immediate atomic snapshots and fast renaming. In: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing. (1993) 41–52
16. Munkres, J.: Elements of algebraic topology. Addison-Wesley, Menlo Park CA (1984)

The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures

Maurice Herlihy[1], Victor Luchangco[2], and Mark Moir[2]

[1] Computer Science Department, Box 1910, Brown University, Providence, RI 02912
[2] Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803

Abstract. We define the *Repeat Offender Problem* (ROP). Elsewhere, we have presented the first dynamic-sized, lock-free data structures that can free memory to any standard memory allocator—even after thread failures—without requiring special support from the operating system, the memory allocator, or the hardware. These results depend on a solution to the ROP problem. Here we present the first solution to the ROP problem and its correctness proof. Our solution is implementable in most modern shared memory multiprocessors.

1 Introduction

A lock-free data structure is *dynamic-sized* if it can grow and shrink over time. Modern programming environments typically provide support for dynamically allocating and freeing memory (for example, the **malloc** and **free** library calls). A data structure is *lock-free* if it guarantees that after a finite number of steps of any operation on the data structure, *some* operation completes. Lock-free data structures avoid many problems associated with the use of locking, including convoying, susceptibility to failures and delays, and, in real-time systems, priority inversion.

This paper presents a new memory management mechanism for dynamic-sized lock-free data structures. Designing such data structures is not easy: a number of papers describe clever and subtle ad-hoc algorithms for relatively mundane data structures such as stacks [14], queues [11], and linked lists [15,5].

We define an abstract problem, the *Repeat Offender Problem* (ROP), that captures the essence of the memory management problem for dynamic-sized lock-free data structures. Any solution to ROP can be used to permit dynamic-sized lock-free data structure implementations to return unused memory to standard memory allocators. We have formulated this problem to support the use of one or more "worker" threads, perhaps running on spare processors, to do most of the work in parallel with the application's threads.

We present the first solution to the ROP problem, which we call "Pass-the-Buck". In this paper, we focus on the problem statement and a detailed but informal explanation of the algorithm. Elsewhere [6], we show how to apply ROP solutions to achieve the first truly dynamic-sized lock-free data structures, and we evaluate one such implementation. (As we discuss in Section 3, Maged Michael [10] has concurrently and independently developed a similar technique.)

In the remainder of this section, we discuss why dynamic-sized data structures are challenging to implement in a lock-free manner and then briefly summarize previous related work.

Before freeing an object that is part of a dynamic-sized data structure (say, a node of a linked list), we must ensure that no thread will subsequently modify the object. Otherwise, a thread might corrupt an object allocated later that happens to reuse some of the memory used by the first object. Furthermore, in some systems, even read-only accesses to freed objects can be problematic: the operating system may remove the page containing the object from the thread's address space, causing a subsequent access to crash the program because the address is no longer valid [14].

The use of locks makes it relatively easy to ensure that freed objects are not subsequently accessed because we can prevent access by other threads to (parts of) the data structure while removing objects from it. In contrast, without locks, multiple operations may access the data structure concurrently, and a thread cannot determine whether other threads are already committed to accessing the object that it wishes to free (this can only be ascertained by inspecting the stacks and registers of other threads). This is the root of the problem that our work aims to address.

Below we discuss various previous approaches for dealing with the problem described above.[1] One easy approach is to use garbage collection (GC). GC ensures that an object is not freed while any pointer to it exists, so threads cannot access objects after they are freed. This approach is especially attractive because recent experience (e.g., [2]) shows that GC significantly simplifies the design of lock-free, dynamic-sized data structures. However, GC is not available in all languages and environments, and in particular, we cannot rely on GC to implement GC!

Another common approach is to tag values stored in objects. If we access such values only through compare-and-swap (CAS) operations, we can ensure that a CAS applied to a value after the object has been deallocated will fail [14, 12,13]. This approach implies that the memory used for tag values can never be used for anything else. One way to ensure that tag memory is never reused is for the application itself to maintain an explicit pool of objects not currently in use [14,12].

Rather than returning an unused object to the environment's memory management subsystem (say, via the **free** library call), the application places it into its own object pool. An important limitation of application-specific pools is that the application's data structures are not truly dynamic-sized: if the data structures grow large and subsequently shrink, then the application's object pool contains many objects that cannot be coalesced or reused by other applications. In

[1] These approaches are all forms of *type-stable memory* (TSM), defined by Greenwald [4] as follows: "TSM [provides] a guarantee that an object O of type T remains type T as long as a pointer to O exists."

fairness, object pools can provide performance advantages for some applications under some circumstances, bypassing calls to the environment's general-purpose memory allocator.

Elsewhere [6], we show how to eliminate object pools from Michael and Scott's lock-free FIFO queue implementation [12] using the techniques presented in this paper. We also show how to combine the best of both approaches, constructing application-specific object pools that can free excess unused objects to the environment's memory allocator. We also present performance results that show the overhead of making these data structures dynamic-sized is negligible in the absence of contention, and low in all cases. We believe these are the first lock-free, dynamic-sized concurrent data structures that can continue to reclaim memory even after threads fail.

Valois [15] proposed an approach in which the memory allocator maintains reference counts for objects to determine when they can be freed. An object's reference count may be accessed even after the object has been released to the environment's memory allocator. This behavior restricts what the memory allocator can do with released objects (for example, released objects cannot be coalesced). Thus, this approach shares the principal disadvantages of explicit object pools. Valois's approach requires the memory allocator to support certain nonstandard functions, which may make it difficult to port applications to new platforms. Finally, the space overhead for per-object reference counts may be prohibitive. (In [3], we proposed a similar approach that does allow memory allocators to be interchanged, but depends on double compare-and-swap (DCAS), which is not widely supported.)

Our goal is to permit dynamic-sized lock-free data structure implementations to free unneeded memory to the environment's memory allocator through standard interfaces, ensuring that memory allocators can be switched with ease, and that freed memory is not subsequently accessed, permitting the memory allocator to unmap those pages.

Interestingly, the previous work that comes closest to meeting this goal predates the work discussed above by almost a decade. Treiber [14] proposes a technique called "Obligation Passing". The instance of this technique for which Treiber presents specific details is in the implementation of a lock-free linked list supporting search, insert, and delete operations. This implementation allows freed nodes to be returned to the memory allocator through standard interfaces and without requiring any special functionality of the memory allocator. Nevertheless, Obligation Passing employs a "use counter" such that memory is reclaimed only by the "last" thread to access the linked list in any period. As a result, this implementation can be prevented from ever recovering any memory by a failed thread (which defeats one of the main purposes of using lock-free implementations). Another disadvantage of this implementation is that the Obligation Passing code is bundled together with the linked-list maintenance code (all of which is presented in assembly code). Because it is not clear what aspects of the linked-list code are depended upon by the Obligation Passing code, it is difficult to apply this technique to other situations.

$$\begin{tikzcd}
\mathsf{free} \arrow[r, "\mathsf{Arrest(v)}"'] & \mathsf{injail} \arrow[r, "\mathsf{Liberate}(S), v \in S"'] & \mathsf{escaping} \arrow[ll, bend right, "\mathsf{Liberate}\text{ returns } T, v \in T"']
\end{tikzcd}$$

Fig. 1. Tranisition diagram for value v.

2 The Repeat Offender Problem

In this section, we specify the *Repeat Offender Problem* (ROP). Informally, we are given a set of uninterpreted *values*, each of which can be one of three states: free, injail, or escaping. Initially, all values are free. We are given a set of *clients* that interact with values. At any time, a client can Arrest a free value, causing it to become injail, or it can cause an injail value to become escaping. An escaping value can finish escaping and become free again.

Clients *use* values, but must never use a value that is free. A client can attempt to prevent a value v from escaping (becoming free) by "posting a guard" on v. If, however, the guard is posted too late, v may escape anyway. To be safe, a client first posts a guard on v, and then checks whether v is still injail. If so, then a ROP solution must ensure that v does not escape before the guard is removed or redeployed.

Our motivation is to use ROP solutions to allow threads (clients) to avoid dereferencing (using) a pointer (value) to an object that has been freed. In this context, an injail pointer is one that has been allocated (arrested) since it was last freed, and can therefore be used.

It is sometimes possible for a client p to determine independently of ROP that a value it wants to use will remain injail until p uses the value (see [6]). In this case, p can use the value without posting a guard.

To support these interactions, ROP solutions provide the following procedures. A thread *posts* a guard g on a value v by invoking PostGuard(g,v), which removes the guard from any value it previously guarded. (A special **null** value is used to *stand down* the guard, that is, to remove the guard from the previously guarded value without posting the guard on a new value). A thread causes a set of values S to begin *escaping* by invoking Liberate(S); the application must ensure that each value in S is injail before this call, and the call causes each value to become escaping. The Liberate procedure returns a (possibly different) set of escaping values causing them to be *liberated* (each returned value becomes free). These transitions are summarized in Figure 1. Finally, a thread can check whether a value v is injail by invoking IsInJail(v); if this invocation returns *true*, then v was injail at some point during the invocation (the converse is not necessarily true, as explained later). Although the Arrest action is specific to the application, the ROP solution must be aware of arrests in order to detect when a free value becomes injail.

