Lecture Notes in Computer Science 2218
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

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

Rachid Guerraoui (Ed.)

Middleware 2001

IFIP/ACM International Conference
on Distributed Systems Platforms
Heidelberg, Germany, November 12-16, 2001
Proceedings

Springer

Series Editors

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

Volume Editor

Rachid Guerraoui
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
E-mail: rachid.guerraoui@epfl.ch

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Middleware 2001 : proceedings / IFIP-ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany, November 12 - 16, 2001. Rachid Guerraoui (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2218)
ISBN 3-540-42800-3

CR Subject Classification (1998): C.2.4, D.4, C.2, D.1.3, D.3.2, D.2

ISSN 0302-9743
ISBN 3-540-42800-3 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10840892 06/3142 5 4 3 2 1 0

Preface

In theory, the term middleware denotes any software that can be used for more than one application. In practice, the term middleware tends to denote software abstractions for distributed computing, including communication abstractions such as RPC, and reliability abstractions such as transactions. Devising and implementing such adequate abstractions has constituted an active area of research in the last decade, bridging the gap between various fields, including programming languages, distributed systems, networking, and databases.

Middleware 2001 built upon the success of Middleware 2000 and Middleware 1998, as ideal forums for researchers and industrialists in these fields to present and discuss the latest middleware results.

Among the 116 initial submissions to Middleware 2001, 20 papers were selected for inclusion in the technical program of the conference. Every paper was reviewed by at least three members of the program committee. The selection of the papers was performed during a program committee meeting, held in Lausanne on 6 July 2001. The papers were judged according to their originality, presentation quality, and relevance to the conference topics. The accepted papers cover various subjects such as mobility, scalability, quality of service, and reliability.

I would like to express my deepest appreciation to the authors of the submitted papers, to the program committee members for their diligence in reviewing the submissions, to Richard van de Stadt for perfectly running CyberChair, and finally to the members of the steering committee and the other organizing committee members for their efforts towards making Middleware 2001 a successful conference.

August 2001 Rachid Guerraoui

Organization

Middleware 2001 was organized under the auspices of IFIP TC6 WG6.1 (International Federation for Information Processing, Technical Committee 6 [Communications Systems]. Working Group 6.1 [Architecture and Protocals for Computer Networks]).

Steering Committee

Gordon Blair	Lancaster University
Jan de Meer	Condat AG
Peter Honeyman	CITI, University of Michigan
Guy LeDuc	University of Liege
Kerry Raymond	DSTC
Alexander Schill	TU Dresden
Jacob Slonim	Dalhousie University

Sponsoring Institutions

IFIP (International Federation for Information Processing)
`http://www.ifip.or.at`

ACM (Association for Computing Machinery)
`http://www.acm.org`

Cooperating Institutions

USENIX
`http://www.usenix.org`

ACM SIGCOMM
`http://www.acm.org/sigcomm/`

SIGOPS

ACM SIGOPS
`http://www.acm.org/sigops/`

Supporting Companies

Agilent Technologies
http://www.agilent.com

BBN Technologies
http://www.bbn.com

IBM
http://www.research.ibm.com

Organizing Committee

General Chair:	Joe Sventek, Agilent Laboratories
Program Chair:	Rachid Guerraoui, EPFL
WIP and Poster Chair:	Maarten van Steen, Vrije Universiteit
Tutorials Chair:	Frank Buschmann, Siemens AG
Publicity Chair:	Joe Sventek, Agilent Laboratories
Advanced Workshop Chair:	Guruduth Banavar, IBM
Local Arrangements Chair:	Alex Buchmann, Universität Darmstadt

Program Committee

Gustavo Alonso	ETH Zürich
Jean Bacon	University of Cambridge
Gregor v. Bochmann	University of Ottawa
Geoff Coulson	Lancaster University
Naranker Dulay	Imperial College
Jean-Charle Fabre	Laas-CNRS
Pascal Felber	Bell Labs
Svend Frølund	HP Labs
Peter Honeyman	CITI, University of Michigan
Yennun Huang	AT&T Labs
Anne-Marie Kermarrec	Microsoft Research
Doug Lea	SUNY at Oswego
Christoph Liebig	Universität Darmstadt
Claudia Linnhoff-Popien	Universität München
Silvano Maffeis	Softwired Inc.
Louise Moser	University of California at Santa Barbara
Fabio Panzieri	Università di Bologna
Luis Rodrigues	Universidade de Lisboa
Isabelle Rouvellou	IBM
Santosh Shrivastava	Newcastle University
Jean-Bernard Stefani	INRIA, Grenoble
Zahir Tari	RMIT University
Steve Vinoski	IONA Technologies
Werner Vogels	Cornell University

Referees

Alok Aggarwal
Alessandro Amoroso
Pedro Antunes
Filipe Araújo
Gordon Blair
Alejandro Buchmann
Antonio Casimiro
Dan Chalmers
Mariano Cilia
Nick Cook
Thierry Coupaye
Silvano Dal Zilio
Lou Degenaro
Matt Denny
Bruno Dillenseger
Deepak Dutt
Khalil El-Khatib
Christian Ensel
Paul D. Ezhilchelvan
António Ferreira
Ludger Fiege
Markus Garschhammer
Ioannis Georgiadis
Vittorio Ghini
Indranil Gupta
Nabila Hadibi
Daniel Hagimont
Peer Hasselmeyer
Rainer Hauck
Naomaru Itoi
Axel Küpper
Roger Kehr
Terence P. Kelly
Bernhard Kempter
Brigitte Kerhervé
Arasnath Kimis
Olga Kornievskaia
Samuel Kounev
Alexandre Lefebvre
Claudia Linnhoff-Popien
Mark C. Little
Emil Lupu
Gero Mühl
Vania Marangozova
Eric Marsden
Stan Matwin
Patrick McDaniel
P.M. Melliar-Smith
Thomas A. Mikalsen
Hugo Miranda
Graham Morgan
Lukasz Opyrchal
José Orlando Pereira
Peter Pietzuch
Nuno Preguica
Niels Provos
Igor Radisic
Pedro Rafael
Kerry Raymond
Helmut Reiser
Marco Roccetti
Manuel Rodriguez
Harald Roelle
Alex Romanovsky
David Rosenblum
Juan-Carlos Ruiz
Mohamed-Vall M. Salem
Michael Samulowitz
Holger Schmidt
Marc Shapiro
Mário J. Silva
Jacob Slonim
Jesper H. Spring
Jan Steffan
Stanley M. Sutton Jr.
François Taïani
Stefan Tai
Gerald Vogt
Shalini Yajnik
Walt Yao
Haiwei Ye
Andreas Zeidler
Stuart M. Wheater

Table of Contents

Java

Mobility

Distributed Abstractions

Reliability

Home & Office

Scalability

Quality of Service

Automated Analysis of Java Message Service Providers

Dean Kuo and Doug Palmer

Software Architectures and Component Technologies Group
CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra, Australia
{dean.kuo, doug.palmer}@cmis.csiro.au

Abstract. The Java Message Service (JMS) is a specification that provides a consistent Java API for accessing message-oriented middleware services. This paper presents a test harness that automates the testing of JMS implementations (providers) for correctness and performance. Since the JMS specification is expressed in informal language, a formal model for JMS behaviour is developed, based on the I/O automata used in other group communication systems. The test harness has been successfully used to test a number of JMS implementations. This paper contains a descriptive presentation of the formal model, the full details are found in a technical report.

Keywords: Automated component testing, performance analysis, formal model, JMS, I/O Automata.

1 Introduction

Message-Oriented Middleware [13] (MOM) is a popular building block for the construction of distributed systems especially loosely coupled systems. When using MOM, decoupled software components communicate by asynchronously exchanging messages. The message management layer can be separated from the application code and MOM systems typically provide a platform for reliably delivering messages between these independent software components. Messaging systems can be contrasted with synchronous remote procedure call systems, such as CORBA; the asynchronous nature of MOM makes it a suitable candidate for designs where a high level of autonomy is a desirable design goal. See [2] for a survey of middleware models.

In general, the semantics of MOM are quite simple: the construction of a message, passing it to the middleware with a destination and receiving a message from the middleware. One source of additional complexity arises from services supplying guaranteed levels of reliability and management. For example, it may be necessary to have messages survive across failures or be delivered in the same order relative to messages from other senders. Another source of complexity is whether the messaging system uses point to point, broadcast or multicast[1] delivery. The problems of reliable message delivery have been extensively studied in *Group Communication Systems* (GCSs). GCS can be categorised according to the properties that they have in terms of message delivery, reliability and membership management [16,3].

[1] Normally called publish/subscribe or pub/sub in MOM circles.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 1–14, 2001.

MOM can be provided as a simple API, since it can be modelled as a single service to which requests for delivery or notification are made. The *Java Message Service* (JMS) is a Java interface specification to MOMs [4,12]. Individual JMS providers supply a underlying platform that supports the semantics of JMS. Examples of JMS providers are MQSeries [5] and SoniqMQ [14].

When building a JMS provider, the question of whether the provider accurately follows the JMS specification arises. JMS provides a large number of features: persistent delivery, durable subscription, transactional delivery, to name a few. Testing a JMS provider is complicated by JMS being part of a distributed, asynchronous system, making questions of whether a message has been received, and whether it *should* have been received an essential part of the testing process. Ideally, the JMS specification would contain a formal model that exactly specifies which messages are required to be delivered and to which receivers. Testing a JMS provider would then be a matter of comparing the behaviour of the provider against this formal model. However, the JMS specification is designed only as a standard API to existing MOM and is not a model for messaging. As a result, the JMS specification is, at places, somewhat vague on the expected behaviour of a JMS provider, with delivery latency and subscription latency affecting the delivery of messages.

The properties of GCSs can be used to build a formal model of JMS behaviour. Many of the properties of GCS described by Vitenberg et al [16] can be directly applied to JMS. Some of the formalism of GCSs, however, rely on more knowledge than JMS is able to provide.

GCS group membership is expressed in terms of *views:* sets of processes to which a message should be delivered. The sender of a message knows which view a message has been sent in and, therefore, which receivers should be getting the message. Publish/subscribe, however, is usually an anonymous communication model with a publisher publishing on a *topic* with no knowledge of the potential subscribers [10]. A consequence of this anonymous model is that JMS provides no means of determining view membership.[2] As a result, the view formalism cannot be applied to JMS without a considerable white-box knowledge of the underlying provider implementation.

Similarly, the liveness properties discussed in Vitenberg et al cannot be directly derived from the JMS specification, as failure detection is not part of the JMS specification. In addition, JMS is a purely asynchronous system; no upper bound is placed on message delay[3]. In general, however, performance measures, such as the maximum sustainable throughput of messages and the maximum message delay can be used to impose an upper bound; messages that are delayed beyond this time are assumed to be lost.

This paper describes a methodology for black-box testing of a JMS provider. A test harness exercises the JMS provider and a model of the expected behaviour of the JMS provider is built; this model can then be compared against the actual behaviour of the provider. Black-box testing relies on the externally observable behaviour of a system. Since views are not observable entities in JMS, the model uses initial and final message deliveries to a receiver to mark changes of view. Performance measures are used in place of testing liveness properties. This methodology has been implemented as a distributed

[2] A JMS provider may have a distributed architecture that makes exact calculation of view either impossible or very expensive. For example, a collection of listeners acting as filters for groups of subscribers.

test harness which manages a series of tests and analyses the results. The test harness has been used to aid the component testing of Fujitsu's JMS product that will be later released onto the market. The harness is also being used to compare performance of different JMS implementations. It should be noted that this paper is not about discussing the results of testing a variety of JMS implementations but to describe a tool that automates the testing.

The test harness provides JMS application developers a vendor independent tool to compare the performance of different JMS providers. The developers can configure the tests to closely resemble the target environment. The result is that developers can gather accurate information on which JMS provider best suits their requirements.

The Ballista project [6] describes the automation of robustness testing of off-the-self software components. That is, it tests for component failures caused by exceptional inputs or stressful environment conditions. They do not test for correct behaviour nor performance testing, which is the focus of the JMS test harness described in this paper.

The paper is structured as follows: Section 2 contains a discussion of the properties of GCSs and their application to JMS. An informal analysis model for JMS, based on the properties discussed in section 2 is presented in section 3. The formal analysis model, upon which section 3 is based, can be found in [7]. Design, implementation and experience gained in developing the test harness is given in section 4. Conclusions and future directions is given in section 5.

2 Background

This section provides a brief overview of JMS and GCS.

2.1 JMS

JMS defines a Java API for message oriented middleware (MOM) and a number of implementations already exist. JMS defines two forms of messaging: point to point and publish and subscribe.

A typical JMS client uses JNDI (Java Naming and Directory Interface) [8] to load a *ConnectionFactory* and a set of destination objects into a JMS client. The connection factory is used to create connections with the MOM which is then used to create a set of sessions. Message producers then use sessions and destinations to send messages to message consumers. Message consumers also use sessions and destinations to receive messages.

A session can be specified as transacted. Each transaction groups a set of message sends and receives into a unit of work. If the session commits then all received messages are acknowledged and all outgoing messages are sent. If the session aborts, all messages received are recovered while all outgoing messages are destroyed — that is, the operations in the transaction never took place.

For non transactional receivers, JMS defines three acknowledgement modes. The options include lazy acknowledgement which reduces the work done by a session but duplicate messages may be delivered; sessions automatically acknowledges each message; and the JMS client manually acknowledges the receipt of message(s).

A JMS client that produces messages create them via sessions. The client specifies whether the message is a persistent message or a non-persistent message. Persistent messages are guaranteed to eventually arrive at its destination(s) even if failures (system or communication) occur. For non-persistent messaging, the messages *should be* delivered since failures may cause it to be lost.

Determining the destination of a message is simple for point to point messaging. The message producer explicitly specifies the intended destination. Defining correct behaviour is thus straightforward.

For publish and subscribe messaging, defining *correct behaviour* requires us to know the destinations of a published message. However, the destination of a message is unknown to the publisher since it has no knowledge of who has subscribed to the topic and there is no way, using the JMS API, to determine who are the current subscribers (i.e. no view membership API in JMS). Discussion and the definition of correct behaviour for publish and subscribe is described in section 3.

JMS defines two forms of subscribers: non-durable and durable. A non-durable subscriber will only receive messages while it is active. For durable subscribers, messages that would have been delivered while it was inactive will be delivered to the subscriber when the subscriber becomes active again — that is, the JMS implementation has to retain all the messages while the subscriber was inactive.

JMS defines a 10 level priority $(0-9)$ where 9 is the highest priority and 0 the lowest. The JMS implementation only needs to provide a best effort in delivering higher priority messages before lower priority messages.

Messages from the same message producer are guaranteed to be delivered to consumers in the same order as they were sent if their message priorities and delivery modes are the same. In addition, messages sent in non-persistent mode may skip ahead of messages sent in persistent mode but the reverse is not permitted.

2.2 GCS

The publish and subscribe component of JMS is a lightweight form of (GCS) [16]. GCSs typically provide reliable multicast and group membership services. Each message sent by an application will be delivered, if certain conditions are met, to all the processes in the current membership group. This is similar to JMS delivering messages published on a particular topic to all subscribers that have subscribed to the topic. One major difference however is that JMS does not expose to the JMS application who the subscribers to a topic are while in GCS, each process is notified when a *view* changes.

Applications such as state-machine replication and distributed transactions rely on a precise membership service. We view JMS as a lightweight version of a GCS since JMS does not provide any sort of precise membership service and thus do not provide properties such as *sending* and *same view delivery* [16].

GCS and related papers [16,3,10,9] and papers referenced in [16] typically describe algorithms, analysis algorithms, define correctness and describe implementations of GCS. JMS is a standard and just defines an API, and informally defines correctness.

Both JMS and GCS share some basic safety and liveness properties such as delivery integrity (messages received by consumers where actually sent by some message producer), FIFO delivery and reliable message delivery. JMS' reliable message delivery requires that persistent messages be eventually delivered even if communication

or process failures occur while in GCS, it only requires that messages be delivered in the absence of failures. Thus, JMS' notion of reliable message delivery is stronger than GCS'.

3 Analysis

The main objective of the JMS test harness is to analyse the behaviour — correctness and performance — of any JMS implementation. This section describes how the JMS test harness analyses the results from tests. For each test, an execution trace (logged events from a test) is generated by the JMS test harness. The trace is then analysed to verify correctness and to determine performance.

The first task is to formalise the specification as the JMS specification [4] is currently only informally specified in English. There are two types of correctness properties, *safety properties* and *liveness properties* [9]. Safety properties basically say that what has happened is correct or something bad has not happened. Liveness properties basically say that something *good* will eventually happen if certain conditions are met.

The test harness logs each event that occurs. Thus, it is straightforward to analyse an execution trace to verify that safety properties have not been violated. However, this is not the case for liveness properties since one can not determine if an event will *eventually* happen. The approach taken by the test harness is only test for safety, and instead of testing for liveness, perform performance evaluation.

Sections 3.1 and 3.2 respectively describe the JMS safety properties and the performance analyses. The safety properties discussed in this section are presented informally, giving an intuitive feel for the structure of the properties. A full formal definition is given in [7].

3.1 Safety Properties

This section describes the JMS safety properties. They cover message delivery, transactions, message ordering, durable subscriptions, message priority and message expiration.

Point-to-point messaging essentially delivers messages to a destination; a named queue. Receivers can pick up messages from this destination asynchronously, with the messages waiting until a receiver appears. Publish-subscribe messaging delivers messages to the group of subscribers that have subscribed to the topic the message was published on. Senders to a queue and publishers on a topic are collectively termed message *producers*. Receivers from a queue or subscribers to a topic are message *consumers*.

Message Delivery. This section describes the safety properties for both publish and subscribe and point-to-point messaging. *Message delivery* tests that all messages that are required to be delivered to a consumer are actually delivered to the consumer. Defining exactly which messages fall into this category is not straightforward due to message delivery latency times and latency times to register a subscription.

When a message is sent to a queue or published to a topic, there is a *delivery latency* as the message passes through the message system before arriving at its destination(s). The delivery latency may mean that the message, even though sent while the consumer was

active may arrive after the consumer has closed its connection, leading to the message not being received by the ex-consumer.

When a subscriber subscribes to a topic, there is a *subscription latency* while the subscription propagates throughout the system. Since publishers are not likely to send messages to non-subscribing nodes, it is possible for a published message not to be delivered to a subscriber even though the subscriber has recently subscribed.