If a guard g is posted on a value v, and v is injail at some time t after g is posted on v and before g is subsequently stood down or reposted on a different value, then we say that g *traps* v from time t until g is stood down or reposted. The main correctness condition for ROP is that it does not allow a value to escape (i.e., become free) while it is trapped.

We now turn our attention to some additional important but mundane details, together with a formal specification of ROP. In some applications (for example, [6]), a client must guard multiple values at the same time. Clients may *hire* and *fire* guards by invoking the HireGuard and FireGuard procedures. Applications' use of guards is expected to follow obvious well-formedness properties, such as ensuring that a thread posts only those guards it employs.

A formal definition of ROP is given by the I/O automaton ([9]) shown in Figure 2, explained below.

Notational Conventions. Unless otherwise specified, p and q denote clients (threads) from P, the set of all clients (threads); g denotes a guard from G, the set of all guards; v denotes a value from V, the set of all values, and S and T denote sets of values (i.e., subsets of V). We assume that V contains a special **null** value that is never used, arrested, or liberated. □

The automaton consists of a set of *environment actions* and a set of *ROP output actions*. Each action consists of a *precondition* for performing the action and the *effect* on state variables of performing the action. Most environment actions are invocations of ROP operations, and are paired with matching ROP output actions that represent the system's response to the invocations. For example, the $\mathsf{PostInv}_p(g, v)$ action models client p invoking PostGuard(g,v), and the $\mathsf{PostResp}_p()$ action models the completion of this procedure, and similarly for HireGuard(), FireGuard(), and Liberate(). Finally, the Arrest(v) action models the environment (application) arresting value v.

The state variable *status*[v] records the current status of value v: free, injail, or escaping. Transitions between status values are caused by calls to and returns from ROP procedures, as well as by the application-specific Arrest action. The *post* variable maps each guard to the value (if any) it currently guards. The pc_p variable models the control flow (program counter) of client p, for example ensuring that p does not invoke a procedure before the previous invocation completes; pc_p also sometimes encodes procedure parameters. The $guards_p$ variable represents the set of guards currently employed by client p. The *numEscaping* variable is an auxiliary variable used to specify nontriviality properties, as discussed later. Finally, *trapping* maps each guard g to a boolean value that is true iff g has been posted on some value v, and has not subsequently been reposted or stood down, and at some point since the guard was posted on v, v has been injail (i.e., it captures the notion of guard g trapping the value on which it has been posted). This is used by the LiberateResp action to determine whether v can be returned. (Recall that a value should not be returned if it is trapped.) Preconditions on the invocation actions specify assumptions about the circumstances under which the application invokes the corresponding ROP procedures. Most of these preconditions are mundane well-formedness conditions such as the requirement that a client posts only guards that it currently employs. The

actions

Environment	ROP output
$\mathsf{HireInv}_p()$	$\mathsf{HireResp}_p(g)$
$\mathsf{FireInv}_p(g)$	$\mathsf{FireResp}_p()$
$\mathsf{PostInv}_p(g, v)$	$\mathsf{PostResp}_p()$
$\mathsf{IsInJailInv}_p(v)$	$\mathsf{IsInJailResp}_p(b)$
$\mathsf{LiberateInv}_p(S)$	$\mathsf{LiberateResp}_p(S)$
$\mathsf{Arrest}(v)$	

state variables

```
For each client $p \in P$:
  $pc_p$: {idle, hire, fire, post($g, v$),
           injail($v$), liberate} **init** idle
  $guards_p$: set of guards **init** empty
For each value $v \in V$:
  $status[v]$: {injail, escaping, free}
                                   **init** free
For each guard $g \in G$:
  $post[g] : V$ **init null**;
  $trapping[g]$ : **bool init** false;
$numEscaping$: **int init** 0
```

transitions

```
$\mathsf{HireInv}_p()$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p \leftarrow \mathsf{hire}$

$\mathsf{FireInv}_p(g)$
Pre: $pc_p = \mathsf{idle}$
     $g \in guards_p$
     $post[g] = \mathbf{null}$
Eff: $pc_p \leftarrow \mathsf{fire}$
     $guards_p \leftarrow guards_p \quad \{g\}$

$\mathsf{PostInv}_p(g, v)$
Pre: $pc_p = \mathsf{idle}$
     $g \in guards_p$
Eff: $pc_p \leftarrow \mathsf{post}(g, v)$
     $post[g] \leftarrow \mathbf{null}$
     $trapping[g] \leftarrow$ false

$\mathsf{IsInJailInv}_p(v)$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p \leftarrow \mathsf{injail}(v)$

$\mathsf{LiberateInv}_p(S)$
Pre: $pc_p = \mathsf{idle}$
     for all $v \in S$,
         $v \neq \mathbf{null}$ and $status[v] = \mathsf{injail}$
Eff: $pc_p \leftarrow \mathsf{liberate}$
     $numEscaping \leftarrow numEscaping + |S|$
     for all $v \in S$, $status[v] \leftarrow \mathsf{escaping}$

$\mathsf{Arrest}(v)$
Pre: $status[v] = \mathsf{free}$
     $v \neq \mathbf{null}$
Eff: $status[v] \leftarrow \mathsf{injail}$
     for all $g$ such that $post[g] = v$,
         $trapping[g] \leftarrow$ true

$\mathsf{HireResp}_p(g)$
Pre: $pc_p = \mathsf{hire}$
     $g \in G$
     $g \notin \bigcup_q guards_q$
Eff: $pc_p \leftarrow \mathsf{idle}$
     $guards_p \leftarrow guards_p \cup \{g\}$

$\mathsf{FireResp}_p()$
Pre: $pc_p = \mathsf{fire}$
Eff: $pc_p \leftarrow \mathsf{idle}$

$\mathsf{PostResp}_p()$
Pre: for some $g, v$, $pc_p = \mathsf{post}(g, v)$
Eff: $pc_p \leftarrow \mathsf{idle}$
     $post[g] \leftarrow v$
     $trapping[g] \leftarrow (status[v] = \mathsf{injail})$

$\mathsf{IsInJailResp}_p(b)$
Pre: for some $v$, $pc_p = \mathsf{injail}(v)$
     $b \Rightarrow (status[v] = \mathsf{injail})$
Eff: $pc_p \leftarrow \mathsf{idle}$

$\mathsf{LiberateResp}_p(S)$
Pre: $pc_p = \mathsf{liberate}$
     for all $v \in S$,
         $status[v] = \mathsf{escaping}$
         and for all $g \in \bigcup_q guards_q$,
             $(post[g] \neq v$ or $\neg trapping[g])$
Eff: $pc_p \leftarrow \mathsf{idle}$
     $numEscaping \leftarrow numEscaping \quad |S|$
     for all $v \in S$, $status[v] \leftarrow \mathsf{free}$
```

Fig. 2. I/O Automaton specifying the Repeat Offender Problem.

precondition for LiberateInv captures the assumption that the application passes only injail values to Liberate, and the precondition for the Arrest action captures the assumption that only free values are arrested. The application designer must determine how these guarantees are made.

Preconditions on the response actions specify the circumstances under which the ROP procedures can return. Again, most of these preconditions are mundane. The interesting case is the precondition of LiberateResp, which states that Liberate can return a value only if it has been passed to (some invocation of) Liberate, it has not subsequently been returned by (any invocation of) Liberate, and no guard g has been continually guarding the value since the last time it was injail (this property is captured by *trapping*$[g]$).

Liveness Properties

As specified so far, a ROP solution in which Liberate always returns the empty set, or simply does not terminate, is correct. Clearly, such solutions are unacceptable because each escaping value represents a resource that will be reclaimed only when the value is liberated (returned by some invocation of Liberate). One might be tempted to specify that every value passed to a Liberate operation is eventually returned by some Liberate operation. However, without special operating system support, it is not possible to guarantee such a strong property if threads can fail. Rather than proposing a single nontriviality condition, we instead discuss a range of alternative conditions.

The state variable *numEscaping* counts the number of values currently escaping (that is, passed to some invocation of Liberate and not subsequently returned from any invocation of Liberate). If we require a solution to ensure that *numEscaping* is bounded by some function of application-specific quantities, we exclude the trivial solution in which Liberate always returns the empty set. However, because this bound necessarily depends on the number of concurrent Liberate operations, and the number of values each Liberate operation is invoked with, it does not exclude the solution in which Liberate never returns.

A combination of a boundedness requirement and some form of progress requirement on Liberate operations seems to be the most appropriate way to specify the nontriviality requirement. We later prove that the Pass The Buck algorithm provides a bound on *numEscaping* that depends on the number of concurrent Liberate operations. Because the bound (necessarily) depends on the number of concurrent Liberate operations, if an unbounded number of threads fail while executing Liberate, then an unbounded number of values can be escaping. We emphasize, however, that our implementation does *not* allow failed threads to prevent values from being freed in the future, as Treiber's approach does [14].

Our Pass The Buck algorithm has two more desirable properties. First, the Liberate operation is wait-free (that is, it completes after a bounded number of steps, regardless of the timing behaviour of other threads). This is useful because it allows us to calculate an upper bound on the amount of time Liberate will take to execute, which is useful in determining how to schedule Liberate work.

Finally, our algorithm has a property we call *value progress*. Roughly, this property guarantees that a value does not remain escaping forever provided Liberate is invoked "enough" times (unless a thread fails).

Modular Decomposition

A key contribution of this paper is the insight that an effective way to solve ROP in practice is to separate the implementation of the IsInJail operation from the others.

In our experience using ROP solutions to implement dynamic-sized lock-free data structures [6], values are used in a manner that allows threads to determine whether a value is injail *with sufficient accuracy for the particular application.* As a concrete example, when values represent pointers to objects that are part of a concurrent data structure, these values become injail (allocated) before the objects they refer to become part of the data structure, and are removed from the data structure before being passed to Liberate. Thus, simply observing that an object is still part of a data structure is sufficient to conclude that a pointer to it is injail.

Because we intend ROP solutions to be used with application-specific implementations of IsInJail, the specification of this operation is somewhat weak: it permits an implementation of IsInJail that always returns *false*. However, such an implementation would be useless, usually because it would not guarantee the required progress properties of the application that uses it. Because the circumstances under which IsInJail can and should return *true* depend on the application, we retain the weak specification of IsInJail, and leave it to application designers to provide implementations of IsInJail that are sufficiently strong for their applications. (Note that an integrated, application-independent implementation of this operation, while possible, would be expensive: it would have to monitor and synchronize with all actions that potentially affect the status of each value.)

This proposed modular decomposition suggests a methodology for implementing dynamic-sized lock-free objects: use an "off-the-shelf" implementation of an ROP solution for the HireGuard, FireGuard, PostGuard, and Liberate operations, and then exploit specific knowledge of the application to design an optimized implementation of IsInJail. More precisely, we decompose the ROP I/O automaton into two component automata: the *ROPlite* automaton, and the *InJail* automaton. *ROPlite* has the same environment and output actions as ROP, except for IsInJailInv and IsInJailResp. *InJail* has input action IsInJailInv and output action IsInJailResp. In addition, the *InJail* automaton "eavesdrops" on *ROPlite*: all environment and output actions of *ROPlite* are input actions of *InJail* (though in many cases the implementation of the *InJail* automata will ignore these inputs because it can determine whether a value is injail without them, as discussed above).

We present our Pass The Buck algorithm, which implements *ROPlite* in a simple and practical way, in Section 3.

3 One Solution: Pass the Buck

In this section, we describe one ROP solution. Our primary goal when designing this solution was to minimize the performance penalty to the application when no values are being liberated. That is, the PostGuard operation should be implemented as efficiently as possible, perhaps at the cost of a more expensive Liberate operation. Such solutions are desirable for at least two reasons. First, PostGuard is necessarily invoked by the application, so its performance always impacts application performance. On the other hand, Liberate work can be done by a spare processor, or by a background thread, so that it does not directly impact application performance. Second, solutions that optimize PostGuard performance are desirable for scenarios in which values are liberated infrequently, but we must retain the ability to liberate them. An example is the implementation of a dynamic-sized data structure that uses an object pool to avoid allocating and freeing objects under "normal" circumstances, but can free elements of the object pool when it grows too large. In this case, no liberating is necessary while the size of the data structure is relatively stable.

Preliminaries. The Pass-the-Buck algorithm is presented in pseudocode, which should be self-explanatory. For convenience, we assume a shared-memory multiprocessor with sequentially consistent memory [8].[2] We further assume that the multiprocessor supports a compare-and-swap (CAS) instruction that accepts three parameters: an *address*, an *old* value, and a *new* value. The CAS instruction atomically compares the contents of the address to the old value, and, if they are equal, stores the new value at the address and returns *true*. If the comparison fails, no changes are made to memory, and the CAS instruction returns *false*. □

The Pass-the-Buck algorithm is shown in Figure 3. The GUARDS array allocates guards to threads. The POST array consists of one location per guard, holding the value that guard is currently assigned to guard, if any, and **null** otherwise. The Liberate operation uses the HANDOFF array to "hand off" responsibility for a value to a later Liberate operation if that value has been trapped by a guard. (For ease of exposition, we assume a bound MG on the number of guards simultaneously employed. It is not difficult to eliminate this assumption by replacing the static GUARDS, POST and HANDOFF arrays with a linked list; guards can be added as needed by appending nodes to the end of the list.)

The HireGuard and FireGuard procedures are based on a long-lived renaming algorithm [1]. Each guard g has an entry GUARDS[g], initially false. Thread p hires guard g by atomically changing GUARDS[g] from *false* (unemployed) to *true* (employed); p attempts hiring each guard in turn until it succeeds (lines 2 and 3). The FireGuard procedure simply sets the guard back to *false* (line 7).

[2] We have implemented our algorithms for multiprocessor architectures based on SPARC® processors that provide Total Store Ordering (TSO) [16], a memory model weaker than sequential consistency that requires additional memory barrier instructions.

The HireGuard procedure also maintains the shared variable MAXG, used by the Liberate procedure to determine how many guards to consider. The Liberate operation considers every guard for which a HireGuard operation has completed. In the loop at lines 4 and 5, each HireGuard operation ensures that MAXG is at least the index of the guard returned.

To make PostGuard as efficient as possible, it is implemented as a single store of the value to be guarded in the specified guard's POST entry (line 9).

```
// handoff entry (CAS target)
typedef struct { value val; int ver } entry;

const int MG = ... // max num guards

// shared variables
bool GUARDS[MG];    // initially false
value POST[MG];     // initially null
entry HANDOFF[MG]; // initially {null,0}
int MAXG= 0;

int HireGuard() {
1  int i = 0, max;
2  while (!CAS(&GUARDS[i],false,true))
3    i++;
4  while ((max = MAXG) < i)
5    CAS(&MAXG,max,i);
6  return i;
}

void FireGuard(int i) {
7  GUARDS[i] = false;
8  return
}

void PostGuard(int i, value v) {
9  POST[i] = v;
10 return
}
```

```
value set Liberate(value set vs) {
11 int i = 0;
12 while (i <= MAXG) {
13   int attempts = 0;
14   entry h = HANDOFF[i];
15   value v = POST[i];
16   if (v != null && vs.search(v)) {
17     while (true) {
18       if (CAS(&HANDOFF[i],
              h, ⟨v, h.ver+1⟩)) {
19         vs.delete(v);
20         if (h.val != null)
              vs.insert(h.val);
21         break;
         }
22       attempts++;
23       if (attempts == 3) break;
24       h = HANDOFF[i];
25       if (attempts == 2
              && h.val != null)
           break;
26       if (v != POST[i]) break;
       }
27   } else {
28     if (h.val != null && h.val != v)
29       if (CAS(&HANDOFF[i],
              h, ⟨null, h.ver+1⟩))
30         vs.insert(h.val);
     }
31   i++;
   }
32 return vs;
}
```

Fig. 3. I/O Automaton specifying the Repeat Offender Problem.

The most interesting part of the Pass-the-Buck algorithm lies in the Liberate procedure. Recall that Liberate should return a set of values that have been passed to Liberate and have not since been returned by Liberate (i.e., escaping values), subject to the constraint that Liberate cannot return a value that has been continuously guarded by the same guard since some point when it was injail (i.e., Liberate must not return trapped values). The Liberate procedure maintains a set of escaping values, initially those values passed to it. It checks each guard, removing any values in the set that may be trapped and leaving them behind for later Liberate operations. It also adds to its set values that were left behind by previous Liberate operations but are no longer trapped. After it has checked all guards, it returns the values that remain in its value set.

Suppose thread p is executing a call to Liberate, value v is in the p's value set, and guard g is guarding v. To ensure that Liberate is wait-free, p must either determine that g is not trapping v, or remove v from its value set. To guarantee value progress, if p removes v from its set, then it must ensure that v will be examined by later calls to Liberate. The interesting aspects of the Pass-the-Buck algorithm concern how threads determine that a value is not trapped, and how they store values while keeping space overhead for stored values low.

The loop at lines 12 through 31 iterates over all guards ever hired. For each guard, if p cannot determine for some value v in its set that v is not trapped by that guard, then p attempts to "hand off" that value (there can be at most one such value per guard). If p succeeds (line 18), it removes v from its set (line 19) and proceeds to the next guard (lines 21 and 31). Also, as explained in more detail below, p might simultaneously add to its set a value handed off previously by another Liberate operation; it can be shown that any such value is not trapped by that guard. If p fails to hand v off, then it retries. If it fails repeatedly, it can be shown that v is not trapped by that guard, so p can move on to the next guard without removing v from its set (lines 22 and 25). When p has examined all guards (see line 12), it can safely return any values remaining in its set (line 32).

The following lemma (proved in the full paper) is helpful for understanding why the algorithm works.

Single Location Lemma: Each escaping value v is stored at a single guard or is in the value set of a single Liberate operation (but not both). Also, no non-escaping value is in any of these locations.

At lines 15 and 16, p determines whether the value currently guarded by g (if any) is in its set. If so, p executes the loop at lines 17 through 26 in order to either determine that the value—call it v—is not trapped, or to remove v from its set. To avoid losing v in the latter case, p "hands off" v by storing it in the HANDOFF array. Each entry of this array consists of a value and a version number. Version numbers are incremented with each modification of the entry for reasons discussed below. We assume version numbers are large enough that they can be considered unique for practical purposes (see [13] for discussion and justification).