Queue receivers and durable subscribers have the property that messages not delivered to a consumer wait until a consumer becomes available. The newly active consumer may not be the same consumer as any of the previous consumers. Given this behaviour, it is thus natural to treat a destination or subscription as an end-point for message delivery and reason about groups of consumers. Note, the group of consumers for a non-durable subscription contains exactly one process as the subscription terminates as soon as the process closes the subscription.

The set of messages that are required to be delivered to a consumer depends on the operational modes of the producer and consumer — e.g. transactional, durable (subscriber) and persistence. Before we can define exactly which messages are required to be delivered, we need a number of preliminary definitions to capture the semantics of transactional producers and consumers, and durable subscribers. The first definition defines exactly which messages have been sent by a producer. In the definition, notice that for transactional producers, a message is not deemed to be sent until the producer commits.

Definition 1. ***Sent Messages***
If the producer p is *non-transactional*, then the message msg is defined as sent by p if the producer successfully sends the message — that is, the method `publish(msg)` or `send(msg)` returns with no exception thrown.

If the producer p is *transactional*, then the message msg is defined as sent by p if the publisher successfully sends the message within a transactional context t and t later commits. □

Similar to the above definition, we define exactly which messages have been received by a consumer.

Definition 2. ***Received Messages***
If the consumer c is *non-transactional*, then a message is defined as received by c if the consumer successfully receives the message.

If the consumer s is *transactional*, then a message is defined as received by s if the consumer successfully receives the message within a transactional context t and t later commits. □

It is now possible to define the basic delivery integrity property, which states that each message received by a consumer was actually produced by some producer. This property is the same as property 4.1 in [16].

Property 1 ***Delivery Integrity***
For each consumer c and each message m in c's *Received Messages*, m is also in the set *Published Messages* for some producer p. □

Due to delivery latency and subscription latency, it is not possible to define ahead of time exactly which messages will be delivered to a consumer. Instead, given the first message from a producer that arrives at a consumer and the last message from a producer that arrives before the consumer closes, it is possible to construct the set of messages that should have been received in the intervening period. To do this, it is necessary to identify the next message from a producer that should follow a received message.

Definition 3. ***Next Message***
Given a message m sent by a producer p and received by a consumer c, the *next message* that should be received by c from p is the message produced immediately after m by p.

Messages are assumed to be delivered to either queues or subscriptions (each with a unique identifier[3]), representing a *consumer group*. Consumer groups are given the identifier of the queue or the subscription that they represent. A consumer becomes the active consumer of a consumer group when it opens a destination and it stops being the active consumer once it closes the destination. Note, message may still arrive at a consumer group even if there are no active consumers. It is therefore necessary to identify the last close and last message delivered. Note that, for non-durable subscriptions, there will be at most one close operation since the consumer group for a non-durable subscription contains exactly one subscriber.

Definition 4. ***Last Close***
The *last close* of a queue or subscription id is the last close operation of consumer (queue receiver or subscription) group on id. □

Definition 5. ***Last Message***
The last message from a producer p to a consumer group id is the last message sent from p to id which is received before the last close of id. □

The last message defines the last message received by a consumer group from a producer. It is also necessary to identify the first message received by the consumer. Subscription latency in publish and subscribe style of messaging means that the definition of *first message* will be different for a queue and a subscription.

Definition 6. ***First Message***
The first message, m, from a producer p to a consumer group id is either:

- If id is a queue, then m is the first message sent by p.
- If id is a subscription, then m is the first message sent by p that was received by a subscriber to id. □

Once the first and last messages have been identified, correctness requires that all messages between these messages are received by some consumer of the queue or subscription.

Property 2 ***Required Messages***
The *required message set* for an end-point (queue or subscription) id and a producer p is defined, recursively, as:

[3] Non-durable subscribers are allocated an artificial subscription for the life of the subscriber.

- If m is the first message from p to id then m is in the required message set.
- If m is in the required message set, and m' is the next message after m from p, then either m is the last message from p to id or m' is in the required message set.

The required message set is the union of all required message sets over all p and id. Correctness requires that the union of all messages received by consumers be a super set of the required message set. □

Message Ordering. Messages sent by a message producer with the same message priority and delivery mode and, on the same topic in the case of pub/sub messaging style, must be delivered in the same order as it was sent. Message ordering is a basic property of any GCS and property 6.1 in [16] describes this property. Verifying that message ordering has been preserved can be reduced to verifying the following property:

Property 3 ***Message Ordering***
Take any message msg received by a message consumer and message msg' is the previous message received by the consumer that is from the same producer, on the same topic (in the case of pub/sub messaging) with the same message priority and delivery mode as the message msg.

Ordering is preserved if msg' was published before msg. □

Figure 1 illustrates that if message ordering is not preserved then there exist messages msg and msg' satisfying the conditions in property 3 where msg' was *not* published before msg.

Message Priority. Since the JMS Specification specifies that supporting message priority only requires a *best effort*, the following property can be relaxed or dropped altogether. There is no corresponding property in [16].

Property 4 ***Message Priority***
The mean message delivery time between a producer and consumer for a lower message

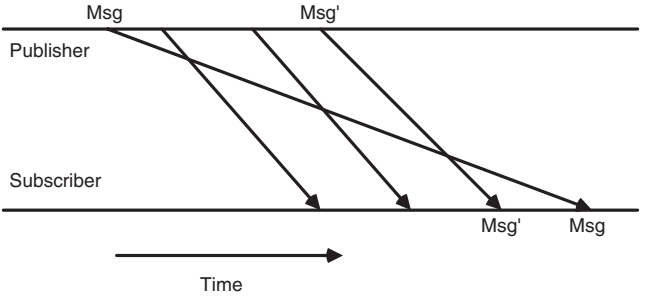


Fig. 1. Message Ordering

priority is greater or equal to the mean message delivery time for a higher message priority.

This correctness property assumes that messages produced for the different priorities are produced at the same rate, on the same topic or to the same queue and with the same message delivery mode. □

Expired Messages. The JMS specification requires that a JMS provider should do its best to accurately expire messages [4]. However, the specification does not define the accuracy required by a JMS provider, beyond stating that time-to-live should not be simply ignored. There are two parts to this property: one relates to the number of expired messages that have been delivered to a consumer; the other relates to the number of non-expired messages that were not delivered to the consumer. The test harness (a JMS client) has no way of determining which messages have been expired while in the JMS provider, and which messages were correctly delivered, but expired by the time the client processed them. Precise tests for message expiry are, therefore, impossible from the perspective of the test harness.

An expectation model allows the expiry property to be tested, without too much knowledge of the internal workings of a JMS provider. To build the expectation model, it is necessary to define which messages *could* have been received by a consumer.

Definition 7. ***Possibly Received Messages***
A message m sent to a queue q is a member of the possibly received messages of q. A message m published on topic t is a member of the possibly received messages of s, if s has subscribed to t. □

The simple expectation model that we have deployed specifies that a possibly received message is expected to be delivered if the mean latency time is less than or equal to the time-to-live time of the message or when the message's time-to-live is 0[4]; Otherwise, the message should not be delivered.

Property 5 ***Expired Messages***
The property is in two parts.

- The number of expired messages that are delivered as a percentage of the number of expected expired messages is less than some pre-specified percentage.
- The number of non-expired messages delivered as a percentage of the number of expected non-expired messages is greater than some pre-defined percentage. □

The configuration we define to test message expiration sets message time-to-live to either 1 (milli-seconds) or 0 (infinity) — thus, all messages with time-to-live of 1 are not expected to be delivered while all messages with time-to-live of 0 are expected to be delivered.[5]

A more realistic expectation model would use a message's time-to-live and the run's delay histogram to determine the probability that a message should be delivered. As the test configuration only use time-to-live of 0 and 1, the simplified expectation model is sufficient for testing purposes.

[4] A time-to-live of 0 means that the message never expires — i.e. infinite time-to-live

[5] Granularity considerations within the JMS provider may result in some inaccuracy in expiry. For example, a 1 milli-second time-to-live may be rounded up to 1 second.

3.2 Performance

A trivial JMS implementation — one that never delivers any messages — will satisfy all the safety properties discussed in Section 3.1. Liveness properties show that a JMS implementation does something useful but it is not possible, from analysing an execution trace, to verify liveness properties. Instead of testing for liveness, the JMS test harness measures the performances of the JMS implementations. Trivial JMS implementations can easily be identified, since their throughput will always be zero messages per second.

Performance analysis enables users of JMS to compare the throughput of different JMS implementations. They can then use these results to determine the implementations that meet their performance requirements.

The configuration of a test specifies the rate at which messages should be sent. From the execution trace, the analysis determines the rate at which messages were actually sent and the rate at which messages are delivered. The configuration is highly flexible and allows the users to specify the message body type (StreamMessage, MapMessage, TextMessage, ObjectMessage and BytesMessage) and size of messages to be sent, the message priority, delivery mode, whether the producers and consumers are transactional, the acknowledgement modes of the consumers (if applicable), amongst others. Tests can also be configured such that the senders send messages in bursts or with a profile corresponding to a piosson distrution. The test harness can be employed to determine the performance of the JMS provider under different configurations without the need to write any code.

Figures 2 and 3 show two different behaviours of two JMS providers under pub/sub as the message send rate is increased over a number of tests — note that these figures do not have equal scales. The first figure also shows that the throughput remains constant when the send rate is greater than then maximum throughput that the JMS implementation can handle. The second figure shows the subscriber throughput drops as the system is over-stressed.

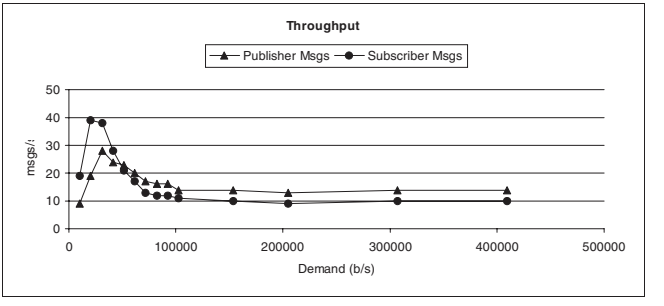


Fig. 2. Example Throughput Results: JMS Provider I

The performance measures taken give some indication of the overall behaviour and usefulness of the system. The performance measures are also used to evaluate some correctness properties, where approximate measures are taken (properties 4 and 5).

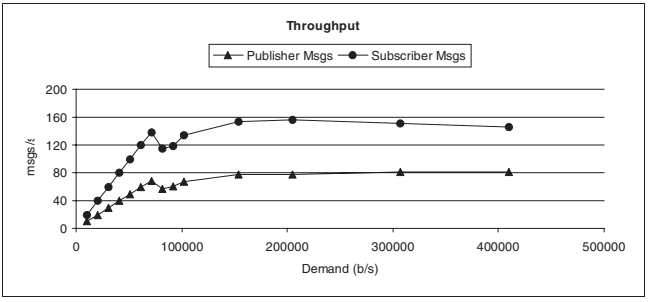


Fig. 3. Example Throughput Results: JMS Provider II

A running test is divided into three periods: *warm-up, run* and *warm-down*. The warm-up period is a pre-defined period at the start of each test. After the warm-up period, the harness assumes that the system is in a steady state. The warm-down period represents a period during which producers stop producing messages, allowing consumers to catch up with any tail of un-consumed messages that has built up. The correctness properties apply to all three periods. Performance measurements are taken only for messages produced during the run period.

The performance measures are calculated by examining the test harness logs and matching message identifiers. The following performance measures are taken:

producer throughput. The rate of message production, both in terms of messages per second and message body bytes per second.

consumer throughput. The rate of message consumption, both in terms of messages per second and message body bytes per second.

message delay. Message delay is calculated as time between the start of the message delivery to a consumer and the start of the call to send or publish the message[6]. The maximum, minimum, mean and standard deviation of message delays are calculated.

fairness. Fairness is a measure of the level to which the provider favours a producer or consumer over other producers and consumers — the standard deviation of the message delay is another measure of fairness. Unfairness is defined as the standard deviation of the per-producer or per-consumer mean delay.

4 Design

The JMS test harness is designed to allow automated testing of a JMS provider across a number of independent machines, connected by a network. The architecture of the harness is shown in figure 4.

[6] An alternative measurement of delay would be the time between the message delivery and the time the message production method returns. Some JMS providers appear to implement message delivery, particularly for non-persistent messages, via a series of synchronous calls. The alternative measurement, in these environments, can produce apparently negative delays.

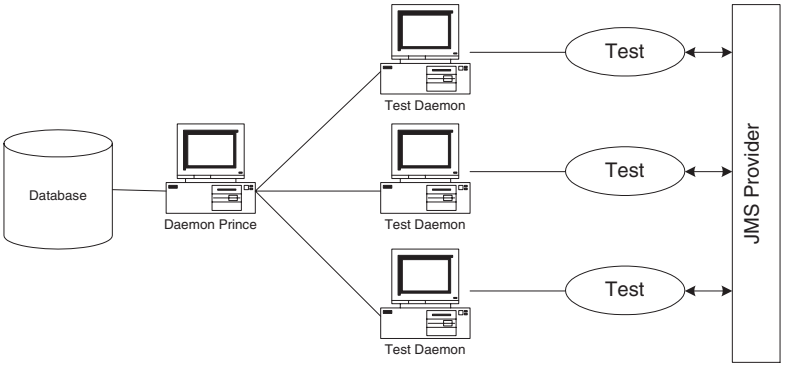


Fig. 4. Test Harness Architecture

A number of tests are run, in separate Java virtual machines and distributed across several systems. A test creates a variety of producers and consumers and starts sending and receiving messages. As each message is sent and received, these events are logged to disk, along with the unique message identifier and a timestamp.[7] Individual producers and consumers can be configured with different message production, persistence, durability and other characteristics, as well as connection and disconnection behaviour. The producers and consumers are grouped into *nodes*, which can be configured to share resources such as JMS connections or sessions.

The running of a test is handled by *test daemons,* usually one to a machine. The test daemons are responsible for launching the tests, starting the tests in a coordinated fashion and monitoring the tests for completion (or failure). The test daemons are coordinated by a *daemon prince,* a program responsible for scheduling tests and ensuring that the test daemons stay coordinated. The daemon prince, test daemons and tests use RMI [15] to coordinate, to avoid relying on the same middleware as the middleware being tested.

When a test completes successfully, the test logs are collected and returned to the daemon prince. The daemon prince then inserts the logs into a SQL database, using JDBC [17]. A set of SQL statements are then used to verify correctness and to determine performance. The SQL statements correspond to the JMS properties described in section 3.

The SQL database used is Microsoft Access. Configuration screens, test generators and results reporting are handled by the Access forms and reports facilities.

4.1 Experience

In general, the test harness has performed well. It has proved quite robust, catching crashed tests, cleaning up and continuing on with the next test. Hooks for initialisation scripts allow the JMS provider to be reset between each test, if the JMS provider has failed.

[7] The test analysis is dependent, particularly when testing performance, on all system clocks being synchronised. The network time protocol [11] (NTP) provides synchronisation to millisecond accuracy — the precision of the timestamps.

During simple correctness testing, where the number of messages produced is fairly small, the above design is adequate. However, when performance or stress testing is performed, the number of messages is larger. With the larger message load, JDBC represents a bottleneck as each message needs to be loaded into the database[8]. In addition Microsoft Access tends to slow with the larger numbers of records, even when tables and queries are optimised. For performance testing, a database is not really necessary, as only simple statistical information needs to be gathered. This information can be computed by the daemon prince and then inserted into the database.

5 Conclusions and Future Work

This paper has described the design and implementation of a JMS test harness that automates the process of correctness testing and performance evaluation. For JMS vendors, the harness automates the process of component testing. For JMS users, the harness automates independent performance evaluation of a number of JMS implementations. The results is then used to decide which is the most suitable JMS implementation that meets the user's requirements.

We have also shown that, even given the restrictions placed on the harness by the JMS API, it is possible to construct a rigourous set of test criteria for most of the requirements of the JMS specification. The test criteria are basically the formalisation of parts of the JMS specification. In the implementation, these are then translated to SQL statements to test for correctness. Similarly, a set of SQL statements are used to generate performance evaluation reports.

The JMS test harness, at this time, can not initiate a system or program crash and then recover from the failure. This is required to fully test persistent delivery mode. Future work includes adding this feature into the test harness.

There are several other possible areas for future work. The criteria for testing message priority and time-to-live (criteria 4 and 5 in section 3) are not particularly rigourous. The strictness of message priority analysis can be enhanced by building a model that indicates whether two messages are candidates for priority considerations. A more accurate expectation model for message expiry can also be built, allowing more accurate testing. Improving both criteria requires more sophisticated delay expectation models. More sophisticated models can be built either by constructing a histogram of message delays throughout the run period or by using a normal distribution for expected message delay, using the mean and standard deviation for the run.

By the time a JMS provider is released, it is only to be expected that the provider conforms to the JMS specification. However, different providers show markedly different performance characteristics.[9] When a client chooses a JMS provider, a considerable amount of effort needs to be expended in acquiring and configuring JMS providers and running tests that simulate the load that the client's application is expected to produce. The test harness described in this paper allows the automated testing of multiple JMS platforms. The platforms can be configured by a service vendor and a web interface

[8] Analysis is done at the end of each test and before a new test starts and thus does not affect the performance of JMS

[9] A preliminary comparison of MQ Series[5], WebLogic[1] and SoniqMQ[14] JMS providers shows performance differences of a factor of 10 in some cases.

used to describe the type of scenario envisaged. The client can then see the performance results of several systems, allowing them to focus on a few likely candidates for more rigourous evaluation.