Because at most one value is trapped by guard g at any time, a single location HANDOFF[g] for each guard g is sufficient. To see why, observe that p attempts to hand off v only if v is in p's value set. If a value w was previously handed off (i.e., it is in HANDOFF[g]), then the Single Location Lemma implies that $v \neq w$, so w is not trapped by g. Thus, p can add w to its value set.

To hand v off, p uses a CAS operation to attempt to replace the value previously stored in HANDOFF[g] with v (line 18). If the CAS succeeds, p adds the replaced value to its set (line 20). We explain below why it is safe to do so. If the CAS fails, then p rereads HANDOFF[g] (line 24) and retries the hand-off. The algorithm is wait-free because the loop completes after at most three CAS operations (lines 13, 22, and 23).

As described so far, p picks up a value from HANDOFF[g] only if its value set contains a value that is guarded by guard g. To ensure that a value does not remain in HANDOFF[g] forever (violating the value progress property), if p does not need to remove a value from its set, it still picks up any previously handed off value and replaces it with **null** (lines 28 through 30).

We now consider each of the ways p can break out of the loop at lines 17 through 26, and explain why it is safe to do so. Suppose p exits the loop after a successful CAS at line 18. As described earlier, p removes v from its set (line 19), adds the previous value in HANDOFF[g] to its set (line 20), and moves on to the next guard (lines 21 and 31). Why is it safe to take the previous value w of HANDOFF[g] to the next guard? The reason is that we read POST[g] (line 15 or 26) between reading HANDOFF[g] (line 14 or 24) and attempting the CAS at line 18. Because each modification to HANDOFF[g] increments its version number field, it follows that w was in HANDOFF[g] when p read POST[g]. Also, recall that $w \neq v$ in this case. Therefore, when p read POST[g], w was not guarded by g. Furthermore, because w remained in HANDOFF[g] from that moment until the CAS, w cannot become trapped in this interval (because a value can become trapped only while it is injail, and all values in the HANDOFF array and in the sets of Liberate operations are escaping). The same argument explains why it is safe to pick up the value replaced by **null** at line 29.

It remains to consider how p can break out of the loop *without* performing a successful CAS. In each case, p can infer that v is not trapped by g, so it can give up on its attempt to hand off v. If p breaks out of the loop at line 26, then v is not trapped by g at that moment simply because it is not guarded by g. The other two places where p may break out of the loop (lines 23 and 25) occur only after the thread has executed several failed CAS operations. The full paper [7] contains a detailed case analysis showing that in each case, v is not trapped.

Discussion

The Pass-the-Buck algorithm satisfies the value progress property because a value cannot remain handed off at a particular guard forever if Liberate is executed enough times. If a value v is handed off at guard g, then the first Liberate operation to begin processing g after v is not trapped by g will ensure that v is picked up and taken to the next guard (or returned from Liberate if g is the last guard), either by that Liberate operation or by a concurrent Liberate operation.

As noted earlier, Michael [10] has independently and concurrently developed a solution to a very similar problem. Michael's algorithm buffers to-be-freed values so that it can control the number of values passed to *Scan* (his equivalent of the Liberate operation) at a time. This has the disadvantage that there are usually $O(GP)$ values that could potentially be freed, but are not (where G is the number of "hazard pointers"—the equivalent of guards—and P is the number of participating threads). However, this technique allows him to achieve a nice amortized bound on time spent per value freed. He also has weaker requirements for *Scan* than we have for Liberate; in particular, *Scan* can return some values to the buffer if they cannot yet be freed. This admits a very simple solution that uses only read and write primitives (recall that ours requires CAS) and allows several optimizations. However, it also means that if a thread terminates while values remain in its buffer, then those values will never be freed, so his algorithm does not satisfy the *value progress* property. (Michael alludes to possible methods for handing off values to other threads but does not explain how this can be achieved efficiently and wait-free using only reads and writes.) This is undesirable because a single value might represent a large amount of resources, which would never be reclaimed in this case. The number of such values is bounded by $O(GP)$. We can perform the same optimizations and achieve the same amortized bound under normal operation, while still retaining the *value progress* property (although we would require threads to invoke a special wait-free operation before terminating to achieve this). In this case, our algorithm would perform almost identically to Michael's (with a slight increase in overhead upon thread termination), but would of course share the disadvantages discussed above, except for the lack of value progress.

Elsewhere [6], we present a dynamic-sized lock-free FIFO queue constructed by applying our ROP solution to the non-dynamic-sized implementation of Michael and Scott [12] together with the non-dynamic-sized freelist of Treiber [14]. Experiments show that the overhead of the dynamic-sized FIFO queue over the non-dynamic-sized one of [12] is negligible in the absence of contention, and low in all cases.

Michael has shown how to apply his technique to achieve dynamic-sized implementations of a number of different data structures, including queues, double-ended queues, list-based sets, and hash tables (see [10] for references). Because the interfaces and safety properties of our approaches are almost identical, those results can all be achieved using any ROP solution too. In addition, using Pass-the-Buck would allow us to achieve value progress in those implementations. Michael also identified a small number of implementations to which his method is *not* applicable. In some cases, this may be because Michael's approach is restricted to use a fixed number of hazard pointers per thread; in contrast, ROP solutions provide for dynamic allocation of guards. Furthermore, we have presented [6] a general methodology based on any ROP solution that can be applied to achieve dynamic-sized versions of these data structures too. This methodology is based on reference counts, and therefore has disadvantages such as space and time overhead, and inability to reclaim cyclic garbage.

4 Concluding Remarks

We have defined the Repeat Offenders Problem (ROP), and presented one solution to this problem. Such solutions provide a mechanism for supporting memory management in lock-free, dynamic-sized data structures. The utility of this mechanism has been demonstrated elsewhere [6], where we present what we believe are the first dynamic-sized, lock-free data structures that can continue to reclaim memory even if some threads fail (although Maged Michael [10] has independently and concurrently achieved such implementations, as discussed in Section 3).

By specifying the ROP as an abstract and general problem, we allow for the possibility of using different solutions for different applications and settings, without the need to redesign or reverify the data structure implementations that employ ROP solutions. We have paid particular attention to allowing much of the work of managing dynamically allocated memory to be done concurrently with the application, using additional processors if they are available.

The ideas in this paper came directly from insights gained and questions raised in our work on lock-free reference counting [3]. This further demonstrates the value of research that assumes stronger synchronization primitives than are currently widely supported.

Future work includes exploring other ROP solutions, and applying ROP solutions to the design of other lock-free data structures. It would be particularly interesting to explore the various ways for scheduling Liberate work.

Acknowledgments. We thank Steve Heller, Paul Martin, and Maged Michael for useful feedback, suggestions, and discussions. In particular, Steve Heller suggested formulating ROP to allow the use of "spare" processors for memory management work.

References

1. J. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11:1–20, 1997. A preliminary version appeared in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, 1994, pp. 141-150.
2. D. Detlefs, C. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. Steele. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73, 2000.
3. D. Detlefs, P. Martin, M. Moir, and G. Steele. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.
4. M. Greenwald. *Non-Blocking Synchronization and System Design.* PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, August 1999.
5. T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing*, 2001. To appear.

6. M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Dynamic-sized lock-free data structures. Technical Report TR-2002-110, Sun Microsystems Laboratories, 2002.
7. M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. Technical Report TR-2002-112, Sun Microsystems Laboratories, 2002.
8. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.
9. N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report CWI-Quarterly, 2(3), Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1989.
10. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Distributed Computing*, 2002.
11. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, pages 267–276, 1996.
12. M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.
13. M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, 1997.
14. R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.
15. J. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Dsitributed Computing*, pages 214–22, 1995. See `http://www.cs.sunysb.edu/~valois` for errata.
16. D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9.* PTR Prentice Hall, Englewood Cliffs, NJ 07632, USA, 1994.

On the Impact of Fast Failure Detectors on Real-Time Fault-Tolerant Systems

Marcos K. Aguilera[1], Gérard Le Lann[2], and Sam Toueg[3]

[1] HP Systems Research Center, 1501 Page Mill Road, Palo Alto, CA, 94304, USA,
aguilera@hpl.hp.com
[2] INRIA Rocquencourt, BP 105, F-78153 Le Chesnay Cedex, France,
Gerard.Le_Lann@inria.fr
[3] Department of Computer Science, University of Toronto, Toronto, Canada,
sam@cs.toronto.edu

Abstract. We investigate whether fast failure detectors can be useful — and if so by how much — in the design of real-time fault-tolerant systems. Specifically, we show how fast failure detectors can speed up consensus and fault-tolerant broadcasts, by providing fast algorithms and deriving some matching lower bounds, for synchronous systems with crashes. These results show that a fast failure detector service (implemented using specialized hardware or expedited message delivery) can be an important tool in the design of real-time mission-critical systems.