References

1. BEA Systems. Programming weblogic 6.0 jms. `http://e-docs.bea.com/wls/docs60/jms/`.
2. P. A. Bernstein. Middleware: A model for distributed system services. *CACM*, 39(2):86–98, 1996.
3. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullendar, editor, *Distributed Systems*, pages 97–145. Addison-Wesley, 2nd edition, 1994.
4. M. Hapner, R. Burridge, and R. Sharma. *Java Message Service (JMS) 1.0.2*. Sun Microsystems, Java Software, Nov. 1999.
5. IBM. Mqseries family. `http://www.ibm.com/software/ts/mqseries/`.
6. N. Kropp, P. Koopman, and D. Siewiorek. Automated robustness testing of off-the-shelf softare components. *Fault Tolerant Computing Symposium*, 1998.
7. D. Kuo and D. Palmer. Automated analysis of java message service providers. Technical Report 01/123, CSIRO Mathematical and Information Sciences, GPO Box 664 Canberra ACT Australia, 2001. `http://www.cmis.csiro.au/adsat/reports`.
8. R. Lee and S. Seligman. *JNDI API Tutorial and Reference*. Addison-Wesley, 2000.
9. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
10. A. G. Mathur, R. W. Hall, F. Jahanian, A. Prakash, and C. Rasmussen. The publish/subscribe paradigm for scalable group collaboration systems. Technical Report CSE-TR-270-95, Department of Electrical Engineering and Computer Science, Univerity of Michigan, Nov. 1995.
11. D. Mills. Network time protocol (ntp). `http://www.eecis.udel.edu/~ntp/`.
12. R. Monson-Haefel and D. A. Chappell. *Java Message Service*. O'Reilly, 2001.
13. T. Ouellette. For many companies, MOM knows best. *Computerworld*, 30(24), 1996.
14. Progress Software. Soniqmq. `http://www.progress.com/sonicmq/`.
15. Sun Microsystems, Java Software. *Java Remote Method Invocation Specification*, 1999. `ftp://ftp.java.sun.com/docs/j2se1.3/rmi-spec-1.3.pdf`.
16. R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report CS0964, Computer Science Department, the Technion – Israel Institute of Technology, Sept. 1999.
17. S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC(TM) API Tutorial and Reference*. Addison-Wesley, 2nd edition, 1999.

Efficient Object Caching for Distributed Java RMI Applications*

John Eberhard** and Anand Tripathi

Department of Computer Science, University of Minnesota,
Minneapolis, MN 55431
{eberhard, tripathi}@cs.umn.edu

Abstract. Java-based[1] distributed applications generally use RMI (Remote Method Invocation) for accessing remote objects. When used in a wide-area environment, the performance of such applications can be poor because of the high latency of RMI. This latency can be reduced by caching objects at the client node. However, the use of caching introduces other issues, including the expense of caching the object as well as the expense of managing the consistency of the object. This paper presents a middleware for object caching in Java RMI-based distributed applications. The mechanisms used by the middleware are fully compatible with Java RMI and are transparent to the clients. Using this middleware, the system designer can select the caching strategy and consistency protocol most appropriate for the application. The paper illustrates the benefits of using these mechanisms to improve the performance of RMI applications.

1 Introduction

Distributed object systems are commonly implemented using some type of remote procedure call, or in the Java case, remote method invocation (RMI) [20]. One problem with using RMI is the high latency associated with invoking a method on an object. This problem is especially severe in wide-area networks, due to large latencies dictated by the distances between sites. To overcome the latency inherent in remote method invocation, the programmer is forced to structure applications so that interactions are reduced, or the programmer must rely on other schemes to reduce latency related problems.

One approach to overcome the latency problem is to cache objects at the client's node. For example, a programmer could structure a client program so that a copy of the object resides at the client. When manually replicating the object in this manner, the programmer must decide the granularity of caching. The programmer must also ensure that the replicated copy of an object remains consistent with its other copies. Obviously, programatically adding caching in

* This work was supported by NSF ITR 0082215.

** Author is currently employed by IBM Rochester, Rochester, MN 55901

[1] Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 15–35, 2001.

this manner is expensive and error-prone. A better solution is the use of a middleware to transparently add caching to an RMI application. This middleware should permit the programmer to choose the caching and consistency protocols best suited to the application.

A middleware that improves the performance of Java RMI through the use of caching must meet certain requirements. First, for caching to be useful, it must be compatible with RMI and transparent to clients currently using RMI. Second, object caching mechanisms must support consistency protocols tailored to both the semantics and usage of an object. Third, when using caching, only the data needed by the client should be cached. One technique to do this is to selectively choose which objects should be cached. Another technique is to cache an object that contains only a portion of the data of the original object. To accomplish this, we introduce the concept of "reduced object."

Based on these requirements, we have designed a set of object caching mechanisms, integrated into a middleware system and a set of tools, for adding caching to any existing Java RMI application. This paper describes the following contributions of our work.

- A set of caching mechanisms transparent to and compatible with the existing RMI clients of an application.
- These mechanisms support integration of different kinds of consistency protocols by the server.
- These mechanisms retain semantics of Java's *wait/notify* synchronization mechanisms.
- A set of tools to create the objects necessary to support caching in Java RMI. These tools create the Java classes necessary to cache an RMI object.
- The notion of *reduced object* is introduced to support caching only parts of an object as needed by a client, to reduce the overheads in caching the full object.
- We present the results of our experimental evaluation of this system using two different kinds of applications, which demonstrate the benefits of these mechanisms for RMI applications.

As shown in this paper, adding caching to an RMI application can be accomplished using a middleware system that easily permits caching and consistency policies to be chosen for RMI objects.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 presents the requirements guiding the development of the RMI caching middleware. Section 4 presents the middleware and associated caching mechanisms. Section 5 illustrates the benefit of this middleware using two different RMI applications. Section 6 presents the conclusions.

2 Related Work

Several researchers have investigated caching to improve the performance of RMI and Java distributed object systems. Other researchers have used caching

in distributed object systems. In comparison to existing work, our mechanisms retain RMI compatibility, permitting existing RMI clients to transparently use caching. Our mechanisms permit different consistency protocols to be associated with different objects. We also support reduced objects, which contain a subset of the state of the original objects. Below, we briefly review the work most relevant to this paper.

In their work on RMI performance, Krishnaswamy, et al., [14] describe a caching system used to improve RMI. Their system differs from our work because they cache the serialized version of the object inside the reference layer of the Java RMI implementation. All interactions with the object take place on the serialized object. A consistency framework assures that the entire serialized object remains consistent with the object on the server. Like their approach, our system retains RMI compatibility. Unlike their approach, our system caches the objects in their unserialized forms. Furthermore, rather than caching complete objects, our system permits caching portions of objects.

In a more recent work, Krishnaswamy, et al., [13] developed object caching for distributed Java applications. Their work is similar to our work in that they permit a variety of consistency protocols to be used to manage cached objects. Their work is concerned with using the quality of service specified by the client to guide caching decisions. While we take a server centric approach, the client can still provide information about its environment. However, unlike our work, they do not examine the impact of caching portions of objects.

Other researchers[3] have implemented Java caching using distributed shared memory. These approaches have required changes to the underlying Java virtual machine and would not be appropriate in a heterogeneous wide-area environment, nor would they be usable by existing Java RMI clients. Unlike their approach, our work does not require changes to the Java virtual machine.

Lipkind, et at., [16], describe a Java distributed object system with caching. Their architecture is based on "object views", where the programmer explicitly states how an object is to be used. The information from the object views is used to optimize the behavior of a distributed shared memory system running on a cluster of workstations. The work presented in this paper differs from their work in that we use RMI for client-server communication, thus permitting its use in a wide-area network.

Another distributed object system is Globe[4][22]. Like the system presented in this paper, several objects are used to represent a cached distributed object. These objects serve roughly the same purpose as the objects in the caching architecture presented in this paper, i.e., a local cache object, a consistency object, and a communications object. However, with their local cache object, they currently do not have a mechanism to use a subset of the instance variables of the classes. The use of reduced objects that contain only a subset of the instance variables of an object is a contribution of our work.

Rover is a distributed object system that caches objects. Its primary goal is support for disconnected operation. It has a single consistency policy that uses optimistic concurrency control to execute methods on a cached object. Requests for method execution are queued for eventual execution at the server. Like most

object systems, Rover caches a complete copy of the object. In contrast, our work provides support for integration of different consistency protocols. As mentioned earlier, our work uses reduced objects to minimize the amount of state cached at the client.

Like many other systems, we use the proxy principle[19] to implement a distributed object system. Like GARF[7], we separate the functionality of the object from other concerns. While GARF was focused on providing fault tolerance, our focus is on providing caching to improve the performance of an existing distributed object system.

A well-known object system is Thor[17]. Thor uses page-based, transactional object caching that uses a single protocol based on optimistic concurrency. Our work differs in that we do not require transactional boundaries and we support multiple consistency protocols. Instead of caching objects as a physical page, our work uses the logical relationships between objects to direct caching decisions.

Our work applies the principle of binary rewriting to creating objects that support caching in a middleware system. Binary rewriting has been used in the past to remove synchronization[1][5] and also to improve the performance of applet-loaded classes[12]. We extend the use of binary rewriting to the creation of objects for caching. As described below, we have developed tools to analyze the byte code in a Java class file and to manipulate that byte code to create the objects in our system. To achieve our byte code manipulation, we use the JavaClass API written by Markus Dahm[6].

3 Background and Requirements

Before undertaking the caching of Java RMI objects, the nature of such objects must be understood. Once this is understood, requirements can be developed to assure efficient and usable caching. These key requirements are RMI compatibility, flexible caching and consistency policy support, and minimal caching. These requirements, their motivating factors, and corresponding challenges are explained below. We also briefly discuss the issue of cache replacement.

3.1 Object Model

A Java RMI object is an object that implements an interface extending the *java.rmi.Remote* interface. The object is accessed at the client using the methods of the interface. To cache an RMI object, the structure of a Java RMI object must be understood.

When viewed in the strictest sense, an object contains fields that are either native types or references to other objects. The Java serialization mechanism serializes an object so that it can be transmitted to another JVM environment. When moving the object to another system, it is a simple matter to copy the native types. However, moving a reference implies also moving the referenced object. Some fields of the object and some referenced objects cannot be serialized. A field designated as *transient* cannot be copied to a client, since the Java serialization mechanism will not serialize a *transient* field when the object is

serialized. Also, an object that is not serializable cannot be cached. An example of this type of object is a FileReader object, which cannot be serialized because it refers to an open file. When a Java RMI object, or *Remote* object, is serialized, the object is replaced with a remote reference.

An object is cached by sending to the client the portions of the object that are expected to be used by the client. Once the object has been cached, the object and associated objects may be changed using methods of the interface. A method manipulates the fields of the object and referenced objects. A native field is manipulated using reads or writes to the field. A reference field can be modified to refer to another object, or a method can be invoked on the referenced object to change that object. If an object is cached, the caching mechanism must have some means of assuring that changes to fields and referenced objects are consistent with its other copies, including the primary copy of the object residing at the server.

Any middleware system that caches RMI objects must be aware of the object structure and provide mechanisms to assure the consistency of the fields of the cached object as well as the serializable objects referenced by the cached object.

In summary, the characteristics of RMI objects that affect caching are the following:

- transient variables cannot be cached
- non-serializable objects cannot be cached
- remote objects are serialized as RMI stubs and accessed via RMI stubs
- the serializable portion of an object as well as the serializable and non-Remote objects referenced by the object can be cached
- the system must assure that the cached object, including referenced objects, remain consistent with other copies

3.2 RMI Compatibility

A key requirement is RMI compatibility. Compatibility should be evidenced in the following ways. Caching should easily be added to an existing RMI application. Objects should be transparently cached at the client in a manner fully compatible with RMI. It should be possible to use caching with some *Remote* objects, yet access other *Remote* objects using RMI. Objects should be accessed using RMI, if caching the object will not improve performance. Security concerns may also prevent an object from being cached.

Because the current goal of our work is to determine the benefits of caching, we require the use of RMI as the underlying communication mechanism. Because prior work has established that RMI can be improved using other techniques[14][18], we wish to show that the benefits of our work are solely due to caching, not other factors.

Meeting these requirements poses several challenges. To be compatible with RMI, a suitable replacement for the standard RMI stub is needed. This stub requires more logic than a normal RMI stub in order to determine whether the stub should be passed as a reference, or if it should contain a cached RMI object. Furthermore, a tool similar to the RMI stub generator, *rmic*, needs to be designed to create these new stubs

3.3 Flexible Usage of Consistency Protocols

Another key requirement is that the caching mechanism should permit the integration of different kinds of consistency protocols, where each object can be assigned a consistency protocol best suited to its behavior. When copies of an object are located at several locations, in general, some mechanism must assure that the various copies are consistent with each other. This can be accomplished by assigning a consistency manager to each object cached on a client. Because it is possible for the object to be accessed at the server, a consistency manager is also required at the server. These consistency managers should be able to use a variety of consistency protocols, permitting each object to use a protocol suited to its semantics and usage.

Because objects differ widely in their usage and semantics, they have different consistency requirements. Consequently, consistency managers should not be tied to a specific consistency model or management protocol. When designing protocols, it should be possible to use a given protocol with a wide variety of objects. For each object in an application, the application developer should be able to select or create a protocol that meets the object's consistency and performance requirements.

Consistency protocols use different and often conflicting techniques. Examples of these techniques[2][15] include *cache invalidation* versus *cache update*, *write through* versus *write back*, and *data shipping* versus *method shipping*. Each of these techniques performs well in some situations and poorly in other situations.

To support multiple consistency protocols, a proper design must permit consistency managers to mediate access to an object and the cached copies of the object. These consistency managers also need to be able to access the state of the object. Developing consistency managers which can be generally applied to a wide variety of objects, yet at the same time have intimate knowledge of the structure of the object is a challenge to design and implement.

3.4 Minimize Cached Data

As mentioned earlier, a Java object may reference other objects. When caching an object, caching all objects referenced by the object is not desirable. Only those remote objects expected to be used should be cached. Other remote objects can either continue to be accessed using RMI or the appropriate mechanisms should be available to transparently cache the object either when it is first referenced or when it is first used. Furthermore, when caching an object, it may not be desirable to cache all contents of the object, since not all instance variables of an object may be used at the client. Consequently, the object cached at the client should only contain those instance variables that will be used.

The selective caching of objects and their instance variables provides efficiency in two ways. First the network overhead of transferring the unused data is eliminated. More important, the overhead of maintaining the consistency of the data is also eliminated. Another reason not to cache an instance variable

or object at the client is security. For example, if an instance variable or object contains sensitive data, it should not be exposed to the client or the network.

To accomplish the selective caching of instance variables of an object, we have developed the concept of a *reduced object*. A reduced object is a version of an object where unused instance variables, as well as methods that use those variables, have been removed. To effectively use reduced objects, several challenges had to be met. Mechanisms to generate the reduced object had to be created, including mechanisms to determine what instance variables should be included in a reduced object.

3.5 Cache Replacement

To simplify our design, we do not consider the issue of cache replacement. We assume that the cache will be large enough to hold all objects that will be cached. Gray and Shenoy suggest that because of the high latency of web access, a "cache everything" strategy should be used when caching web pages[8]. The same factors that influence the "cache everything" strategy for web pages will influence the use of large caches for distributed objects. While we do not consider the eviction of objects from the cache, in our design, objects in the cache are removed by the Java garbage collector when they are no longer in use.

4 Caching Mechanisms

This section presents the mechanisms of our system satisfying the requirements outlined in the previous section. These mechanisms are based on structures called the *cache template* and *server template*, which respectively represent the RMI object at the client and the server. The objects within the cache template and server template work together to manage the global state of the object as well as enable RMI compatibility. We also develop mechanisms that permit the development of consistency protocols that are applicable to a wide variety of objects.

4.1 Caching Structures

To assure consistency, all access to the copies of the object that reside at the clients as well as access to the server object must be mediated and coordinated. While some systems provide either virtual memory mechanisms, such as those available in software distributed shared memory systems like DOSA[10], or reflective mechanisms, like in MetaJava [11], to assist in managing fine grained access to an object, these mechanisms are not available in standard Java. To assure consistency in our system, we use interposition, in which an object manager is interposed between a copy of an object and the users of the object. We use interposition on both the client and the server. The middleware described in this paper uses two primary structures, one on the clients and one on the server, to ensure that all access to the object is mediated. These structures are described below.

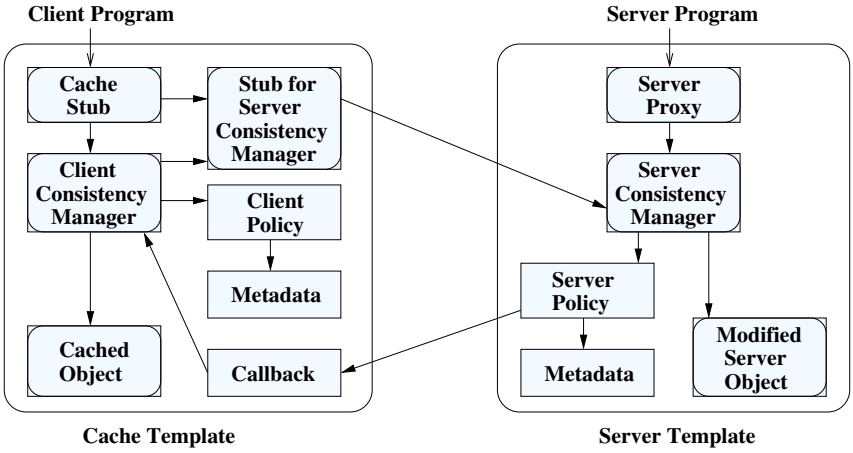


Fig. 1. Cache Template and Server Template

Cache Template. The *Cache Template* is our mechanism to cache RMI objects at a client. Its root object, the *Cache Stub*, replaces the standard RMI stub. The cache template consists of a cacheable version of the RMI object, along with accessory objects that manage the client's access to the cached object. As shown in Figure 1, the cache template consists of seven objects: the *Cached Object,* the *Client Consistency Manager*, the *Metadata*, the *Client Policy*, the *Stub for Server Consistency Manager*, the *Callback,* and the *Cache Stub*. Each of these objects plays a role in satisfying the caching requirements previously described.

The central object in the cache template is the cached object. The cached object is a copy of the object that resides at the server. This copy implements non-Remote versions of the interfaces of the object being cached. These non-Remote versions of the interfaces are identical to the interfaces of the object being cached with the exception that the interfaces no longer extend the *java.rmi.Remote* interface. This must be done because Java RMI will not serialize an object that implements the *java.rmi.Remote* interface (instead it will attempt to serialize the RMI stub for the object).

The cache object may be a reduced object. As mentioned earlier, a reduced object contains a subset of the instance variables of the original RMI object. A tool generates the classes needed for a reduced object. This tool is provided a list of methods that should be enabled in the reduced object. The tool determines which instance variables are needed by those methods and creates a new class that contains only those variables. Because the reduced class must implement all the interfaces of the object, all the method signatures are present. However, unsupported methods are changed so that they throw a "Not Available" exception if called. As explained below, the other objects in the cache template assure that an unsupported method will not be called. The reduced object is also changed

so that it properly supports the Java wait/notify mechanism. This is described below in Section 4.3.