1 Introduction

Consensus and various types of fault-tolerant broadcast — such as atomic and reliable broadcast — are central paradigms for fault-tolerant distributed computing. Unfortunately, the implementation of these paradigms can be expensive, particularly for real-time systems which are concerned about worst-case time behavior. For instance, consensus requires in the worst-case time $D(1 + f_{max})$ in a synchronous system, where D is the *maximum* message delay and f_{max} is the maximum number of process that may crash. This worst-case time complexity, which also applies to several types of fault-tolerant broadcasts, hinders the widespread use of these paradigms in many applications. This is especially problematic in real-time applications, which are particularly concerned about worst-case scenarios.

In this paper we explore a new approach, namely, the use of fast failure detection, to circumvent this lower bound and obtain faster algorithms for consensus and fault-tolerant broadcasts. Our results show that this approach is particularly suitable to the design of real-time mission-critical systems, where worst-case performance is crucial and where we can use special mechanisms to implement fast failure detection.

There are several ways one can achieve fast failure detection. One way is to use *specialized hardware*. For example, in some mission-critical systems, such as spaceborne ones [19], and in some Tandem systems, failure detection has been implemented in hardware and is very fast.

Another way to achieve fast failure detection is to use expedited message delivery. In fact, researchers in the real-time community have long considered ways to expedite the delivery of selected messages that are critical to the performance of applications, e.g., control or clock synchronization messages. One common way to do so is to tag the urgent messages so that they can be processed ahead of the others in the network and in the waiting queues of the sender and receiver (e.g., [16,17]). Another way to expedite urgent messages is to use a *physically separate medium* to carry them, as in [5] or in control networks of process control and transportation systems, or in signalling channels of telecommunications systems. Whatever mechanism is used, these approaches boil down to having two types of messaging services: a regular one for most application-level messages, and an expedited service for a small set of urgent, time-critical messages. In some of these systems, there can be a large difference between the *maximum* message delays of regular and urgent messages; in fact, they can be orders of magnitude apart. For example, in some local area networks, the highest priority messages are 8 to 30 times faster than the lowest priority ones, assuming only one stage of waiting queues [21]. Recent work [13] has shown how to use expedited message delivery to build a fast failure detector with an analytically-quantified maximum detection time d that is much smaller than the maximum delay D of regular messages.[1]

In this paper, we investigate whether fast failure detectors can speed up consensus and/or various types of fault-tolerant broadcasts, and if so, by how much. It is not entirely obvious that one can take advantage of a fast failure detector to speed up some or all of these problems. For example, a natural attempt is to use the fast failure detector to simulate synchronous rounds, and the hope is that the failure detector will shorten some of these rounds, namely, those that have failures. This in turn would speed up existing round-based algorithms in the worst-case runs, which are exactly those that have failures. But this naive idea does not work: every round could have one correct sender whose message to every process takes the maximum delay D to arrive; thus every round could take D.

We show that fast failure detectors can indeed be used to significantly speed up consensus and fault-tolerant broadcasts, and we also provide some tight lower bounds. Specifically, our results are as follows. We consider a synchronous system with process crashes, where the maximum message delay is some known constant D, and processes have access to a failure detector that detects crashes within some known constant d, where $d \leq D$. We first give an algorithm for consensus that reaches decision in time $D + f_{max}d$, where f_{max} is the maximum number of processes that may crash.[2] We then give an *early-deciding* [9] algorithm for consensus that reaches decision within time $D + fd$, where f is the number of crashes that actually occur (in many runs $f < f_{max}$). Note that in failure-free runs, decision occurs within time D — the fastest possible. In addition, our consensus algorithms are message efficient: they send at most $(f + 1)n$ point-to-point messages.

The consensus algorithms in this paper are time optimal: we prove that in a synchronous system with fast failure detection, consensus requires at least time $D + fd$.

[1] Here, we stress that D is the *maximum* and not the *average* message delay. In many systems, the maximum message delay of regular messages is *orders of magnitude* greater than their average delay.

[2] Actually, this algorithm solves *uniform* consensus. In fact all algorithms in this paper solve the uniform version of the problem in question.

This proof is novel in two respects: it is the first lower bound proof for synchronous systems equipped with failure detection, and moreover it uses a new technique to deal with continuous-time synchronous systems (as opposed to round-based ones), which we believe to be applicable in other contexts.

We then consider several forms of fault-tolerant broadcasts [12], and for each of them we give *early-delivering* algorithms, i.e., algorithms where message delivery time is proportional to f not f_{max}. Specifically, we first give an algorithm for terminating reliable broadcast that delivers messages within time $D + fd$. We next present a reliable broadcast algorithm that delivers within time $2D + (f - 1)d$. Finally, we describe an atomic broadcast algorithm that delivers within time $2D + (f - 1)d$. All these broadcast algorithms deliver within time D in the failure-free case — this is the best possible.[3] Moreover, the algorithms are message efficient: they use at most $(f + 2)n$ point-to-point messages. Note it is surprising that we can solve atomic broadcast in time $2D + (f - 1)d$. For instance, once the diffusion of a broadcast message has started, it is not clear that a process should be allowed to stop it, lest the message be delivered by some but not all correct processes. But continuing the diffusion could lead to a message relay chain of size f, and hence the broadcast time would be $(f + 1)D > 2D + (f - 1)d$.

All the results above assume that messages are not lost, but we show how to extend them to allow the following type of message losses. In most systems, a process sends a message by placing it in an outgoing waiting queue before it can be transmitted. If the machine hosting the process crashes, the waiting queue is wiped out and the message is lost. We model such losses by defining a parameter δ such that, if the sender crashes within δ time of sending a message, then this message may be lost (δ can be interpreted as the maximum time a message may spend in the outgoing queue; note that if $\delta = 0$ we fall back to the case without message losses). We then describe an early-deciding consensus algorithm that takes at most $D + f(d + \delta)$ time to decide. We next show that consensus requires at least time $D + f(d + \delta)$ — and thus our algorithm is time optimal. Finally, we give message-efficient early-delivering algorithms for the fault-tolerant broadcasts described above.

In summary, the contributions of this paper are the following:

– This is the first paper to study the impact of fast failure detectors on real-time fault-tolerant synchronous systems. In such systems, we focus on solving consensus and various types of fault-tolerant broadcasts.
– We give fast and simple algorithms for consensus, terminating reliable broadcast, reliable broadcast and atomic broadcast. Our algorithms are early-deciding or early-delivering. Their time complexity is $O(D + fd)$ and their message complexity is $O(fn)$ where the constants are all very small (1 or 2).
– We show that our consensus algorithms are time optimal, by proving a lower bound of time $D + fd$. (This bound carries over to all our broadcast problems.) The proof uses new techniques to handle failure detection in synchronous systems and to deal with continuous time.
– We extend our results to tolerate message losses that occur when the sender crashes within a certain period after sending.

[3] To achieve delivery in time D, two of our algorithms require some extra optimizations, as we describe in the paper.

We believe that the above results provide guidance on the design of future real-time fault-tolerant systems by showing that implementing a fast failure detection mechanism is indeed useful, and by quantifying the speed-up that can be obtained.

A remark is now in order. As we noted earlier, fast failure detection can be achieved by using specialized hardware or by expediting selected messages. In the latter case, one may wonder why not expedite directly the consensus or broadcast messages, instead of the failure detector's. The reason is that this method would not scale: the number of selected messages that can be expedited is limited, and as we increase the number of concurrent consensus and/or broadcasts this limit can be exceeded. In contrast, a failure detector can be implemented as a service that is *shared* among concurrent applications running on the same set of machines, and one can ensure that the number of messages that the failure detector sends is dependent on the number of machines and is relatively independent of the number of concurrent applications (this assumes that each failure detector message can hold a large number of process id's). Such a shared failure detector service has recently been designed and built [8,15].

Related work. Failure detectors have been used to solve a variety of problems in different environments (e.g., [6,4,11,1,14,2,8]). The idea to use priorities or deadlines to tag messages and to process them faster in the waiting queues has long been explored in scheduling theory and queueing theory (e.g., [10,22]). This idea is used in [13] to build a fault-tolerant distributed system for real-time applications. This is also the first paper to speed up consensus using expedited delivery; however, the work is for asynchronous systems and it assumes a failure detector that has been tailored to carry some consensus messages. [5] uses a separate network to ensure timely delivery of the messages in their timely computing base. One could use such a separate network for the failure detector messages.

Roadmap. The paper is organized as follows. In Section 2 we give our model with fast failure detectors. We consider the problem of consensus in Section 3: in Section 3.1 we present our basic consensus algorithm, and in Section 3.2 we present our early-deciding one. We show optimality of our consensus algorithms in Section 3.3. We then turn our attention to broadcast problems: first to terminating reliable broadcast in Section 4, then to spontaneous reliable broadcast in Section 5, and finally to atomic broadcast in Section 6. In Section 7 we consider message losses arising from process crashes: we first explain how to modify our algorithms to tolerate such losses in Section 7.1; we then show optimality of the modified consensus algorithm in Section 7.2. Due to space limitations all proofs are omitted, but they can be found in the full version of the paper.

2 Model

We consider a distributed system with n processes that can communicate with each other by sending messages over a fully-connected network. The system is synchronous in that there is a bound D on the time it takes to receive and process a message. Processes have access to a clock, which we assume to be perfectly synchronized for ease of presentation, but our results can be easily extended to clocks that are only nearly synchronized.