Because the access to the cached object is controlled by other objects in the cache template, if any of the methods of the original object were declared as *synchronized*, that synchronization is no longer needed. Any synchronized methods of the original object have their synchronization removed from the cached object. This implies that the client consistency manager and client policy, described below, are responsible for the correct ordering of any methods invoked on the cached object. This includes the assurance that the state of the cached object is consistent.

Since other objects in the cache template need to update the fields of the object, the fields must be accessible. This is accomplished by changing the access privileges of any private fields in the RMI object to *package private*.

Since the cached object must be kept consistent with the object at the server, a direct reference to the cached object is not provided to the client program. Instead, a *client consistency manager* is interposed between the client program and the cached object. Like the cached object, the client consistency manager implements a non-Remote version of the interfaces of the object being cached. The client consistency manager works in conjunction with a *Client Policy* to maintain the consistency of the cached object. For each method invoked on the object, the client consistency manager consults the client policy, which determines what actions should take place to handle the method invocation. The client policy directs the actions of the client consistency manager using an *ActionsList*, which is discussed below. For a reduced object, the client policy assures that a method that is not supported by the reduced object will be executed on the server instead of on the client. In summary, the purpose of the client consistency manager is to enable the use of a generic policy with a specific object.

The client policy needs information about the object to guide the actions of the client consistency manager. This information is contained in the metadata object. The metadata object describes each method of the reduced object in terms of the instance variables which are accessed by the method and the manner in which each instance variable is used. Using the information about how each method uses the instance variables of the object, the metadata also classifies a method into one of the following five categories:

- NONE
  This type of method does not use any instance variables of the reduced object. Consequently, this type of method may be safely invoked at any time.
- IMMUTABLE
  This type of method only uses immutable instance variables of the reduced object. Since immutable instance variables never change, they are always valid if the object has been cached.
- READ-ONLY
  This type of method only reads instance variables of the reduced object. Since this type of method may read instance variables that may be changed

by other clients, the policy must assure that the instance variables read by the method are consistent.

- READ-WRITE
  This type of method both reads and writes instance variables of the reduced object. Since this type of method may change an object, the policy must assure that the invocation of this type of method will cause the object to remain consistent.
- SERVER-ONLY
  This type of method must be executed on the server because some instance variables used by the method are not available in the reduced object. A client policy will cause this type of method to be executed at the server.

The metadata for the reduced object is created by a tool which analyzes the byte codes of a Java class to determine which instance variables of the object are used by each supported method. Since an instance variables is only accessed using the "PUTFIELD" and "GETFIELD" byte codes, this analysis is fairly straight forward. Using this information, the methods of the reduced object are categorized into one of the above categories.

The client consistency manager communicates with the server using a *Stub for the Server Consistency Manager*. This object is a standard Java RMI stub for an object on the server, the server consistency manager, which is described below. It implements an interface that contains a modified version of all *Remote* methods of the object being cached. For each method, two changes are made. First, an additional *client context* parameter is added to the interface. The client context passes information to the server about the context in which the method is being called. This parameter is set by the client policy. One example of the context, which is always passed to the server, is the identity of the client. The second change to the interface is that an actions list is returned. This actions list permits the server to control the behavior of the client consistency manager. More details about the actions list are provided below.

Because the server may need to communicate with the client consistency manager, the cache template contains a *Callback* object to which the server may send requests. The callback object accepts remote requests from the server and forwards those requests to the client consistency manager. This permits the server to send requests such as cache updates and cache invalidations.

The client program accesses the object using a *Cache Stub*. The main purpose of the cache stub is to ensure RMI transparency and compatibility. The cache stub replaces the RMI stub found in standard RMI. The cache stub contains logic to assure that it is serialized in the correct form when passed via an RMI method. When passed to a client, the cache stub causes the sending of either an entire cache template or a *partial cache template*, consisting only of the cache stub and stub for server consistency manager. For example, if the cache stub is being sent to the RMI name registry, then the partial cache template is sent. Clients then receive the partial cache template when they lookup the object from the name registry. If the partial cache template is accessed by a client, the cache stub contains the logic needed to contact the server, using the stub for the

Table 1. Primary Function of Objects in Cache Template

Object	**Primary Function**
Cached Object	Contains state of cached RMI object
Client Consistency Manager	Manages access to the cache object based on the client policy
Client Policy	Defines in an object-independent manner how object is managed
Metadata	Describes the cached object
Callback	Permits the server to contact the client
Stub for Server Consistency Manager	Provides communication with the server
Cache Stub	Provides RMI compatibility and object faulting

server consistency manager, and request the object. This we refer to as *object fault handling*.

The cache stub also overcomes one of the shortcomings of the current RMI implementation. Suppose, a server receives through an RMI call, either as a parameter or return value, a reference to an RMI object residing at the server. If the server uses this reference, which is an RMI stub, to access the object, the communication overhead of RMI will be incurred. However, if a cache stub is received on the server, its deserialization routine will use the ID of the cached object to determine if it resides on the server. If so, it will allow itself to directly access the consistency manager for that object, thus avoiding unnecessary communication overhead.

The classes for the cache stub, client consistency manager, cache object, and stub for server consistency manager must be created for each class of object being cached. To facilitate the creation of the appropriate classes, we have developed suitable code generators. These generators use the Java class file of the remote object as input and produce the necessary classes.

The objects in the cache template permit the caching of an RMI object and control when a method is invoked on the cache object, assuring that the cache object is in a consistent state. In summary, the objects in the cache template and their functions are given in Table 1.

Server Template. The *Server Template* is the mechanism used to manage the primary object residing at the server. As shown in Figure 1, it consists of five objects. The original remote object is replaced by a modified server object. Like the cache template, the server template also has consistency manager and policy objects. To assure RMI compatibility, the server template also has a proxy object.

The modified server object is a slightly modified version of the original RMI remote object. A modified version of the object is needed for many of the same reasons that the client contains a modified version of the object. These reasons include changing the access privileges of instance variables and replacing synchronization primitives with primitives described in section 4.3.

Table 2. Primary Function of Objects in Server Template

Object	Primary Function
Modified Server Object	Maintains state of the object at the server
Server Consistency Manager	Manages access to the modified server object
Server Policy	Dictates in an object-independent manner how object is managed
Metadata	Describes the server object
Server Proxy	Provides RMI compatibility

```
public void sampleMethod(String inputParm) {
  actionsList = clientPolicy.getActionsList(methodId);
  executeActionsList(actionsList);
}
```

Fig. 2. General Structure of a Method of the Client Consistency Manager

The server consistency manager plays the same role as the client consistency manager on the client. It manages method requests from the server and clients. Like the client consistency manager, on each request it consults a server policy. The server policy returns an actions list to direct the behavior of the server consistency manager. Like the client policy, the server policy uses a metadata object to guide its decisions.

At the server, the object is accessed using the *Server Proxy*. The purpose of the server proxy is to provide RMI compatibility. If it is passed via RMI, RMI serialization will serialize the cache stub of the cache template. In other words, the RMI system considers the server proxy to be a Remote object that has the *Cache Stub* as its stub.

To use the cached template, minimal changes to the server are needed. First, a modified server object must be created instead of the original RMI objects. Second, instead of exporting the object using the export method of UnicastRemoteObject, the application must export the object using the export method of the CacheableObject class, which we provide.

In summary, the objects in the server template and their functions are shown in Table 2.

4.2 A Framework for Consistency Management

As mentioned earlier, the purpose of the client consistency manager is to intercept, via interposition, each method invocation. It consults the client policy for the actions to take to handle the method invocation. Figure 2 provides an example of the structure of a method in the client consistency manager. This method retrieves an actions list from the client policy and executes the actions in this list.

An *ActionsList* is a list of actions that are to be executed by a client consistency manager. We have identified several actions used to control a client consistency manager. Some of these actions are listed in Table 3.

Table 3. Actions Executable by Client Consistency Manager.

Action	Description
1. Invoke method	Invoke the method on the locally cached object.
2. Get actions list from server	Contact the server, passing the method id and the client context. Returns an action list.
3. Invoke method on server	Invoke the method on the server using RMI, passing method arguments and the client context. Returns an action list.
4. Set return value	Sets the return value of the method.
5. Update variable	Updates an instance variable of the cached object to a value.
6. Update policy state	Updates the state of the client policy. (This is explained in Section 4.2)

When a client policy is consulted by a client consistency manager, it considers the state of the cached object as well as the characteristics of the method. If the client policy determines that a method can be safely invoked (for example, the method is an IMMUTABLE method), it returns the "Invoke Method" action. Otherwise it may return an action to contact the server, either action 2 or 3 in Table 3. For these actions, the server consistency manager returns an additional actions list to be executed by the client consistency manager. For example, this list could contain actions to set the return value of the method, to update the instance variables of the cached object, and to update the state of the client policy. For example, this state could be used to track which instance variables of the cached object are valid.

Examples of actions executed by the client for certain scenarios are given in Table 4. For example, in scenario 1, a method only uses immutable instance variables. The client policy returns the action to invoke the method locally on the cached object. In scenario 2, a method uses instance variables that are only accessible at the server. The client policy returns an action to invoke the method on the server. After executing the method, the server returns an action list to set the return value of the method. In scenario 3, the object state used by the method is invalid. The client policy returns an action to get an actions list from the server. The server returns actions to update the instance variables of the cached object, update the policy state to indicate that the cached object is valid, and finally to invoke the method on the cached object.

The goal of the consistency mechanisms is to enable actions to be taken both before and after the method on the cached object is invoked. The client consistency manager is directed by the client policy as well as the server policy objects. The policy objects at the client and the server cooperate to assure that the consistency and synchronization requirements associated with the object are met.

Table 4. Examples of actions executed by client consistency manager for particular situations. Actions in SMALL CAPITAL LETTERS indicate actions from the client policy. Actions with an asterisk (*) and in *italics* were received from the server.

Scenario	**Actions Executed by Client Consistency Manager**
1. Method uses immutable instance variables	INVOKE METHOD LOCALLY
2. Method uses instance variables only available at server	INVOKE METHOD ON SERVER * *Set return value*
3. Method uses instance variables that are not valid at the client	GET ACTIONS LIST FROM SERVER * *Update variable* * *Update policy state* * *Invoke method locally*

4.3 Wait and Notify Support

This middleware assures consistency by controlling the principal entry points into a cached object, namely the method entry and exit. However, an object may be entered and exited in another manner. This is through the Java *wait* and *notify* mechanisms. The system must ensure that the semantics of *wait* and *notify* are preserved. To accomplish this, our tools modify the cached object (as well as the server object) such that the invocations of the *wait* and *notify* methods are respectively replaced by invocations of the *handleWait* and *handleNotify* methods of the consistency manager.

When a *handleWait* method of the client consistency manager is invoked, it asks the client policy object for an actions list. This actions list typically contains an action to inform the server that a client object has executed the *wait* method.

The *handleNotify* method must be implemented differently. Java semantics dictate that when a thread issues a *notify,* a waiting thread will awake and acquire the monitor lock after the current thread releases the monitor lock. In the most common case, this occurs when a synchronized method returns. For this reason, the *handleNotify* method sets a flag to indicate that *notify* has been invoked. After the method returns, if this flag is set, the client consistency manager executes an actions list that it obtains from the client policy.

Figure 3 shows the sequence of method calls that take place in order to handle *wait* and *notify*. In general, a *wait* invoked on a client causes that client to wait. That client is awakened after another client invokes *notify*. For clarity, interactions with client policy and server policy are not shown. In step 1, a method of the cached object is called on client A. Where the original object called *wait*, the cache object now invokes the *handleWait* method, as shown in step 2. In step 3, the client consistency manager notifies the server that there is a client waiting, returning any object state that has been modified at the client. The method invocation of client A then blocks at the server. In step 4, the client consistency manager of client B invokes a method on the cached object. In step 5, this method invokes the *handleNotify* method in the place where the original object called *notify*. As described above, a flag is set to indicate that a notify is pending. In step 6, the object returns from the method that invoked

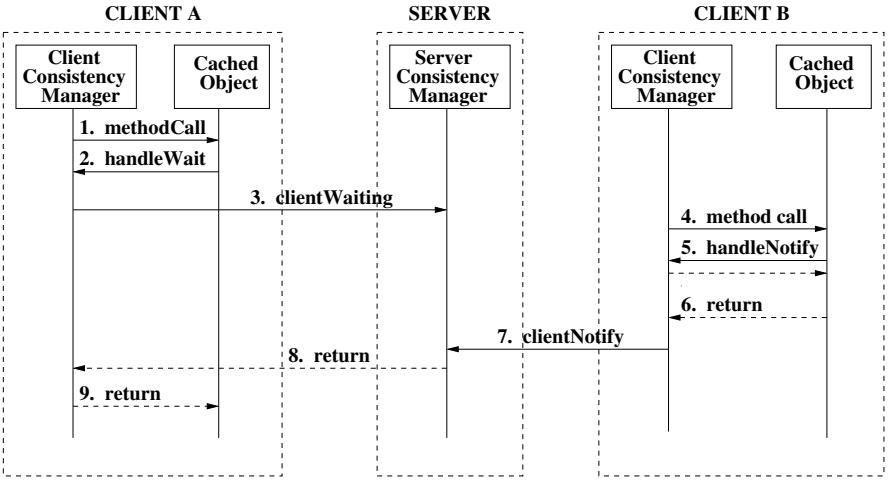


Fig. 3. Sequence of Method Calls for Wait Notify

handleNotify. At this point, as shown in step 7, the client consistency manager notifies the server that a notify has occurred, and passed any updates that have been made. The server consistency manager then wakes up the waiting thread of client A as shown in steps 8 and 9. The thread then continues executing after the *handleWait* call in the cached object.

5 Experimental Results

To evaluate our mechanisms, we used two different RMI applications. The first is a simple producer-consumer application that is used to determine compatibility of our system with the Java wait/notify mechanisms. The second is a medium size object benchmark similar to the TPC-C benchmark[21]. In this benchmark we show the use of two different consistency policies and reduced objects. We use these two applications to determine the baseline performance of RMI. We then compare this baseline with the performance of these applications after caching has been added.

5.1 Distributed Producer-Consumer Application

We use a simple producer-consumer application to evaluate our support for Java wait-notify. In this application, a server exports a buffer object that is accessed by two clients. The *produce* and *consume* methods of the buffer object are shown in Figure 4.

To cache this object, we designed client policy and server policy objects that ensure that the buffer can only be cached at one client at a time. These policy objects were also designed to minimize communication with the server, when the

```
synchronized public void produce(Object item) {
    while (produceAt >= (consumeAt + items.length)) wait();
    items[produceAt % items.length] = item;
    produceAt++;
    notify();
}
synchronized public Object consume() {
    while (consumeAt == produceAt)  wait();
    Object item = items[consumeAt % items.length];
    consumeAt++;
    notify();
    return item;
}
```

Fig. 4. Producer Consumer Object

Table 5. Performance of Producer Consumer Application

Buffer Size	RMI	Caching
64	4311 ms	3893 ms
512	4202 ms	3450 ms
2048	4281 ms	2824 ms

buffer methods are invoked by a producer and a consumer. A *wait* method is processed as shown in Figure 3. However, the sending of a *clientNotify* message is deferred until either a *wait* method is called or until the object has not be used for 100 milliseconds. At that time, the cached object is invalidated and the changes to the object are returned to the server. Overall, this protocol permits the producer to produce many objects before the cached object is returned to the server. The server then updates the buffer at the consumer and the consumer can consume many objects before releasing the object. We call this policy the *delayed notify* policy.

For our experiments, the producer creates 4096 string objects. We measure the time from when the producer deposits the first item to the time when the consumer retrieves the last item. We verify correctness by assuring the sequence of objects consumed is the same as the sequence of objects produced.

We evaluated the performance of our caching mechanism in a LAN environment, using various buffer sizes in the object. In this environment, the server, producer, and consumer were connected via a 100 Mbps switched Ethernet connection. We compared the performance of RMI with the performance of our caching system. The results of our experiments are shown in Table 5.

As seen in the table, the use of caching improves performance as the buffer size increases. This is because with larger buffer sizes, network interactions are reduced. With a buffer size of 64, the performance improved by 10% and with a buffer size of 2048, the performance gain was 34%.

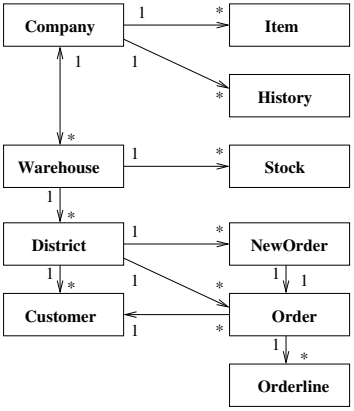


Fig. 5. Objects in the rmiBOB Benchmark

5.2 Object Benchmark

To measure the performance of our mechanism, we created rmiBOB, a port (using Java RMI) of the Business Object Benchmark (BOB) benchmark from the book Enterprise Java Performance [9]. The benchmark implements the business logic in the TPC-C benchmark[21]. The benchmark creates a "company" and associated objects. The benchmark measures the performance of a set of "transaction" operations on the objects.

We ported the BOB benchmark to RMI by changing the primary objects in the application to be RMI objects. The primary objects in the benchmark are shown in Figure 5. There is one company object that has the items sold by the company as well as a history of payments made by customer. This company has warehouses that stock the items sold by the company. Each warehouse has sales districts, with customers assigned to each sales districts. Each district has orders that are placed by customers, and each order contains a number of order lines. For our tests, we used an initial population of 6,141 objects, as shown in Table 6.

The benchmark executes five types of "transaction" operations on these objects. The transactions are executed such that for every 23 transactions the following number of transactions are executed: 10 new order, 10 payment, 1 order status, 1 delivery, and 1 stock level.

We modified the benchmark to run 1000 transactions. To verify that the consistency protocols were working correctly, we enabled screen writes and saved the output to a file. At the end of the run, we compare the output to a previous RMI run to assure that the results are correct.

5.3 Baseline Performance

To get an idea of the overhead of the architecture, we performed two experiments. We first measured the performance when the benchmark used RMI. We

Table 6. Initial Number of Objects

Class	Count
Company	1
Item	1000
History	300
Warehouse	1
Stock	1000
District	10
Customer	300
NewOrder	210
Order	300
Orderline	3019
Total	6141

Table 7. Performance Comparison of RMI and the Middleware with Caching Disabled

configuration	average transaction	server calls	average server call
rmi	187.947 ms	-	-
middleware with no caching	187.561 ms	79632	2.589ms

then measured the performance when the benchmark used our mechanisms with caching disabled. We disabled caching by using a client policy that causes all method invocations to go to the server. The performance of these two configurations is shown in Table 7. The reader will notice that our mechanism performs slightly better than the RMI version. This improvement is due to the benefit, described in section 4.1, of directly accessing the object when a cache stub is returned to the server.

5.4 Caching Using Consistency Protocols

To experiment with our mechanisms, we implemented two simple consistency protocols. These protocols ensure serial consistency by assuring that all writes and reads are coordinated.

The first protocol we implemented is a *server-write* protocol. In this protocol, all methods that change the state of an object are considered write methods. All write methods go to the server, which invalidates all clients using the object. The updates to the object are returned to the client which called the method. Other clients are updated when they access the object.

The second protocol we implemented is a *multiple-readers / single-writer* (MRSW) protocol. If a client accesses an object in write mode, it implicitly obtains a lock for the object and can write the object locally. If the server or another client then accesses the object, the lock is recalled by the server using the callback object and the updates are returned to the server.

We experimented with these protocols to select the most appropriate protocols for each object. In our application, since all the objects of the same class

Table 8. Consistency Protocols Used for Caching Objects

Object Class	Protocol
Company	Server Write
Warehouse	Server Write
Stock	Server Write
Customer	MRSW
Orderline	ServerWrite

Table 9. Performance of Caching and Reduced Objects

configuration	average transaction	server calls	average server call
caching	126.060 ms	15931	9.744 ms
reduced object	117.524 ms	14681	8.90 ms

were used in the same manner, we used the class as the basis for selecting the protocol. We selected the combination shown in Table 8 as the best mix. The results of this combination is shown in Table 9. Using this mix, we measured an average transaction cost of 126.06 ms, which is 33% less than the RMI transaction cost.

5.5 Benefits of Reduced Object

We then performed an experiment to determine the benefit of reduced object caching. We first ran our benchmark using a client policy that recorded which methods where being used on the cached objects. Then, using the list of methods used by each object as input to our tools, we created reduced objects for the objects in Table 8. The results of the experiment are shown in Table 9. The use of reduced objects had the following impacts. The average transaction cost fell to 117.524 milliseconds, which is 37% less than the RMI cost. When compared to caching without reduced objects, the number of server calls dropped by 8% and the average cost of a call to the server dropped by 9%. This improvement is due to the removal of false sharing between the client and the server. Since reduced objects were used, the cached items were not invalidated when the server used portions of the object that were not cached at the client. Since the client cache was not invalidated, the client did not need to contact the server to request a valid copy before executing a method locally.

6 Conclusion

In this paper, we have presented requirements and mechanisms for efficient caching of objects for Java RMI applications. Using these mechanisms, caching can be easily and transparently added to existing RMI applications, while preserving RMI compatibility. This includes the ability to support the Java *wait* and *notify* mechanisms. No changes are required to existing clients and only

minimal changes are required at the server. The central mechanism on the client is the *cache template* containing three important objects: a cache stub ensuring RMI compatibility, a client policy implementing a consistency protocol, and a *reduced object* containing a subset of the state of the object being cached. On the server, similar objects exist in the *server template*.

Using these mechanisms in conjunction with appropriate consistency and caching policies, we illustrated the benefits of caching using two RMI applications. Our first experiment used a *delayed notify* policy to improve the performance of a producer-consumer application by 34%. Our second experiment involved a medium sized object benchmark. We showed how *server write* and *multiple reader single writer* protocols could be used on the same application to improve performance by 33%. We also show how using a *reduced object* improved performance by 37%.

These mechanisms serve as the basis of our ongoing research in caching of Java objects. While our experience showed performance improvements in a LAN environment, we can expect better performance when using caching in wide-area networks. We plan to expand our work to include consistency protocols suitable for use in a wide-area network environment, including the support of distributed collaborative applications. We also plan to expand our work to permit the reduced object to change dynamically as the client's usage of the object changes.

References

1. Aldrich, J., Chambers, C., Sirer, E. G., and Eggers, S. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium* (Venezia, Italy, Sept. 1999), pp. 19–38.
2. Archibald, J., and Baer, J.-L. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems 4*, 4 (Nov. 1986), 278–298.
3. Aridor, Y., Factor, M., Teperman, A., Eliam, T., and Schuster, A. A high performance cluster JVM presenting a pure single system image. In *Proc of 2000 Java grande conference* (San Francisco, CA, June 2000), pp. 168–177.
4. Bakker, A., van Steen, M., and Tanenbaum, A. S. From remote objects to physically distributed objects. In *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems* (Cape Town, South Africa, Dec. 1999), pp. 47–52.
5. Bogda, J., and Hölzle, U. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Denver, Colorado, Nov. 1999), pp. 35–46.
6. Dahm, M. Byte code engineering with the JavaClass API. Tech. Rep. B-17-98, Freie Universitat Berlin, July 1999.
7. Garbinato, B., Guerraoui, R., and Masouni, K. R. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering Journal 1*, 1 (Mar. 1995).

8. Gray, J., and Shenoy, P. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering* (San Diego, CA, Feb. 2000), pp. 3–12.
9. Halter, S. L., and Munroe, S. J. *Enterprise Java Performance*. Prentice Hall PTR, 2000.
10. Hu, Y. C., Yu, W., Cox, A., Wallach, D., and Zwaenepoel, W. Runtime support for distributed sharing in typed languages. In *Proceedings of LCR2000: The Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers* (Rochester, NY, May 2000).
11. Kleinöder, J., and Golm, M. MetaJava: An efficient run-time meta architecture for Java. In *Proceedings of the Fifth Workshop on Object-Orientation in Operating Systems (IWOOOS '96)* (Seattle, Washington, Oct. 1996), pp. 54–61.
12. Krintz, C., Calder, B., and Hölzle, U. Reducing transfer delay using Java class file splitting and prefetching. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Denver, Colorado, Nov. 1999), pp. 276–291.
13. Krishnaswamy, V., Ganev, I. B., Dharap, J. M., and Ahamad, M. Distributed object implementations for interactive application. In *Proceeeding of Middleware 2000* (New York, New York, Apr. 2000), pp. 45–70.
14. Krishnaswamy, V., Walther, D., Bhola, S., Bommaiah, E., Riley, G., Topol, B., and Ahamad, M. Efficient implementation of Java remote method invocation (RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)* (Sante Fe, New Mexico, Apr. 1998).
15. Levy, E., and Silberschatz, A. Distributed file systems: concepts and examples. *ACM Computing Surveys 22*, 4 (Dec. 1990), 321–374.
16. Lipkind, I., Pechtchanski, I., and Karamcheti, V. Object views: Language support for intelligent object caching in parallel and distributed computations. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Denver, Colorado, November 1999), pp. 447–460.
17. Liskov, B., Castro, M., Shrira, L., and Adya, A. Providing persistent objects in distributed systems. In *Proceedings of ECOOP'99* (Lisbon, Portugal, June 1999), pp. 230–257.
18. Nester, C., Philippsen, M., and Haumacher, B. A more efficient RMI for Java. In *Proceedings of The ACM 1999 Java Grande Conference* (San Francisco, June 1999), pp. 153–159.
19. Shapiro, M. Structure and encapuslation in distributed system – the proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems* (May 1986).
20. SUN Microsystems. Java remote method invocation specification, 1997. http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html
21. Transaction Processing Performance Council. TPC benchmark C. http://www.tpc.org/cspec.html, 1999.
22. van Steen, M., Homburg, P., and Tanenbaum, A. S. Globe: A wide-area distributed system. *IEEE Concurrency 7*, 1 (Mar. 1999), 70–78.

Entity Bean A, B, C's: Enterprise Java Beans Commit Options[1] and Caching

Paul Brebner and Shuping Ran

Software Architectures and Component Technologies Group
CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra, AUSTRALIA
{Paul.Brebner, Shuping.Ran}@cmis.csiro.au

Abstract. Entity Beans provide both data persistence and the possibility of caching objects and data in the middle-tier. The EJB 1.1 specification has three commit options which determine how EJBs are cached across transactions: Option C pools objects without identity; Option B caches objects with identity; Option A caches objects and data. This paper explores the impact on performance of these different commit options, pool and cache sizes on a realistic application using the Borland Application Server.

1 Introduction

Entity Beans are part of the Enterprise Java Beans (EJB) 1.1. specification [1] and provide a data persistence mechanism using a relational database. In conjunction with Container Managed Persistence (CMP) the Entity bean lifecycle also allows for caching in the EJB container. The provision for caching in the EJB model is consistent with previous research which suggests that caching is a critical requirement for the performance of object-oriented systems in general [2, 3], and the performance and scalability of distributed object systems such as Corba [4, 5], and Java/EJB [6, 7].

Three commit options (A, B, C) determine what is cached across transactions: objects without identity (C); objects with identity (B); objects with data (A). However, the specification does not mandate support for all three options. Vendors can choose to implement one or more commit options. Some vendors have chosen to support all three options to distinguish their products from others (e.g. Borland Application Server [8]), or only two options (WebSphere), while other vendors either don't support commit options explicitly (E.g. WebLogic) or only implement option C (the simplest) on the basis that object creation is cheap relative to database access, and object caching doesn't result in any performance improvement (e.g. SilverStream [9], iPlanet).

While conducting performance evaluations of a number of EJB products we experimented with various deployment settings for Entity beans. We varied commit options, and pool and cache sizes to determine their impact on performance, and to find optimal settings. During the course of these experiments we discovered a

[1] EJB Entity Beans can have one of three commit options: A, B and C.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 36-55, 2001.

number of interesting features of the interaction between the products, application, commit options, pool and cache sizes, and the way the tests were run. This paper reports on results for Borland Application Server (BAS), version 4.5.0[2].

The following sections explain in more depth the Entity bean lifecycle, and how it enables object pooling, and object and data caching. *Stock-OnLine* is the test application used [10], and the overall requirements are described, along with the Entity bean implementation and the test setup used. Finally, the four experiments performed are described and the results presented and analysed.

2 Entity Bean Lifecycle and Caching

2.1 Lifecycle and Persistence

The main purpose of Container Managed Persistence (CMP) Entity beans is to provide automatic persistence for their data state in a relational database. This simplifies programming and CMP Entity beans are portable across all the databases supported by EJB 1.1. compliant containers.

Entity beans can be in one of three states: Does not exist; Pooled; Ready (Figure 1).

Instances in the pool are not associated with any particular object - they don't have a primary key. Any pooled instance may be used to execute the entity bean's finder methods.

Pooled instances move to the Ready state when they are Created or Activated, but only if another object with the same identity is not already in the ready state. Once in the Ready state an instance has a particular object identity (it has a primary key), and Load and Store (to synchronise the data state with the database) and Business methods may be called on it.

A Ready instance can be moved back to the Pooled state by:

- Passivation: Disassociating the instance from the object identity (so it has no primary key), possibly even during a transaction, or
- Removal: Removing the entity object including the object's representation in the underlying database.

In theory instances can be activated and passivated on demand, in which case the data state is also refreshed (Load) or saved (Store). After the *unsetEntityContext* event objects are in the "does not exist" state and may be garbage collected.

Note that even though we talk about pooled instances, the container is not required to actually maintain a pool of instances - this is an implementation detail.

---

[2] Ideally the experiments would be repeated on a number of different products. However, not all products explicitly support all three commit options, and other significant differences such as whether pool and cache sizes are per bean or per container make comparisons difficult.

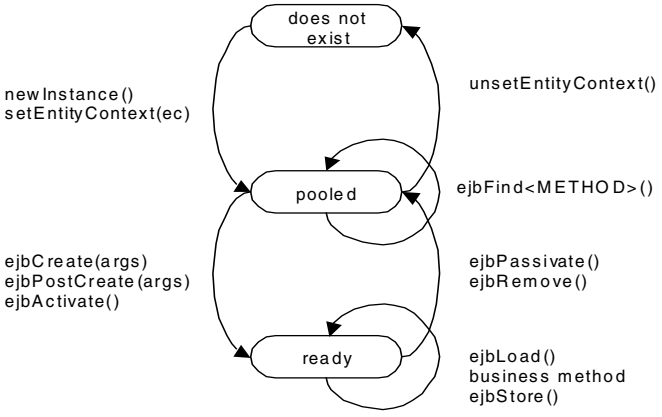


Fig. 1. Life Cycle of an Entity Bean instance (From Figure 23, Section 9.1.4. EJB 1.1. Specification)

2.2 Lifecycle and Caching

Entity beans can therefore do more than just automatically persist data. Having the Ready and Pooled states allows for *caching of Entity beans* (with and without identity) and also *caching of data* in the middle-tier.

There are three "commit options" (what the container does at transaction commit-time) known as A, B and C. These control object and data state across transaction boundaries:

- Object (*Ready*) state: Commit Options A and B maintain the object in the Ready state (activated, with primary key) across transaction boundaries. Option C doesn't, but returns it to the Pooled state (Passivation).
- Data (*Instance*) state: Options B and C do not maintain data state across transactions, but assume that the database may have been changed by another application (*shared*). They load the data state at the start of every transaction (Obviously required for Option C as there is no object in the ready state to cache the data state). Option A assumes the container has *exclusive* access to the database, allowing it to cache a ready instance with data across transactions. That is, while there is a Ready instance the database is not read again. This means that if the database is changed the cached data state will be out of synchronisation.

Option A caches data and object state, Option B caches only object state, and Option C does neither but must get/put instances from/to the Pool at the start/end of every transaction. For all the commit options the data is always written back to the database (Store) at the end of transactions (unless the container has options such as "read only" fields, or checks to see if data has actually changed before writing).

Products such as Borland Application Server that support all three commit options allow the impact of object pooling and caching, and data state caching on the performance of an experimental application to be explored. The *Stock-OnLine* application and the CMP Entity bean implementation of it will now be described.

3 The Sample Application: Stock-OnLine

3.1 Requirements

Stock-OnLine [10] is a simulation of a simple on-line stockbroking system. It enables subscribers to buy and sell stock, inquire about the up-to-date prices of particular stocks, and get a holding statement detailing the stocks they currently own. From a subscriber's perspective, the following services are offered:

- **Create Account:** A person wishing to enrol with Stock-OnLine can create themselves a subscriber account with the service provider.
- **Update Account:** A subscriber can modify their allocated credit limit.
- **Query Stock Value:** A subscriber can query the current price for a given stock. A unique identifier code, or a mnemonic code can be used to identify the stock value to be retrieved.
- **Buy Stock:** A subscriber can place an order to buy a given number of a specified stock. If successful, a transaction record is created for later processing.
- **Sell Stock:** A subscriber can place a request to sell a specified number of any stock item they have purchased through the Stock-OnLine. If successful, a transaction record is created for later processing.
- **Get Holding Statement:** A subscriber can request a statement of all the stock they have purchased through Stock-OnLine and still retain ownership of.

3.2 Database Design

In a real implementation of an on-line stockbroking system, the database would need to store many details in order to track customers, their transactions, payments, and so on. In the example used for performance measurements a simple database design has been used that contains the minimum tables and fields to allow the system to operate sensibly.

The *SubAccount* table holds basic information on a subscriber to the system, and has 4 fields. The primary key for *SubAccount* is the subscriber's account number, *sub_accno*. When a new account is created, the system needs to allocate a new, unique account number, which is done with Oracle Sequences.

Information about each stock that a subscriber can trade through Stock-Online is held in the *StockItem* table. The primary key is *stock_id*, and the other fields represent the stock's trading code, company name, and current value and recent high and low values (a total of 6 fields).

The *StockHolding* table contains information on the amount of a given stock that a subscriber holds, and has 3 fields. The primary key is compound: *sub_accno, stock_id.*

Finally, there is the *StockTransaction* table, which contains information on each transaction that a subscriber performs. The primary key is *trans_id,* which is generated in a similar manner to new accounts using Oracle sequences. The *trans_type* is a code that represents a buy or a sell transaction. Other fields record the subscriber who performed the transaction; the stock item sold or purchased, the amount of stock involved, the price and the date of the transaction.

3.3 Transaction Business Logic

An overview of the business logic for each transaction is given below. The descriptions focus on the database activity that each transaction performs, as these are the most expensive operations. Essentially each action below represents an SQL operation. Exception cases are not described, even though these are handled in the application.

- **Create Account**
  - Get a new account key
  - Insert a new SubAccount record for the new subscriber
- **Update Account**
  - Update the subscriber's credit record in the SubAccount table
- **QueryById**
  - Use the supplied stock identifier to retrieve the current, high and low values from the StockItem table using the primary key
- **QueryByCode**
  - Use the supplied stock code identifier to retrieve the current, high and low values from the StockItem table
- **BuyStock**
  - Read the SubAccount table to retrieve the credit limit for the subscriber, and ensure they have sufficient credit to make the purchase
  - Read the StockItem table to retrieve the current price of the stock the subscriber wishes to purchase
  - If the subscriber has not purchased this stock item before, insert a new record in the StockHolding table to reflect the purchase. If they do hold this stock already, update the record that already exists in the StockHolding table.
  - Get a new transaction key
  - Insert a new record in the StockTransaction table to create a permanent record of the purchase
- **SellStock**
  - Read the StockHolding table to ensure that the subscriber has sufficient holdings of this stock to sell the specified amount
  - Read the StockItem table to retrieve the current price of the stock the subscriber wishes to sell

  - Update the StockHolding table to reflect the sale of some of this stock item by the subscriber
  - Get a new transaction key
  - Insert a new record in the StockTransaction table to create a permanent record of the sale
- **GetHolding Statement**
  - Read the StockHolding table and retrieve up to 20 StockHolding records for the subscriber

The *Buy* and *Sell* transactions are more heavyweight in their demands for database operations. *CreateAccount* and *Update* perform database modifications, but are relatively lightweight. The remaining 3 transactions are all read-only, and should therefore execute quickly.

3.4 Database Initial Population

Prior to each test run, the database is populated with initial test data. Table 1 shows the database tables and their cardinality (rows).

Table 1. Initial database population

Table	Initial Number of Records
SubAccount	3000
StockItem	3000
StockHolding	3000*10 = 30,000
StockTransaction	0

3.5 Client Test Behaviour

The driver for the test application is a multi-threaded Java program running on a separate machine over RMI-IIOP. Each client thread/process performs a number of iterations of a fixed *transaction mix*. The transaction mix represents the concept of one complete business cycle at the client side. The number of transactions of each type per iteration is shown in Table 2.