Processes can fail by crashing permanently, and there is an upper bound f_{max} on the number of processes that can crash. We let $f \leq f_{max}$ be the actual number of failures. Processes have access to a perfect failure detector that reports failed processes. The detection time, i.e. the time to detect a crash [7], is bounded by some constant d. In useful realizations of our model (e.g., [13]), $d \leq D$ and in fact we make that assumption throughout the paper. In addition, for simplicity of presentation we assume that D is a multiple of d, and if it is not, we can simply increase D to $d\lceil D/d \rceil$, the next multiple of D.[4]

Links do not create or duplicate messages. Moreover, we assume there are no message losses in the first part of the paper, but we later extend our results to allow losses due to the sender's crash.

We now provide a more detailed description.

2.1 Processes

The system consists of a known totally-ordered set $\Pi = \{1, \ldots, n\}$ of processes. The computation proceeds by steps, and a step consists of several stages: (1) the process may first send a message to a subset of the processes, (2) the process then attempts to receive messages addressed to it, (3) the process may then query its failure detector, and (4) the process may change its state according to the messages it received and the information it got from the failure detector. We assume that steps are executed instantaneously. Up to f_{max} processes may fail by crashing; processes that do not crash are called *correct*. When a process crashes during a step, it may stop executing at any of the stages above. In particular, it may crash while attempting to send a message to some subset of processes. If that happens, the send may be partially successful in that it succeeds to send to only some of the targeted processes. This behavior is explained in detail in the next section.

2.2 Links

There is a link between every pair of processes, and messages sent through links are unique (uniqueness can be obtained through sequence numbers). Links guarantee the following properties:

– *(No Creation or Duplication)* A message m can be received at most once and only if it has been previously sent.
– *(D-Timeliness)* A message m sent at time t is not received after time $t + D$.

In the first part of the paper we assume that messages are not lost, that is:

– *(No Loss)* If p sends a message m to q at time t and p does not crash at time t then q eventually receives m from p.[5]

We later extend our results to a more general model that allows message losses if the sender crashes within a certain time after sending the message.

Note that D-Timeliness together with No Loss imply that if p sends m to q at time t and does not crash at time t then q receives m by time $t + D$. Moreover, if p crashes at time t then the messages that it sent may or may not be received. In particular, if p sent to both q and q' then it is possible that only one of them receives the message. To simplify presentation we assume that D is a multiple of d, but we remove this assumption in the full version of the paper.

[4] Doing so slightly increases the time complexities of our algorithms, but the complexity can be reduced as we explain in the full version of the paper.

[5] By convention, we assume that processes receive messages even if they are crashed. Of course, crashed process cannot do anything with these messages.

2.3 Failure Detectors

Processes may have access to a perfect failure detector that provides a list of processes deemed to have crashed [6]. If a process q belongs to the list of process p we say that p suspects q. The failure detector guarantees the following properties:

– *(Accuracy)* A process suspects a process q only if q has previously crashed.
– *(d-Timely Completeness)* If a process q crashes at time t then, by time $t + d$, every alive process permanently suspects q.

Note that if a process p crashes between times t and $t + d$ then some, but not necessarily all, processes may suspect p at time $t + d$.

3 Consensus

In the consensus problem, each process initially proposes a value, and processes must reach a unanimous decision on one of the proposed values. The following properties must be satisfied:

– *(Uniform Validity)* If a process decides v then some process previously proposed v.
– *(Agreement)* Correct processes do not decide different values.
– *(Termination)* Every correct process eventually decides.

Note that consensus allows processes that later crash to decide differently from correct processes. A stronger variant, called *uniform* consensus [20], disallows that by requiring a stronger property than Agreement:

– *(Uniform Agreement)* Processes do not decide different values.

To strengthen our results, we use consensus for our lower bounds and provide algorithms that solve uniform consensus.

A consensus algorithm is said to *reach decision* or *decide* when all alive processes have decided. It turns out that no consensus algorithm can always reach decision within time less than $(1 + f_{max})D$ in a synchronous system without failure detection. However, with a fast failure detector it is possible to do better, as shown in the next section.

3.1 Uniform Consensus Using Fast Failure Detection

Figure 1 shows a simple uniform consensus algorithm that terminates within time $f_{max}d + D$. Each process i keeps a variable e_i with its current estimate of the consensual decision value. Its initial value is the value that the process wishes to propose (line 2). In this algorithm, this variable never changes. We divide real time in consecutive rounds of duration d each, so that each round i corresponds to the time interval $[(i-1)d, id)$. Note that these "mini" rounds are *not* the same as the ones in a synchronous round-based system: here, if $D > d$ then messages sent in a round could be received in a higher round.

At the beginning of round i, process i checks if it suspects all processes with a smaller process id and, if so, it broadcasts (e_i, i) to all (line 4).[6] Then, at time $f_{max}d + D$, all processes decide on the estimate received from the largest process id (lines 5–7). It turns out that lines 1–4 need only be executed by processes $1, 2, \ldots, f_{max} + 1$; the other processes may simply wait and decide at time $f_{max}d + D$ in lines 5–7.

[6] For $i = 1$ note that process 1 vacuously suspects all processes with smaller id, because there are none. Thus, process 1 will send $(e_1, 1)$ to all if it does not crash.

```
Code for each process $i$:
1   Initialization:
2       $e_i \leftarrow$ value that process $i$ wishes to propose

3   at time $(i-1)d$ do
4       if suspect $j$ for all $1 \leq j \leq i-1$ then send $(e_i, i)$ to all

5   at time $f_{max}d + D$ do
6       $max \leftarrow$ largest $j$ such that received $(e_j, j)$ from $j$
7       decide $e_{max}$
```

Fig. 1. Optimal uniform consensus algorithm with time complexity $f_{max}d + D$.

Theorem 1. *In a synchronous system with fast failure detection, uniform consensus can be solved with an algorithm that decides in at most time $D + f_{max}d$ using $(f_{max} + 1)n$ messages.*

3.2 Early-Deciding Uniform Consensus

The consensus algorithm in Figure 1 always decides at time $D + f_{max}d$, where f_{max} is the maximum number of process crashes. In practice, most of the time the number f of failures that actually occur is much smaller than f_{max}, and we would like algorithms to decide faster in these common cases. An algorithm is said to be *early-deciding* if its decision time is proportional to f, not f_{max}. We now present such an algorithm that decides within time $D + fd$. Since $f \leq f_{max}$, this algorithm always performs better than our previous one and, with few failures (small f), it performs much better. This early-deciding algorithm, which is shown in Figure 2, assumes that D is an integral multiple of d (we handle the general case in the full version of the paper).

Like in our previous algorithm, at each round i, process i sends its current estimate if it suspects all processes with a smaller id. However, now processes may change their estimate e_i when they receive the estimate of another process. More precisely, processes keep a variable max_i with the id of the largest process from which it has received an estimate (initially it is zero). When a process receives an estimate from a process whose id is larger than max_i it changes its estimate and updates max_i (line 4). At times $(j-1)d + D$ for $j = 1, \ldots, n$, processes check if they trust process j; if so, they decide on their current estimate (line 9).[7]

Theorem 2. *In a synchronous system with fast failure detection, uniform consensus can be solved with an algorithm that decides in at most time $D + fd$ using $(f+1)n$ messages.*

Note that in the failure-free case, delivery occurs within time D using only n messages.

An important remark is in order. A consensus box assumes that all processes have a priori knowledge that they wish to reach consensus, and it is not the role of consensus to convey that knowledge. Thus, to execute consensus, correct processes are expected to propose a value initially (or at some known common fixed time) and all properties

[7] At times $(j-1)d$ or $(i-1)d + D$, if the process receives a message, we assume it executes line 4 before lines 6 or 8.

```
Code for each process $i$:
  Initialization:
    $e_i \leftarrow$ value that process $i$ wishes to propose
    $max_i \leftarrow 0$

  **upon** receive $(e_j, j)$ with $j > max_i$ **do**
      $max_i \leftarrow j; e_i := e_j$

  **at** time $(i-1)d$ **do**
    **if** suspect $j$ for all $1 \leq j \leq i-1$ **then send** $(e_i, i)$ **to** all

  **at** time $(j-1)d + D$ for every $1 \leq j \leq n$ **do**
    **if** trust $j$ and not yet decided **then decide** $e_i$
```

Fig. 2. Optimal early-deciding uniform consensus algorithm with time complexity $D + fd$.

of consensus are contingent on that. It turns out, however, that even if some correct processes do not propose, the consensus algorithms of Figures 1 and 2 always guarantee that if some process decides a value, this value is one of the proposed values (Uniform Validity). This feature of our algorithms will be used in Section 5. However, it is possible for processes to decide differently when some of the correct processes do not propose. We do not believe this will not be a problem for most applications, but when it is, one could use the atomic broadcast algorithm of Section 6 to solve consensus in the obvious way: to propose a value, a process atomically broadcasts it and then processes decide on the first atomically delivered value. By doing so, we get a consensus algorithm that always satisfies Uniform Validity and Uniform Agreement, even if some correct processes do not propose.

3.3 Time Optimality

Our consensus algorithms are time optimal:

Theorem 3. *In a synchronous system with fast failure detection, every consensus algorithm has a run in which decision takes time at least $D + f_{max}d$.*

This theorem is a special case of a more general one stated in Section 7.2.