Table 2. Test transaction mix

Number	Transaction
1	CreateAccount
3	Buy
3	Sell
1	Update
15	QueryById
15	QueryByCode,
5	GetHoldingStatement

This transaction mix comprises 43 individual transactions, and is a combination of mainly read-only (81%) transactions and some update (19%) transactions. Each client performs this transaction mix 10 times, with no wait times. The accounts and items use in each transaction are chosen at random, so there is no locality of reference. The only exception is the buy and sells, which are paired so that whatever is bought is sold again in the same iteration.

3.6 Entity Bean Architecture

The Container Managed Persistence (CMP) Entity bean version of *Stock-OnLine* was designed using the standard EJB design pattern of a Session bean front-end to the Entity beans. This is to prevent the client code from interacting with the Entity beans directly, only via the Session bean business methods. A single session bean implements all seven transaction business methods. Four Entity beans, one each for the database tables, manage the database access (see Figure 2). Transactions are container managed, so when a Session bean business method is invoked a new transaction is started, and all the Entity beans called participate in that transaction context.

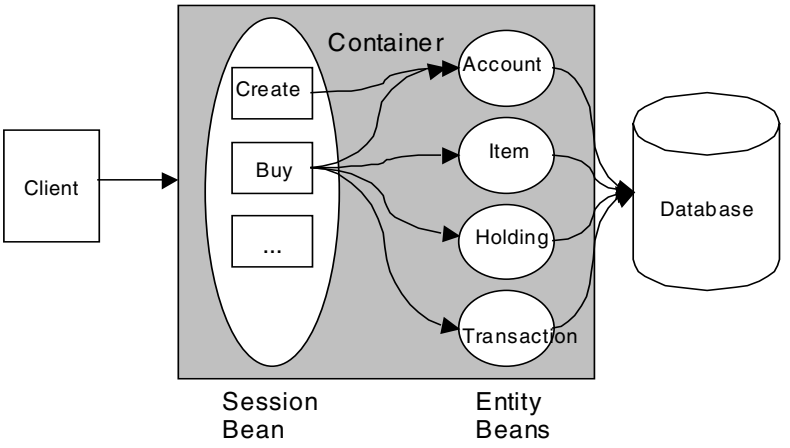


Fig. 2. Stock-OnLine Entity Bean Architecture

3.7 Entity Bean Details

Four entity beans are used to map to the four database tables. In order to understand the potential impact of the different commit options, pool and cache sizes, the dynamic behaviour of each of the four entity beans over the course of a single 100 concurrent client run is as follows (and summarized in Table 3):

- **Account** (Maps onto the SubAccount table): Initially 3000 instances. Some fields modified during tests. 1 instance created per iteration, 10 created per client. By the end of a 100 client run the number has increased by 1000 to 4000 (33% increase). 8000 instances are used (7000 read and 1000 created) during a 100 client run.
- **Item** (Maps onto the StockItem table): Initially 3000 instances. No fields modified during tests (but could be, as application allows for stock price changes). None created during tests. 36,000 instances are used (read) during a 100 client run.
- **Holding** (Maps onto the StockHolding table): Initially 30,000 instances (10 per Account). 3 instances are created per iteration, 30 created per client. Sells are paired with Buys. That is, each Holding that is bought will also be sold in the same iteration. The Holding's amount is set to 0, but the Holding is not actually deleted. The number of Holdings therefore increases constantly during test runs (at 3 times the rate of Account instances). By end of a 100 client run the number has increased by 3000 to 33,000 (10% increase). Holdings has a compound key which is sparse (i.e. key is *sub_accno* and *stock_id*, which has 9M permutations, of which only 0.33% exist initially. By end of 100 client run still only 0.36% exist). 56,000 instances are used (53,000 read, 3000 created) during a 100 client run.
- **Transaction** (Maps onto the StockTransaction table): Initially there are 0 instances, but the number of instances grows constantly. Not modified, or used during tests. 6 instances are created per iteration, 60 created per client (6 for buy/sell transactions). A 100 client run creates 6000 instances.

Table 3. Objects created and used in 100 client run

Entity Bean	Initial number of instances	New instances after 100 client run	Total instances after 100 client run	Number of instances used in 100 client run
Account	3000	1000	4000	8000
Item	3000	0	3000	36000
Holding	3000*10 = 30,000	3000	33,000	56000
Transaction	0	6000	6000	0

There is therefore one bean that grows rapidly and is heavily used during the tests (Account), one bean that is heavily used but (currently) is never created or modified (Item), one bean that has a sparse key, has 30k instances to start with and which grows and is heavily used during the tests (Holding), and one bean that is created but never used.

It makes sense to fix Transaction to be option C for these tests (otherwise it would need a large cache, but as it's never used a cache is wasted on it) and determine the impact of commit options, pool and cache sizes on the other beans.

3.8 Test Set-Up

The detailed experiments were carried out using Borland Application Server version 4.5 (BAS45), and Oracle 8.0.5, all running on NT 4.0. Three machines were used, 1 each for database, server, and client (Dual processor, 800MHz or faster, 1GB Ram; Client on an Ultra 80), with 100Mbit LAN connecting them. CPU usage on the middle-tier machine with BAS running on it was typically 90%, and CPU usage on the database machine was typically 30% or less.

Performance is measured in terms of Transactions Per Second (TPS). This is measured by dividing the total time taken by the client process (Wall-clock time) by the total number of transactions processed. For some experiments we also report the average client response times for each transaction type.

To ensure consistent and optimal results we limited the concurrency in the BAS ORB to 10 threads maximum. In BAS this limits the number of concurrent transactions, and the number of Session beans, to 10. Note that even though we repeated experiments to check results for consistency, there is still a small amount of experimental error in the TPS reported, up to about 5%.

4 Experiments

BAS calls the Pooled state pool the *Pool*, and the Ready state pool the *Cache*. Objects that have been moved out of the Pool to the "does not exist" state are called *unreferenced*, and when garbage collected they are *finalized*. We will follow this terminology.

For BAS we initially assumed that commit option B would be faster than commit option C, and used the default settings (1000 cache and 1000 pool sizes), producing reasonable results. However, we also tried option C, and surprisingly got slightly better results. We then tried to improve the performance of option B but initially found it difficult to achieve performance comparable to or better than option C. A more systematic testing approach was required.

In order to explore the impact of varying *Pool* and *Cache* sizes, we set them to different sizes determined by the total number of objects existing at the end of a run of 100 clients. A size big enough to accommodate all the starting objects and all the newly created objects corresponds to a cache hit-rate of 100% (200% is twice this). Table 4 enumerates the sizes for 10, 20, 50 and 100 percent hit-rates.

To make reporting and analysis easier we kept the cache/pool hit-rates the same for all the Entity Beans (e.g all 10%, all 20%, etc).

In theory the optimal Entity bean pool size depends *only* on the number of concurrent transactions[3] which is 10 due to limiting ORB threads. Therefore a maximum of 10 Account objects, 10 Item objects, and 200 (10 * 20) Holding objects

[3] In practice this is only the case for Commit Option C in BAS.

in the Pools are required. However, for the sake of completeness (and to test the theory) we varied the pool size across the same range as the cache.

Table 4. Example cache hit-rate sizes

Entity Bean	Total instances after 100 client run	10% cache size	20% cache size	50% cache size	100% cache size
Account	4,000	400	800	2,000	4,000
Item	3,000	300	600	1,500	3,000
Holding	33,000	3,300	6,600	16,500	33,000
Transaction	7,000	N/A	N/A	N/A	N/A

The number of objects pooled during a 100 client run is calculated from the number of instances of each object type used and the cache hit-rate. No objects are pooled with 100% cache hit-rate, and all objects are pooled with no cache (Figure 3).

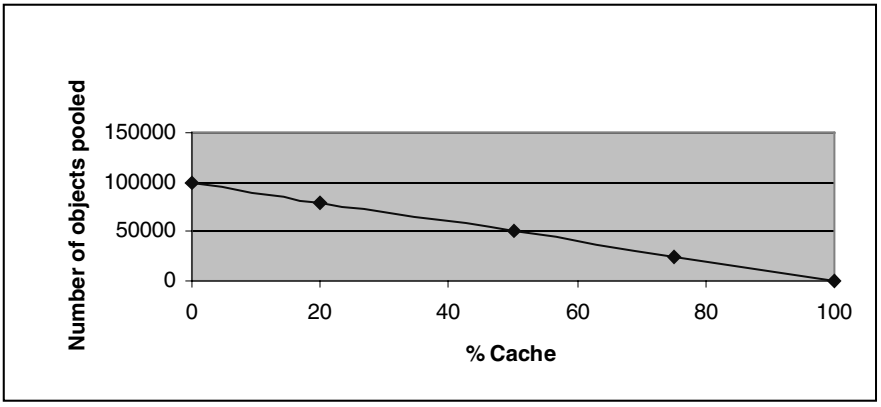


Fig. 3. Pooled objects: Expected total objects pooled during 100 client run

Assuming that object pooling has some cost associated with it we expect to see an increase in performance with increasing cache size.

For each configuration (Commit option, pool and cache sizes) we carry out the following routine:

- Restart the container/server (this clears the pool and cache)
- Load the jar file with the configuration to be tested
- Initialise the database
- Warmup the cache (using readonly operations on Account, Item, and Holding objects)
- Run 100 client test
- Record the TPS

Four experiments with BAS will be described: Experiment 1: Option C and varying Pool size (cache set to 0). Experiment 2: Option B and varying Pool size (cache set to 0). Experiment 3: Option B and varying Cache size (pool set to 20%). Experiment 4: Option A and varying Cache size (pool set to 20%).

4.1 Experiment 1: Option C and Pool

We assume that: Option C doesn't use the ready cache, so cache is set to 0. Optimal/maximum pool sizes for 10 threads are 10 Accounts, 10 Items, and 200 Holdings. We hypothesize that: 0 Pool size will be slowest. The differences in performance will be small, probably less than 10% (i.e. vendors such as SilverStream are largely correct in asserting that pooling makes no significant difference). TPS will increase up to "optimal" pool size, and then show no further increase.

Figure 4 shows the throughput (in TPS) for the different pool sizes, and Figure 5 shows the average client response times.

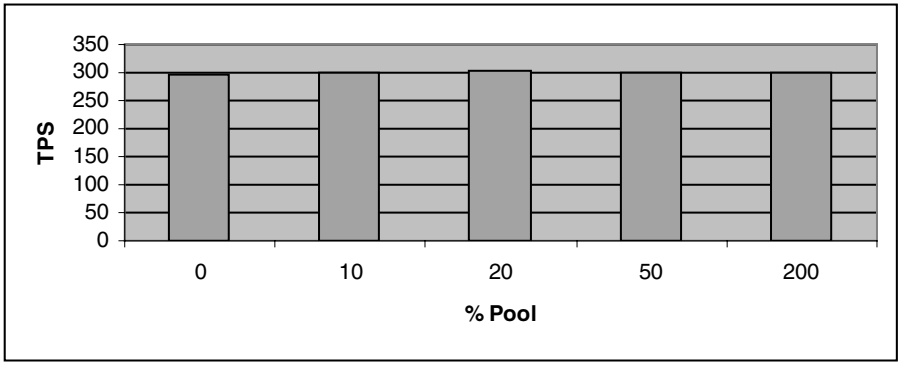


Fig. 4. Option C, Increasing Pool

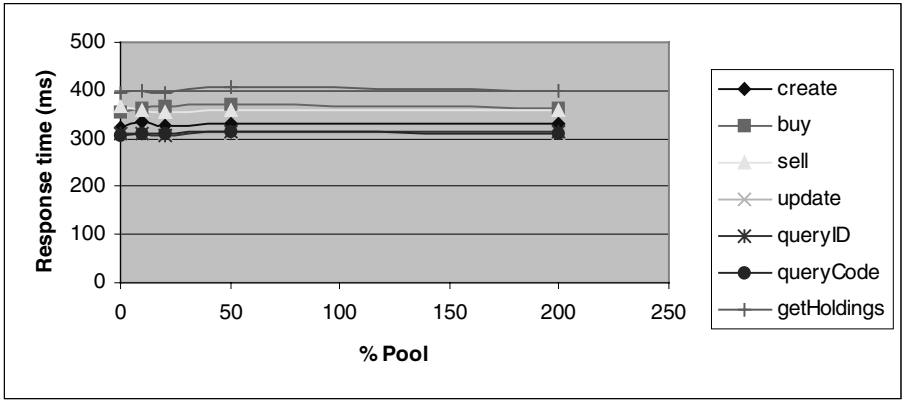


Fig. 5. Option C Client response times

Given that there is up to 5% experimental error these results do not show any significant difference in TPS for varying pool size.

With Option C we observe[4] that the total number of *ready* instances changes during the run but at any one time never exceeds the theoretical maximum based on the number of concurrent transactions (10) and objects used per transaction: i.e. 10 Account, 10 Item, and 200 Holdings.

The maximum number of *pooled* instances observed during the runs, for pool sizes greater than 0, was as expected (10/10/200) and did not depend on the actual pool size setting.

With 0 pool size the total number of unreferenced objects (in the "does not exist" state) at the end of the run was 2200 (no ready, pooled or finalised objects).

Option C with pool greater than 0 produced no unreferenced or finalised objects. That is, the objects are all being moved backwards and forwards to and from the pooled and ready states with no objects being destroyed. This is the "steady" state for the container and is an efficient use of resources: demand and supply of objects is the same, and all the objects are recycled.

Borland confirm that the pool is only resized every 5 seconds [11]. After 5 seconds any inactive beans will be discarded. In the steady state the same number of beans are being used, and they are always active. For pool sizes greater than 0 there will therefore be no (or very few) pooled objects, and less than the pool size, and therefore none need to be discarded. However, for a pool size of 0 the few inactive objects that are pooled will be be discarded, which is the observed behaviour.

The typical Option C performance is around 300TPS.

With 0 cache/pool settings, the server memory usage was only 56MB and constant. This implies that EJB applications using option C can be run on servers with relatively small amounts of memory.

Client side response times are consistent, and in order of slowest to fastest are: getHoldings, buy, sell, create, queries/update.

4.1.1 Conclusions

For BAS, using option C, a pool size greater than 0 has no significant impact on the throughput, and is probably not required. However, we observed that unreferenced objects are created, and will eventually have to be garbage collected, incurring some extra overhead. A small pool (the theoretical optimal/maximum size) is a sensible choice.

Given that 0 pool size is no slower than larger pool sizes, object creation and destruction must have minimal overhead compared to database access.

4.2 Experiment 2: Option B and Pool

This experiment is equivalent to Experiment 1, but using Option B instead. We hypothesize that: As option B with 0 cache is equivalent to Option C the results should be comparable; and different pool size will have minimal impact.

[4] BAS has good facilities for monitoring the number of instances in each state.

The results from this experiment (option B) compared with the previous experiment (option C) are show in Figure 6.

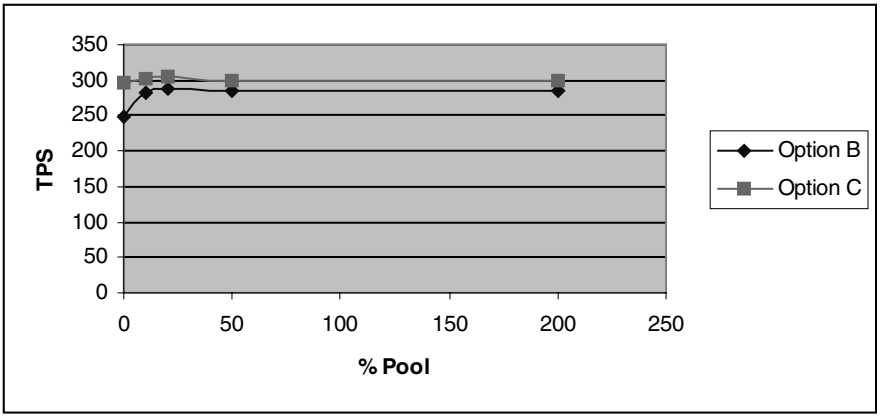


Fig. 6. Options B and C, 0 Cache, increasing Pool

0 pool size is slowest. The TPS increases by 40 from 0% to 20% pool size (16% increase), and flattens out at around 286TPS.

The best option B performance is 287 TPS, which is only 7 TPS less than the best Option C performance (304TPS). This is comparable, within experimental error. However, given that all the points from 20% cache upwards are the same, and none of them exceed the option C TPS, the indication is that the option B is really slower than option C.

Best performance with 20% pool size and above is a surprise given the theoretical expectation that the maximum pool sizes need to be only 10/10/200 (Account/Item/Holding), and the observation that only this many instances are in fact pooled using option C. We set the pool sizes to these theoretical sizes, and confirmed that this gave a slower result of 254 TPS (cf 287TPS for a larger pool).

With option C the number of ready objects never exceeded the maximum expected (10/10/200), while with option B the peak number of ready objects exceeded the maximum and fluctuated widely. This makes the "0" cache results difficult to interpret, but does seem to have an impact on the number of unreferenced objects as follows.

Even with 0 cache and 0 pool there are some ready and pooled objects at the end of the run (Table 5). The total number of unreferenced and finalised objects is in fact 100k, the expected number of pooled objects for a run. This is a lot more than the 2,200 unreferenced/finalised objects observed for Option C (0 cache/0pool).

Table 5. Option B (0 cache, 0 pool) instances at end of run

Entity bean	Ready	Pooled	Unreferenced	Finalised
Account	81	1	7500	260
Item	250	1	30000	390
Holding	700	1000	70000	700

However, with 0 cache and 20% pool the number of unreferenced/finalised objects drops significantly to about 10,000 (all of them Items). When there is no pool, discarding the objects used in a transaction incurs some overhead.

Borland indicate that locking is used to track the cache [11]. This is an overhead even with a (useless) "0" cache size and could be expected to give worse performance than for option C.

4.2.1 Conclusions

With a 0 cache size set, having at least a small pool has a bigger impact on option B than option C.

Option B (with no cache) performance is slightly slower than Option C. However, the difference is not large and we conclude they are comparable.

Even with "0" cache size set there are many ready objects during a run (more than the expected theoretical maximum as seen with option C)[5]. This results in many unreferenced objects being produced if there is no pool. Increasing the pool size reduces the number of unreferenced objects produced and correspondingly increases the TPS. We conclude that the pool needs to be large enough to cope with the peak demand for objects to and from the ready cache.

We will use a 20% pool size for the remaining experiments as a simplification to reduce the number of variables to one only - the cache size. We assume that the result for 0 cache also applies approximately to non-0 cache, but without making any explicit assumptions that this is optimal[6].

4.3 Experiment 3: Option B, Cache and Pool

We hypothesize that: Increasing the cache will increase the TPS until a maximum of 100% hit-rate; Because there is no locality of reference, random beans are accessed. Thus the working set of beans is just the number of rows in the database, and for optimal performance the cache will need to be close to 100%; For some cache size we will get better results than Option C (300TPS); The best TPS for Option B will be between 10 and 30% higher than Option C.

For this test we set the pool size to 20%. See Figure 7 for the throughput results.

Figure 8 shows average client side response times for each transaction type (3 representative points only).

During the test runs we recorded the number of instances pooled. The behaviour of the BAS container is suprising in this regard as it pools a larger number of objects than expected from the option C tests, and creates more objects nearer the middle of the % cache range (See Figure 9). The number of instances pooled is related to the peak ready pool size during the test runs. In fact BAS allows the actual ready pool size to overshoot the "maximum" setting for short periods of time, and the excess objects are then pooled reducing the ready pool size back to the "maximum" again.

---

[5] This is to be expected as option B pools objects with identity, so the maximum number of ready objects is the number of instances of the object (or rows in the database).

[6] An experimental check confirmed no significant difference in TPS for varying pool sizes.

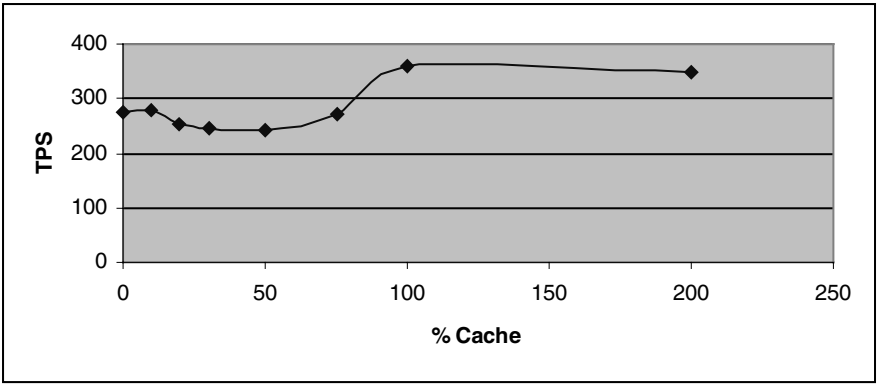


Fig. 7. Option B, Increasing cache

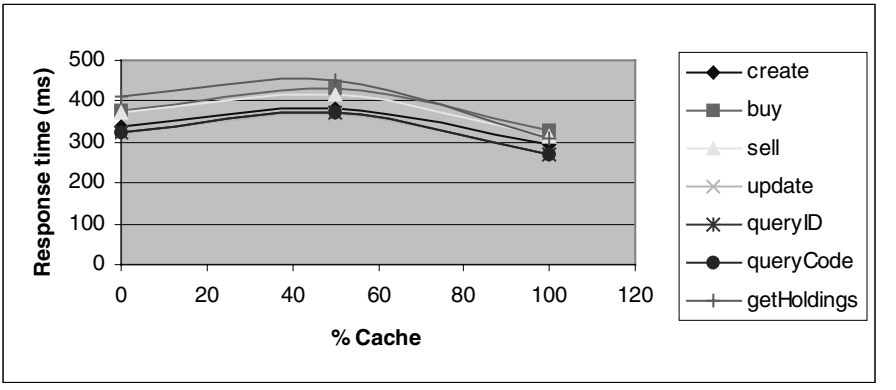


Fig. 8. Option B Client response times

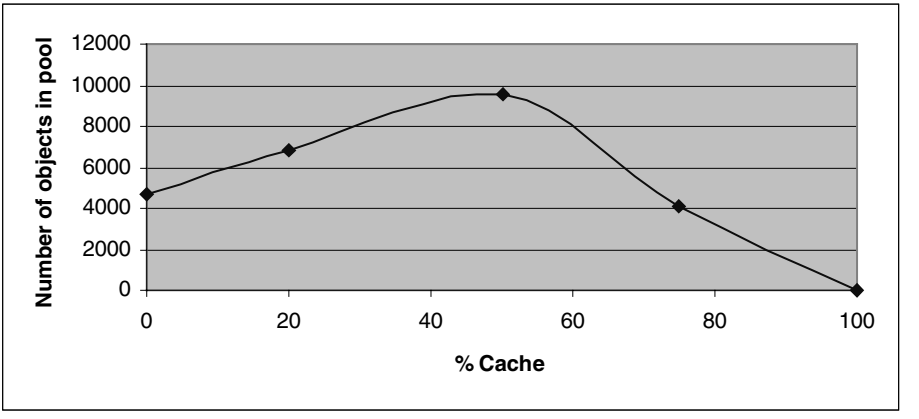


Fig. 9. Total pool sizes (account + item + holding)

With anything less than about 80% cache we get less than the best Option C throughput of 300TPS.

0 and 10% cache sizes give 277TPS, but then the TPS drops with increasing cache size up to 50%, rising to a peak of 360TPS at 100% cache size.

TPS drops again slightly to 200% (350TPS), but this is within experimental error so is unlikely to be significant.

The client response times increase, and then decrease, with getHoldings speeding up by 100% cache (to become faster than buy). Increasing the cache size has greater impact on getHoldings than the other transactions.

Initially we wondered if the number of instances in the pools is related to the shape of the TPS graph. It seemed suspicious that the peak Pool size and the minimum TPS both happen at around 50% cache size. However, we conducted further experiments with a 0 pool size which disproved this (no objects pooled during or at end of run, and same shaped TPS graph).

4.3.1 Conclusions

Option B with greater than about 80% cache is faster than Option C. 100% cache gives 360TPS which is 20% faster than option C. Option B with smaller cache (less than 80%) is slower than option C. Option B with 0 cache is close to option C performance.

The dip in TPS around 50% cache is initially suprising. However, we know that there are two competing factors, a benefit and a cost, involved in caching objects with identity:

- Benefit: More ready instances are available with increasing cache size.
- Cost: There are overheads with increasing cache size[7] (including garbage collection, retrieval of objects from the cache, managing the cache size, and pooling objects when the "maximum" size is exceeded).

It is possible to model the TPS graph by assigning suitable relative weights to these two factors, and subtracting the cost from the benefit. If this model is correct then the relative performance is just some function of the cache size benefit and cost.

Borland have given us extra insight into the impact of the cache locking overhead [11]. The smaller the cache sizes, the more locking is done as the background cache resizing thread needs to do more work to reduce the cache sizes. This provides the final key to understanding the shape of the TPS graph. The cache cost isn't linear, but consists of a locking component which *decreases* with increasing cache size, and a component which increases with increasing cache size (due to garbage collection, etc).

Because the performance is only better than option C with a large (>80% cache), for most real applications this will be difficult to achieve, particularly if there are a large number of objects, or if the number of objects grows.

Given that a significant performance improvement is only observed with a large cache, and that a large cache is practically impossible in most real applications, option C is a better option for consistently good performance and low memory usage.

[7] From the BAS ejb-cmp newsgroup.

4.4 Experiment 4: Option A, Cache and Pool

We hypothesize that: Because Option A caches data as well as object state, we expect the best Option A TPS to be substantially faster than Options B and C (In excess of 30% faster than option C); Even a small (Say 10%) cache size will give better than option C performance.

Figure 10 shows Option A results with increasing Cache, and 20% pool.

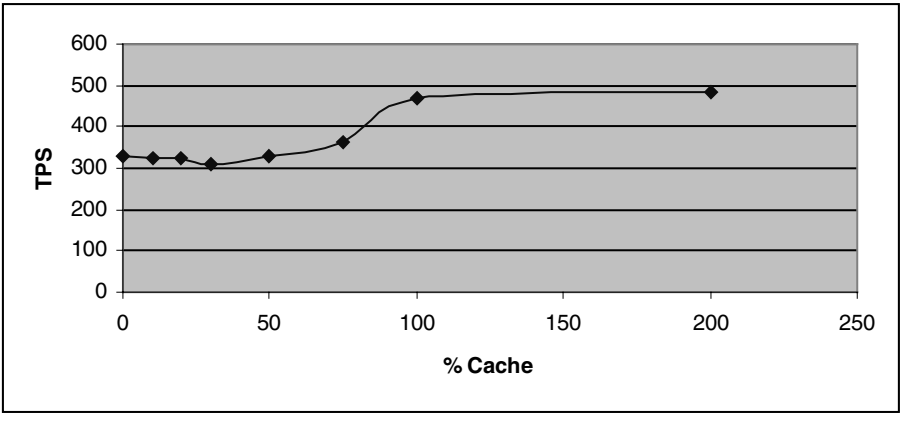


Fig. 10. Option A, Cache and Pool

We observe that: All Option A runs are faster than option C. This is because even with 0 cache specified, the container actually creates some ready instances (and therefore also caches some data). The TPS is fairly flat from 0 to 50% cache, and drops to almost the option C value at around 30% cache. From 50% cache the TPS increases, to almost the maximum at 100%.

A comparison of Options A, B, and C TPS is useful at this point to assist with comparative observations (Figure 11):

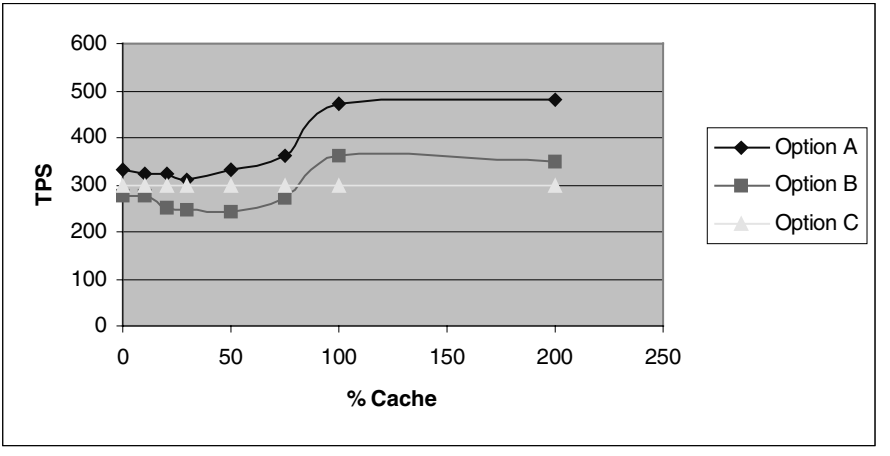


Fig. 11. Comparison of Options A, B and C

Option A has a peak TPS of 482 at 200% cache, and 471TPS at 100%. Using the 100% figure, Option A is 30% faster than the peak Option B (360TPS), and almost 60% faster than the peak option C (300TPS).

Option A and B have similar shaped graphs, although the drop in TPS between 0% and 50% isn't as bad for Option A as Option B. This is probably due to the increased efficiency of data caching almost compensating for the cost of caching in general.

In theory we would expect the Option A, B, and C 0% cache points to be the same, and the slowest, but they aren't. Previously we observed that for Option B 0% cache the container is actually creating ready instances. The maximum number of ready instances recorded for three runs with Options A, B and C (0 cache, 20% pool) are as follows (Table 6).

Table 6. Peak ready instances

Option	% Cache	Maximum Account Ready Instances	Maximum Item Ready Instances	Maximum Holding Ready Instances
C	0 (N/A)	10	10	200
B	0	240	800	1,300
A	0	260	2,700	14,000

For Option B, the ready pool size jumps around from 0 to and from the peak, and finally drops back to 0 at the end of the run. For Option A, the ready pool size increases rapidly to the peak and stays there, dropping back to 0 only at the end of the run. This proves that the cache size isn't a fixed maximum size, and that the actual number of ready instances depends on *both* the commit option and the cache size. If the actual cache size really was the fixed upper limit for the duration of the runs then the graph would look substantially different.

4.4.1 Conclusions

The benefits of the cache are more obvious for Option A than Option B. Until about 80% cache size Option B is still slower than Option C, and then only increases to a maximum just over the worst option A result. However, Option A at 75% is on par with the best Option B result, and improves even further.

5 Conclusions

The benefit of caching just object state and not data (option B) is minimal, with at least 80% cache size required to exceed option C TPS, and 100% cache size required for the maximum difference of 20% over option C. Achieving these hit-rates for cache sizes in real applications is difficult if the number of objects is large or increasing.

Option C gives consistently good results with no object or data caching. The advantage of object pooling for option C is also unobservable, and a pool size of 0 is probably adequate. Setting the pool to the maximum theoretical size (based on the maximum concurrency and the number of instances used per transaction) will

however prevent any overhead caused by unreferenced objects being garbage collected. Using Option C with no pool or a minimal pool size has significant implications for hardware resources, as a BAS45 server can be run in under 64MB RAM using option C, compared to anything up to 512MB for options A and B.

Option A gives slightly better performance up to 50% cache, increasing above 50% (for this application), with substantial benefits if everything can be cached. If the EJB application is the only application accessing the database tables, then it may be worth considering using option A. The throughput is almost doubled, and is roughly equivalent to adding an extra clustered server, but without the benefits of further scalability or failover.

Finally, we stress that these results, explanations and conclusions are product, version[8], and application specific. This in itself is a major lesson. Optimising CMP Entity bean code performance is a time consuming and complex task. Using the vendor's default settings, or even using the theoretically best settings, will be unlikely to produce optimal performance for real applications. An understanding of which commit options, cache and pool sizes are supported by the vendor and what they actually do, how they relate to the EJB specification, controlled experimentation with different deployment settings, and careful observation of what the container is actually doing during the running of the application, are likely to lead to a significantly better outcome.

If CMP Entity beans are an important part of your EJB architecture then careful evaluation and choice of an EJB application server is required. One of the major areas of differentiation between Application Servers is the level of support for Entity beans, including the extent of implementation of the specification, CMP performance, tool support for Entity bean application assembly and deployment, monitoring of deployed beans, and extra features such as clustering of Entity beans. Not all EJB 1.1 compliant products are equal.

References

1. Enterprise JavaBeans Specification, v1.1. Sun Microsystems (1999), http://java.sun.com/products/ejb/docs.html
2. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R., and Barter, C.: Cache coherency and storage management in a persistent object system. In: Proceedings of the Fourth International Workshop on Persistent Object Systems. Martha's Vineyard, MA, USA (1990) 99-109
3. Brebner, P.: An Object-oriented multi-media file system for D-CART; SRS, Design and Performance analysis (Unpublished project documentation). Australian Broadcasting Corporation (ABC) TR&D, Ultimo, Sydney (1995)
4. Kordale, R., Ahamad, M., Devarakonda, M.: Object Caching in a CORBA Compliant System. Conference on Object-oriented Technologies, Toronto, Canada (1996)
5. Sandholm, T., Tai, S., Slama, D., Walshe, E.: Design of Object Caching in a CORBA OTM System. Conference on Advanced Information Systems Engineering (1999) 241-254

[8] Borland have changed some aspects of the caching and pooling implementation from BAS 4.5.0 to 4.5.1.

6. Bretl, B., Otis, A., San Soucie, M., Schuchardt, B., Venkatesh, R.: Persistent Java Objects in 3 tier Architectures. In: Atkinson. M., Jordan, M. (eds.): The Third Persistence and Java Workshop. Tiburon, California, September 1st to 3rd (1998)
7. Roman, E., Öberg, R.: The Technical Benefits of EJB and J2EE Technologies over COM+ and Windows DNA. The Middleware Company (1999)
8. Borland Application Server 4.5, http://www.borland.com/appserver
9. SilverStream Application Server, http://www.silverstream.com
10. Gorton, I.: Enterprise Transaction Processing Systems: Putting the CORBA OTS, Encina++ and OrbixOTM to Work. Addison-Wesley (2000)
11. Weedon, J.: Borland Software Corporation. Personal communication (Email), 21 May (2001)

A WAP-Based Session Layer Supporting Distributed Applications in Nomadic Environments

Timm Reinstorf[1], Rainer Ruggaber[1], Jochen Seitz[2], and Martina Zitterbart[1]

[1] Institute of Telematics,
University of Karlsruhe, Germany,
{reinstor, ruggaber, zit}@tm.uka.de
[2] Institute of Communications Technology and Measurement,
TU Ilmenau, Germany,
jochen.seitz@tu-ilmenau.de

Abstract. Nomadic computing imposes a set of serious problems and new requirements onto middleware platforms supporting distributed applications. Among these are the characteristics of wireless links like sudden and frequent disconnection, long roundtrip times, high bit error rates and small bandwidth. But there are also new requirements like handover support and the necessity to use different networks (bearers). All these problems and requirements lead to the demand for an association between client and server that is independent of a transport connection. In this paper, we present a session layer that provides such an association for the middleware platform CORBA based on the Wireless Application Protocol (WAP) that is especially designed for mobile and wireless devices. It turns out that the session protocol in WAP called WSP is not able to fulfill our requirements, thus, it was necessary to define our own session layer. The session layer provides explicit and implicit mechanisms to suspend and resume a session, a reconnection to the session after the bearer was lost or changed and a solution to the lost reply problem. Furthermore, it contains an interface to be used by session-aware applications to control the presented mechanisms themselves on a fine-grained level. This paper presents a detailed description of the session layer, its integration into CORBA, a mapping of GIOP messages onto WTP and selected implementation details.

1 Introduction

Terminal mobility and wireless communications are two of the most important factors to be considered in frameworks for distributed applications. The success of wireless communications (e.g., GSM or UMTS) increased the interest in distributed applications implementing the paradigm of nomadic computing. However, due to the physical characteristics of the wireless communication link special mechanisms are required to handle high error rates, packet loss or even broken connections. These mechanisms are best implemented in a special layer

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 56–76, 2001.

hiding the occurring problems transparently to the distributed applications. We, therefore, propose a general purpose session layer whose services may be used whenever appropriate (compared to the session layer defined in the ISO reference model [10]).

The need for such a session layer was identified in many frameworks for distributed applications, most notably CORBA. Thus, we take this architecture as an example to show how our session layer can be implemented in a given framework. In Sect. 2.1 and 2.2, we introduce CORBA and its components that are affected by the session layer. Ongoing work on wireless access and terminal mobility inside the OMG is presented in Sec. 2.3. For the wireless link, special communication protocols have evolved, as TCP is not well suited for this environment (see Sec. 2.4). A viable solution is the protocol family called WAP (Wireless Application Protocol) which itself has realized the notion of sessions and is described in Sec. 2.5.