Observation 4 *No early-deciding algorithm can always ensure decision in less time than $D + fd$.*

This observation follows immediately from Theorem 3 (because if only f processes fail then we can take $f_{max} = f$). Thus, our early-deciding algorithm is also optimal.

4 Terminating Reliable Broadcast

In the terminating reliable broadcast problem, a designated process called the sender s wishes to broadcast a message. Processes have a priori knowledge of who the sender is and when it intends to broadcast (but they do not know what its message is). The sender, however, may crash and fail before or during the broadcast. The goal is for all alive processes to either deliver the sender's message or to unanimously agree that the sender has crashed by delivering a special "sender faulty" message. More precisely, terminating reliable broadcast guarantees [12]:[8]

- *Validity:* If s is correct then it eventually delivers the message that it broadcasts.
- *Uniform Agreement:* If any process delivers a message m, then all correct processes eventually deliver m.
- *Uniform Integrity:* A process delivers a message m at most once[9] and if $m \neq$ "sender faulty" then m is the message of the sender.
- *Termination:* Every correct process eventually delivers a message.

To solve terminating reliable broadcast, note that the consensus algorithm in Figure 2 always decides on the value of the first process if this process is correct. Thus, we let the sender be that first process. So in line 2 the sender proposes its message, while other processes propose "sender faulty". By doing so, we get an early-delivering algorithm for terminating reliable broadcast that delivers within time $D + fd$. Its message complexity is $(f + 1)n$.

Theorem 5. *In a synchronous system with fast failure detection, terminating reliable broadcast can be solved with an algorithm that delivers in at most time $D + fd$ using $(f + 1)n$ messages.*

Note that in the failure-free case ($f = 0$), delivery occurs at time D using only n messages.

5 Reliable Broadcast

In terminating reliable broadcast (Section 4), all processes have a priori knowledge that there is a sender that wishes to broadcast a message at some known time. This is not the case with reliable broadcast: any process can broadcast at any time, and that time is unknown to other processes. Reliable broadcast guarantees the following properties [12]:

- *(Validity)* If some correct process p broadcasts a message m then p eventually delivers m.
- *(Uniform Agreement)* If any process q delivers a message m then eventually all correct processes deliver m.
- *(Uniform Integrity)* A process delivers a message m at most once and only if it has been previously broadcast.

We focus on *timely* reliable broadcast, which also satisfies the following:

- *(Δ-Timeliness)* If some process p broadcasts a message m at time t then no process can deliver m after time $t + \Delta$.

[8] This is actually *uniform* terminating reliable broadcast, and in fact all broadcasts we consider in this paper are uniform. For brevity we omit the word "uniform" from our broadcasts.

[9] Throughout this paper we assume that broadcast messages are unique (e.g., they contain a sequence number).

```
Code for each process $p$:
  To broadcast a message $m$ at time $t$:
    **send** $(m, p, t)$ **to** all

  **upon** receive $(m, q, t)$ **do**
    **schedule at** time $t + D$:
      **if** trusted $q$ at time $t + d$ **then** $v \leftarrow m$ **else** $v \leftarrow \perp$
      **propose**$_{q,t}(v)$

  **upon decide**$_{q,t}(m)$ **do if** $m \neq \perp$ **then deliver** $m$
```

Fig. 3. Reliable broadcast algorithm.

Here Δ is a known value that specifies how fast processes must deliver messages: together with Uniform Agreement and Validity, Δ-Timeliness implies that if a correct process p broadcasts m then all correct processes deliver m within time Δ.

In Figure 3, we give a timely reliable broadcast algorithm that uses our early-deciding consensus algorithm in Figure 2. To broadcast m at time t, a process p sends (m, p, t) to all. If a process receives (m, q, t), it schedules the following action at time $t + D$: if it trusts q then it runs our consensus algorithm with m as the initial value; else it runs it with $\perp$ as the proposed value. If and when process p decides some value, p delivers that value if it is not $\perp$ else the process does nothing. If there are multiple concurrent broadcasts, the algorithm may start multiple instances of consensus — one for each broadcast. In order to differentiate between these instances, we have subscripted **propose** and **decide** with a unique identifier containing the identity q of the broadcaster and the time t of broadcast.

Notice that if the broadcaster fails in line 2 and only sends to a subset of the correct processes then some correct processes will not propose and start consensus. In that case, however, all processes that propose will propose $\perp$, because the broadcaster will be suspected at time $t + d$. This is the place in which we need the Uniform Validity property even if not all correct processes propose: it guarantees that the processes that decide will necessarily decide $\perp$ (the only proposed value), and thus they will all act in harmony by not delivering the message.

Theorem 6. *In a synchronous system with fast failure detection, timely reliable broadcast can be solved with an algorithm that delivers*[10] *in at most time* $\Delta = 2D + fd$ *using* $(f + 2)n$ *messages.*

Some simple optimizations can improve the delivery time of our algorithm. The first optimization requires that we order processes differently, so that the first round of the consensus algorithm is not executed by process 1, but by the process that wishes to broadcast — let us call this process q. When processes receive (m, q, t) and propose to the consensus box, they indicate that q should be the first process in the order (recall that there is a total order on the process id's); note that processes make a consistent choice on the first id before they start running consensus. Now the next processes in the

[10] If any delivery actually occurs. Note that if the sender crashes, it is possible that no process ever delivers any message.

order can be fixed or, for load balancing, it could be some hash function of q and the time t of broadcast[11]. Then, process q can start the consensus box by proposing m at time $t + D - d$ instead of $t + D$. It can do so because in our consensus algorithm, only the first process acts in the first d time units of execution. The other processes can join in at time $t + D$, after receiving the (m, q, t) message that q sent at time t. With this optimization we save d time units, i.e., processes deliver within time $2D + (f - 1)d$ and we save n messages, i.e., processes use $(f + 1)n$ messages. Note that if $D = d$ then in the failure-free case processes deliver within time D (it turns out that this is true even if there are failures but the *broadcaster* does not fail).

When $D \geq 2d$, a more important optimization allows processes to deliver within time D if the broadcaster does not fail. More precisely, suppose that p delivers the initial (m, q, t) message of q at some time u. Let us assume $u \geq t + 2d$ (if not, p waits until time $t + 2d$). If p trusted q at time $t + 2d$ then p can deliver m right away at time u, since p knows that (1) q's broadcast did not fail and will reach all processes by time $t + D$ and (2) all correct processes will trust q at time $t + d$ and will propose m to consensus. Note however that p still needs to run the consensus box, because other processes may have suspected q at time $t + 2d$ and they will need the consensus box to deliver m. Now if p still trusted q at time $t + 3d$ then p does not even have to bother starting consensus, since all alive processes must have trusted q at time $t + 2d$ and thus they delivered m at that time. With this optimization, if the broadcaster is correct then processes deliver within time D using only n messages[12]:

Theorem 7. *In a synchronous system with fast failure detection, timely reliable broadcast can be solved with an algorithm that delivers in at most time $\Delta = 2D + (f - 1)d$ using $(f + 1)n$ messages. If the broadcaster is correct then delivery takes at most time D using only n messages.*

6 Atomic Broadcast

In atomic broadcast, correct processes must deliver the same set of messages in exactly the same order. More precisely, (uniform) atomic broadcast is a (uniform) reliable broadcast that satisfies the following additional property:

– *(Uniform Total Order)* If any process delivers message m before m' then no process can deliver m' unless it has previously delivered m.

A traditional way to implement atomic broadcast is to order messages by increasing time of broadcast. To do so, the broadcaster timestamps its message, and a recipient can deliver a message as soon as it knows that there are no outstanding messages with a smaller timestamp. More precisely, if p wishes to atomically broadcast m at time t then p uses timely reliable broadcast to broadcast (m, t). If Δ is the maximum delay of the timely reliable broadcast box, a process that gets (m, t) from the box can atomically

[11] It is possible to modify the algorithm so that the broadcaster q selects the order of id's and includes it in its message, instead of just sending (m, q, t).

[12] It is worth noting that processes can deliver even earlier if they receive the message quickly, i.e., the actual message delay happens to be less than D.

```
Code for each process $p$:
  To atomic-broadcast $m$ at time $t$:
    reliable-broadcast$(m, t)$

  **upon** reliable-deliver$(m, t)$ **do**
    for each $g \leq f_{max} - 1$, **schedule at** time $t + \Delta(g) + d$:
      **if** suspect at most $g$ processes
      **then** atomic-deliver in order all atomic-undelivered
          messages with timestamp up to $t$
    **schedule at** time $t + \Delta(f_{max})$:
      atomic-deliver in order all atomic-undelivered
          messages with timestamp up to $t$
```

Fig. 4. Atomic broadcast algorithm with time complexity $\Delta(g) + d$.

deliver m at time $t + \Delta$, because it knows that the box will not later output any messages with a smaller timestamp than t. This idea can be made to work even if $\Delta = \Delta(f)$ is a function of the number f of failures — as long as processes can know f. It turns out there is a simple way to obtain a conservative bound on f: if at some time $u + d$ a process suspects exactly f_0 processes, then our failure detector guarantees that at time u we necessarily have $f \leq f_0$ (this is because our failure detector guarantees that a crashed process is suspected within d time of the crash). Using this idea, we get the algorithm shown in Figure 4.[13]

This algorithm works as follows: when a process p delivers (m, t) from the timely reliable broadcast box, it schedules for execution some subtasks that will atomic-deliver messages. For each $g \leq f_{max} - 1$, process p schedules at time $t + \Delta(g) + d$ a subtask that checks if p suspects at most g processes and, in that case, delivers in order all messages with timestamp up to t (including m). Process p also schedules at time $t + \Delta(f_{max})$ a subtask that will definitely atomically deliver m if it has not done so yet; here there is no need to test the number of crashed processes, because f_{max} is the maximum allowed.

It turns out that there is a way to shave off an extra d from the running time, as follows. Our early-deciding consensus algorithm of Figure 2 actually decides a little faster than within time $D + fd$: it decides by time $D + gd$ if there are only g suspicions at time $D + gd$.[14] Consequently, our timely reliable broadcast also delivers a little faster: a message broadcast at time t is delivered at time $t + 2D + (g - 1)d$ if there are only g suspicions at that time. That means that our atomic broadcast algorithm can deliver d time earlier: if m is broadcast at time t and at time $t + 2D + (g - 1)d$ there are only g suspicions then at this time it can atomically deliver all messages with timestamp up to t, because there are no messages with lower timestamp in transit. With this optimization, we get the following:

[13] Strictly speaking, this algorithm requires that $\Delta(f)$ be a monotonically nondecreasing function of f, which is always the case for our algorithms.

[14] To see why this can be faster, note that with our failure detector a process is only suspected if it has crashed, so at any time the number of suspicions is at most the number of crashes.

Theorem 8. *In a synchronous system with fast failure detection, atomic broadcast can be solved with an algorithm that delivers in at most time $2D + (f - 1)d$ using $(f + 1)n$ messages. In the failure-free case, it delivers in at most time D using only n messages.*

7 Losses in the Outgoing Buffers

In many systems, if a process crashes soon after sending a message, that message may be lost because the low-level outgoing message buffers of that process get wiped out. We now extend our model to allow this type of message losses. We assume that if the sender of a message crashes within δ time of the sending, this message may be lost. Here δ is a system-dependent constant (clearly $\delta < D$). More precisely, we weaken the No Loss property so that it guarantees no loss only if the sender remains alive for δ time:

– *(δ-Hold No Loss)* If p sends a message m to q at time t and p does not crash by time $t + \delta$ then q eventually receives m from p.

Note that D-Timeliness together with δ-Hold No Loss imply that if p sends m to q at time t and does not crash by time $t + \delta$ then q receives m by time $t + D$. Moreover, if p crashes between times t and $t + \delta$ then the messages that it sent may or may not be received. In particular, if p sent to both q and q' at time t, it is possible that only one of the messages is received. Note that if we set $\delta = 0$ then we get our previous model with no message losses.

Throughout Section 7, we assume that D is a multiple of $d + \delta$ (this can always be ensured by increasing D), but we handle the general case in detail in the full version of the paper.

7.1 Algorithms

In the full paper, we show how to modify our algorithms to work in the lossy model above. We have the following results:

Theorem 9. *Consider a synchronous system with fast failure detection, and suppose that messages sent δ time before the sender crashes may be lost.*

– *Uniform consensus can be solved with an algorithm that decides in at most time $D + f(d + \delta)$ using $(f + 1)n$ messages.*
– *Terminating reliable broadcast can be solved with an algorithm that delivers in at most time $D + f(d + \delta)$ using $(f + 1)n$ messages.*
– *Timely reliable broadcast can be solved with an algorithm that delivers in at most time $\Delta = 2D + (f - 1)(d + \delta)$ using $(f + 1)n$ messages. If the broadcaster is correct then delivery takes at most time* $\max\{D, 2d + \delta\}$ *using only n messages.*
– *Atomic broadcast algorithm can be solved with an algorithm that delivers in at most time $2D+(f-1)(d+\delta)$ using $(f+1)n$ messages. In the failure-free case, it delivers in at most time* $\max\{D, 2d + \delta\}$ *using only n messages.*

7.2 Lower Bound

In the full version of the paper we show that, if $D \geq d + \delta$, our consensus algorithm is time optimal, by proving that no algorithm can guarantee decision in less time than $D + f_{max}(d + \delta)$. This lower bound result automatically carries over to terminating reliable broadcast, timely reliable broadcast and atomic broadcast, since all these problems can be easily used to solve consensus without any extra time.

Theorem 10. *Consider a synchronous system with fast failure detection. Suppose that messages sent δ time before the sender crashes may be lost and $D \geq d + \delta$. Every consensus algorithm has a run in which decision takes time at least $D + f_{max}(d + \delta)$.*

Observation 11 *No early-deciding consensus algorithm can always ensure decision in less time than $D + f(d + \delta)$.*

Note that Theorem 10 is a generalization of Theorem 3 of Section 3.3: we obtain the latter by taking $\delta = 0$.

8 Discussion

A consensus lower bound in the continuous-time synchronous model. There is a well-known lower bound of $1 + f_{max}$ rounds to solve consensus in synchronous systems without failure detectors [18]. This result is for the round-based model, in which processes execute in lock-step rounds, and it does not immediately translate to the continuous-time synchronous model, in which there is no notion of round and processes take steps at any time they wish. In the latter model, one would expect a time lower bound of $(1 + f_{max})D$ to solve consensus. Indeed, such a bound is correct and it follows from Theorem 10: we simply take $d = D$ and $\delta = 0$. This seems to be the first proof of this fact.

Issues on achieving fast failure detection. Some fault-tolerant and real-time systems have fast built-in circuitry to detect failures, but often failure detection is not available in hardware and so it needs to be implemented in software by message passing. To achieve small detection time, one needs to use expedited delivery for the failure detector messages and send them frequently. In practice the bandwidth available for expedited messages is limited and so we should reduce the number of failure detection messages as much as possible. This can be achieved as follows. First, note that the straightforward way implement a perfect failure detector is for each process to periodically send I-am-alive messages to each other — a total of $O(n^2)$ periodic messages. It turns out, however, there are better failure detector implementations that only send linearly many messages while impacting the detection time by only a small factor (using similar techniques as in [3]). Second, note that our consensus algorithms do not require a failure detector among all n processes, but only among $f_{max} + 1$ of them: in many applications, f_{max} is significantly smaller than n. Thus, only $f_{max} + 1$ processes need to run the failure detector to solve consensus.[15] Third, if the failure detector is implemented as a shared service [8,15,13], the failure detector traffic does not significantly increase as the number

[15] This particular optimization does not apply to our broadcast algorithms because any process is allowed to broadcast and the broadcaster needs to be monitored by the failure detector.

of concurrent instances of consensus and broadcast increase: such an implementation sends machine-to-machine messages that combine the I-am-alive messages of different applications that share the same machine. Finally, note that [13] has shown that it is possible to implement a failure detector in a local area network with very fast detection time while maintaining a reasonably low total bandwidth and processor consumption: In an Ethernet network with (a) a total consumption chosen to be limited by 5%, (b) $f_{max} = 5$ and (c) n ranging from 16 to 1024, the optimal values of the maximum detection time d ranged from 51.9ms to 68.5ms (as n varied), while the maximum message delay D of regular messages ranged from 106.6ms to 6 828.7ms *plus* the maximum sojourn time of a regular message in the interprocess waiting queues of sender and receiver.

References

1. M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.
2. M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, Apr. 2000.
3. M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Symposium on Distributed Computing*, Lecture Notes on Computer Science, Oct. 2001.
4. Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. Technical Report UBLCS-95-18, Dept. of Computer Science, University of Bologna, Bologna, Italy, November 1995.
5. A. Casimiro, P. Martins, and P. Veríssimo. How to build a timely computing base using real-time linux. In *Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems*, pages 127–134, Sept. 2000.
6. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996. A preliminary version appeared in *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, Aug., 1991, 325–340.
7. W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, Jan. 2002.
8. B. Deianov and S. Toueg. Failure detector service for dependable computing (fast abstract). In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages B14–B15. IEEE Computer Society, June 2000.
9. D. Dolev and R. Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 32(1):191–204, Jan. 1985.
10. D. Ferrari and D. C. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, Apr. 1990.
11. R. Guerraoui, M. Larrea, and A. Schiper. Non blocking atomic commitment with an unreliable failure detector. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 41–50, Sept. 1995.
12. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Department of Computer Science, Cornell University, Ithaca, New York, May 1994.

13. J.-F. Hermant and G. Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, Aug. 2002. Special issue on Asynchronous Real-Time Distributed Systems.
14. M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, 1999.
15. D. Ivan, M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, November 2001. Prototype of a shared failure detector service with QoS guarantees.
16. J. F. Kurose, M. Schwartz, and Y. Yemini. Multiple-access protocols and time-constrained communication. *ACM Computing Surveys*, 16(1):43–70, Mar. 1984.
17. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.
18. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.
19. G. Le Lann, 2001. Private communication with Astrium, Axlog, European Space Agency.
20. G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.
21. K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(1):147–171, Sept. 1995.
22. H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–1399, Oct. 1995.

Author Index