Section 3 starts with the identification of the session layer requirments. We found that the session idea defined in WAP is not very appropriate for distributed applications as we show in Sec. 3.2. In the remainder of Sec. 3 we present our design of the session layer.

This layer was implemented using the CORBA implementation ORBacus and the Wireless Transaction Protocol WTP of the WAP suite. First results of this implementation are presented in Sec. 4. Related work is given in Sec. 5. We conclude this paper with a summary of the achieved goals in Sec. 6 and give an outlook on work remaining to be done.

2 Basics

2.1 CORBA

CORBA (Common Object Request Broker Architecture) is a popular middleware platform, facilitating the implementation of object-oriented location-independent distributed client/server-applications [15]. CORBA supports the interoperability between clients and servers written in different programming languages, running on different operating systems and connected by different network technologies, thus, making CORBA the ideal basis for developing applications in a heterogeneous environment.

The central component in CORBA is the ORB (Object Request Broker), which is responsible for transparently forwarding requests and replies to the appropriate entity, thus, allowing a client to invoke operations on a server without knowing anything about the location of the server or its implementation.

2.2 GIOP

CORBA-ORBs use the General Inter-ORB Protocol (GIOP) to inter-operate. GIOP can be mapped onto connection-oriented protocols that meet a set of assumptions. GIOP defines the transport protocol independent properties, e.g.

message formats. ORB-Interoperability issues that are transport protocol dependent are defined in the mapping of GIOP onto the specific transport protocol.

The GIOP specification consists of the Common Data Representation (CDR) that maps IDL data types onto a low-level representation for transmission, the GIOP message formats and a set of GIOP transport assumptions.

GIOP messages are exchanged between CORBA entities to invoke requests, locate object implementations and to manage communication channels. GIOP communication is not symmetric. Therefore, to describe GIOP messages it is necessary to define client and server roles. In the GIOP context, a client is an entity that opens a connection and may send *Request*, *LocateRequest* and *CancelRequest* messages. The server accepts connections and may send *Reply*, *LocateReply* and *CloseConnection* messages. Client and server are allowed to send *MessageError* messages.

Every GIOP message contains the address of the server object (Interoperable Object Reference, IOR). The address of a server object consists of a transport dependent part that contains its network address and an opaque transport independent part that identifies the object inside the server. If a server object can be reached via different addresses, e.g. the server is connected to different networks that use a different addressing scheme, multiple addresses can be added to the IOR.

The assumptions the GIOP definition makes about the transport behavior include that the transport protocol is connection-oriented because a connection defines the scope for identifying requests. The connection has to be reliable, which means that bytes are delivered in the same order they are sent, at most once, and that a positive acknowledgment of delivery is available. The transport provides notification of disorderly connection loss. A client may multiplex connections to multiple target objects onto one transport connection.

Transport-dependent specifications like the addressing of the host and the server-object are defined in the mapping.

The Internet Inter-ORB Protocol (IIOP) is a mapping from GIOP to TCP/IP and must be supported by every ORB implementation. Mappings to other transport protocols may be defined. Altogether GIOP defines a kind of *lingua franca* for different middleware implementations.

2.3 Wireless Access and Terminal Mobility in CORBA

The Object Management Group identified the need for supporting wireless access and terminal mobility in CORBA and issued a request for information (RFI) [14]. The most elaborated reply to this RFP is based on the experience gained in the EC/ACTS project DOLMEN that implemented a prototype of CORBA extensions to support terminal mobility.

The architectural framework described in this reply [6] identifies three different domains: the home domain, which keeps track of the current location of the mobile terminal, the visited domain to which the mobile terminal is connected at a given moment using so-called access bridges, and the terminal domain that

is made up by the ORB on the mobile terminal and the terminal bridge that is responsible for connecting to an access bridge.

For communication between terminal and access bridge, a special protocol, the GIOP Tunneling Protocol GTP was defined. This protocol controls the tunneling of GIOP packets between the bridges and provides mechanisms for handoff and connection recovery. Being an abstract protocol, GTP needs to be mapped onto one or more concrete protocols.

The GTP-tunnel spans the unreliable communication link between the mobile terminal and the access node. Therefore, this tunnel must be established before the first method invocation, maintained and re-established to another access node whenever the mobile terminal moves from one area to another. Hence, GTP defines operations for tunnel establishment, tunnel handoff and tunnel release. However, using a connectionless transport protocol like UDP or WDP (as defined in the WAP protocol stack, see Sec. 2.5) makes it hard to determine when a connection is lost or when a handoff should take place.

2.4 TCP-Drawbacks

It is a well-known fact since many years that TCP does not provide a good performance in wireless environments [2,5]. The main reason is the error semantic of TCP and the resulting behavior. If a packet is lost during transmission, TCP assumes a congestion within the network and slows down transmission using the slow start mechanism. While this behavior makes sense in fixed networks, it is in most cases wrong in wireless and mobile networks. Wireless connections typically have much higher error rates including disruptions compared to fixed connections. Thus, packet loss is quite often due to these transmission errors instead of congestion. Furthermore, handover procedures in mobile networks can cause additional packet loss (packets in transit during the handover might be lost). Slowing down the performance is not at all useful in these cases, as the loss does not result from congestion. Several approaches have been proposed to solve this problem, either by splitting up the connection into a wireless and a wired part [1], by doing local retransmissions for lost packets [3], or by additional mechanisms controlling TCP directly [4]. However, all these approaches represent add-ons but not really integrated solutions. Furthermore, mobility increases the complexity of such approaches [7].

Furthermore, TCP does not support mobility of endsystems very well. This is due to the fact, that an IP address does not really identify an endsystem but a point of network attachment. Sockets as the premiere API for using TCP are based on IP addresses and, thus, suffer from the mentioned problems. Systems like Mobile IP [17] support endsystem mobility by integrating additional components and, thus, preserve the reachability of the mobile host with its original address.

2.5 WAP – The Wireless Application Protocol Suite

A viable solution providing Internet services for mobile, wireless devices has been worked out by the Wireless Application Protocol Forum (WAP Forum), which was founded in June 1997 by Ericsson, Motorola, Nokia, and Unwired Planet [23]. The basic objectives of the WAP Forum are to bring different Internet content (e.g., Web pages, push services) and other data services (e.g., stock quotes) to digital cellular phones and other wireless, mobile terminals (e.g., PDAs, laptops). Furthermore, a protocol suite should enable global wireless communication across different wireless network technologies, e.g., GSM, CDPD, UMTS etc. Therefore, the Forum embraces and extends existing standards and technologies of the Internet wherever possible and creates a framework for the description of content and development of applications that scale across a wide range of wireless bearer networks and wireless device types. Hence, the idea of using WAP for CORBA is evident (compare to Sec. 2.3).

Figure 1 gives an overview of the WAP architecture, its protocols and components. The management entities handle protocol initialization, configuration and error conditions (such as loss of connectivity due to the mobile station roaming out of coverage) that are not handled by the protocol itself.

The basis for transmission of data are different bearer services. WAP does not specify bearer services, but uses existing data services and will integrate further services. Examples are message services, such as SMS (short message service), circuit-switched data, such as HSCSD (high-speed circuit switched data), or packet switched data, such as GPRS (general packet radio service) in GSM. Many other bearer services are supported, such as CDPD, IS-136, PHS. No special interface has been specified between the bearer service and the transport layer with its wireless datagram protocol (WDP), because the adaptation of this protocol is bearer-specific. The transport layer offers a bearer independent, consistent datagram-oriented service to the higher layers of the WAP architecture. Communication is done transparently over one of the available bearer services. The transport layer service access point (T-SAP) is the common interface to be used by higher layers independently of the underlying network. If a bearer already offers IP service, then UDP is used as WDP and T-SAP is similar to the socket interface.

The security layer with its wireless transport layer security protocol WTLS is based on the transport layer security (TLS) and has been optimized for use in wireless networks. WTLS can offer data integrity, privacy, authentication, and denial-of-service protection.

The WAP transaction layer with its wireless transaction protocol (WTP) offers a lightweight transaction service. WTP distinguishes between an acknowledgment generated by the WTP entity of the responder and an user-generated acknowledgement, that is explicitly invoked by the receiving application. The service primitives of WTP are `TR-Invoke` to initiate a transaction, `TR-Result` to send back the result of a transaction and `TR-Abort` to abort an existing transaction. The PDUs exchanged between two WTP entities are the `invoke PDU`,

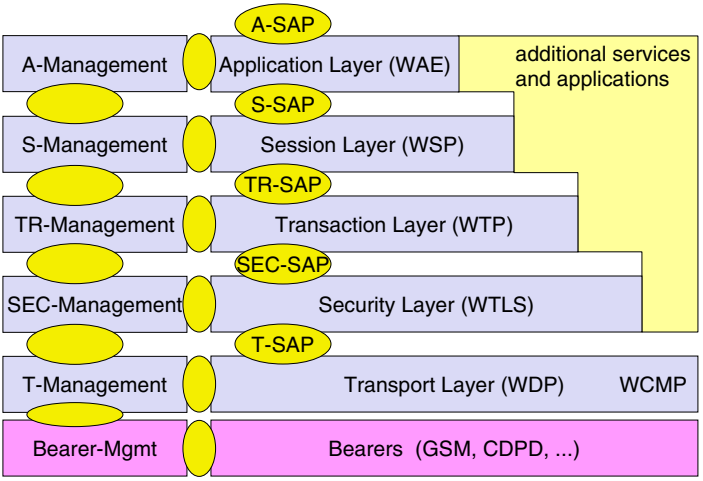


Fig. 1. WAP architecture

`ack PDU`, and `result PDU`. The basis of WTP are the three classes of transaction service:

Class 0 provides unreliable message transfer without a result message.

Class 1 provides reliable message transfer without a result message. For the sender the transaction ends with the reception of the acknowledgement, whereas the receiver keeps the transaction state for some time to be able to retransmit the acknowledgement if it receives the same invoke PDU again indicating a loss of the acknowledgement.

Class 2 provides reliable message transfer with exactly one reliable result message. It provides the classical request/response transaction, that may be directly used in client/server systems. In WTP class 2 transactions the result is acknowledged by the initiator of the transaction, resulting in at least 3 exchanged PDUs. An explicit acknowledgement of the `invoke PDU` puts the initiator on "hold on" to prevent a retransmission of the `invoke PDU` because the initiator might assume a packet loss if no result is sent back within a certain timeframe.

No WTP-class requires connection set-up or tear-down, which avoids unnecessary overhead on the wireless link. WTP does not include any flow or congestion control and thus avoids the main drawback of TCP in wireless environments.

The session layer with the wireless session protocol (WSP) currently offers a connection oriented and a connectionless service. It supports HTTP/1.1 functionality, long-lived session state, session suspend and resume, session migration and other features needed for wireless and mobile access to the web.

Finally, on top of it all, the application layer with the wireless application environment (WAE) offers a framework for the integration of different WWW and mobile telephony applications. The main issues here are scripting languages, special markup languages, interfaces to telephony applications, and many content formats adapted to the special requirements of small, handheld, wireless devices.

WAP does not force all applications to use the whole protocol architecture. Applications may use only a part of the architecture.

3 Session Layer

This section will introduce our design of a session layer using protocols from the WAP stack to support request/reply types of communication in general and as a specific application the remote method invocations in CORBA. The session layer is integrated into the mapping of GIOP onto WTP and is called Wireless Transaction Inter-ORB Protocol (WTIOP). When introducing a new layer into an already established framework like CORBA, there are two alternatives how the new layer could interact with the existing infrastructure:

session-aware: The layer could provide new functionality to applications which are aware of the characteristics of mobile communication. These are called session-aware applications in the remainder of the paper.

transparent support: The layer could be transparently integrated into the framework. This will support legacy applications which are not aware of the new session layer.

Our session layer will support both types of applications. In the following we will outline some requirements which have to be met by such a session layer.

3.1 Requirements

We identified the following requirements of the session layer in a wireless environment:

- The session entity has to provide data transport services for applications using a request/reply model for communication.
- The session entity has to be able to detect the loss of the transport connection.
- After a connection loss, the session entity needs to be able to transparently reconnect to the server.
- The session entity should support the change of the bearer used by the transport layer (vertical handover) and thus the change to another network initiated by the client.
- The session entity must provide a solution to the lost-reply problem, emerging if an invocation request was successfully sent to the server before a disconnection occurs. The computed replies have to be stored in the server and delivered to the client on its request [19].

- Session-aware applications should be able to explicitly suspend and resume the session e.g. in case of low battery power.
- A mechanism is needed to inform session-aware applications of events changing the state of the session. These events include session suspend and resume. Furthermore session-aware application should be able to influence e.g. the time interval after which the session entity should try to reconnect in the case of a connection loss.
- As a result of these requirements, the session entities have to provide an association between client and server, which is independent of the underlying transport connection.

Additionally, to support the use of the session layer for CORBA applications, these additional features need to be provided:

- Support of legacy CORBA applications, which are not aware of the session. In this case the session layer has to be inserted transparently into the CORBA environment (ORB).
- Ensure the "at-most-once" semantic of CORBA requests. A request, which is received by the session layer and is handed to the CORBA server, must be ignored if received a second time (e.g. over a different bearer after a disconnection period).

3.2 Integration

In the following we discuss the capabilities of the WAP layers and their use in a session layer fulfilling the requirements defined in the previous section:

WDP: On the transport layer, WAP defines the wireless datagram protocol that offers a bearer independent, consistent datagram-oriented transport service. When using IP based bearers WDP is realized by UDP. However, WDP is a connectionless and unreliable protocol and does not comply with the requirements of GIOP.

WTP: Therefore, there is the need for a service providing reliability to improve WDP. WAP incorporates the wireless transaction protocol WTP for reliable requests and asynchronous transactions. WTP is able to provide the necessary guarantees required by GIOP. Based on the different transaction classes, a fine grained selection of the required guarantees for a special request is possible. Furthermore, WTP is transaction oriented and thus simplifies the adaptation of CORBA requests to WTP invocations compared to stream-oriented transport protocols like TCP.

WSP: WSP is not well suited for our purposes of a generic session layer as it does not fulfill some of the basic requirements presented in Sec. 3.1. WAP is defined to support web-browsing on mobile devices and provides optimizations for HTTP. The core of the WSP design is a binary form of HTTP. It defines binary representations for information in the http header. As we do not use HTTP for sending request and reply messages these optimizations are useless for our purposes.

WSP provides session suspend and resume mechanisms, that provide disconnection transparency and enable the change of the bearer during session suspend. However, WSP does not support a reply polling on all requests that have been successfully sent to the server but the reply could not be returned to the client due to network disconnection. In this case it is necessary to resend the request, because replies are discarded after the disconnection is detected in the server and all active WTP transactions belonging to this session are aborted.

Hence, WDP and WTP of the WAP architecture provide a reliable transport service substituting TCP in the wireless domain and will be used in our session layer. WSP on the other hand provides special mechanisms supporting web-browsing on mobile devices but does not provide a really added value for generic distributed applications and will, therefore, not be used in our approach. Thus, we have to define our own session layer that fulfills the requirements pointed out in Sec. 3.1. Throughout this section we provide a detailed description of our session layer.

3.3 Architecture

Using WAP in a session layer raises the question how clients taking advantage of the session layer have to be integrated into an existing environment. Basically, there are two ways how this can be realised:

with gateway: Using a gateway corresponds to the WAP programming model. In this model, clients are not directly connected to a server but indirect via a so-called WAP gateway which is located at the crossover between the wireless and wired network. For use in our session layer this gateway has to fulfill some special functions. Apart from an implementation of the server functionality of WAP it has to support the forwarding of CORBA requests (Fig. 2). Thus, general WAP gateways can not be used in this context. A generic proxy platform for CORBA like π^2 [20] is able to fulfill these requirements. π^2 can be integrated into existing applications and provides a platform for value added services. The gateway solution enables the access of standard CORBA servers using IIOP, as the gateway is able to transform object references in a suitable way. On the other hand, if the system consists of more than one gateway it is necessary to provide a handover mechanism, which further complicates the development and deployment of such an architecture. Furthermore, gateway architectures add an additional point of failure and may lead to performance loss.

without gateway: In this architecture the client sends its request directly to the server without using a gateway. It is necessary that the server implements the WAP interfaces to accept requests sent via WTP. This end-to-end solution of supporting applications in nomadic environments is much simpler as no additional complexity (gateway) or handover mechanisms have to be realized. Using WAP this way creates the problem that WAP does

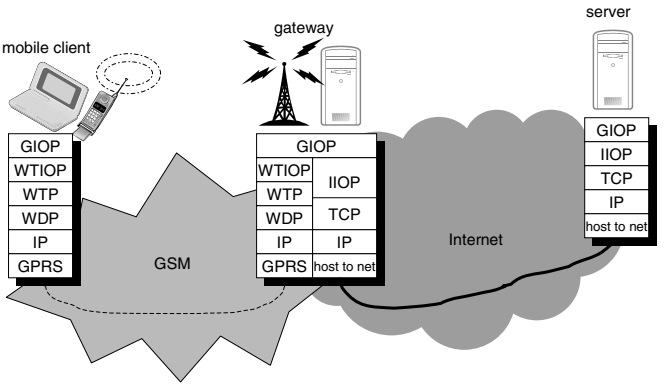


Fig. 2. Using a gateway to connect to a CORBA server via IIOP

not support any flow- or congestion control mechanisms. But because in our scenario mobile devices are connected via links with a small bandwidth, this is not a serious problem for existing networks. Routers in the wired network can handle much higher data rates and therefore can not become congested by packets received from the wireless link.

The session layer that is presented in this paper can be integrated into an architecture with or without a gateway. Using one or the other alternative is a trade-of as none is superior in all characteristics.

3.4 Concept

The purpose of the session layer is to provide a client-server context in which requests and corresponding replies can be transmitted. Multiple transactions can be invoked simultaneously in a single session. The important fact regarding the common problem of disconnections in wireless communication is the session independence of the underlying transport connection. The session layer may be suspended during periods of disconnection and resumed after regaining network connectivity which may use a different bearer. Requests that are already accepted by the session entity but have not been sent to the server and computed replies on the server that have not been sent to the client will not be discarded while a session is in a suspended state. In the following we will present how establishing, suspending and resuming a session works.

A session is identified by a session id, that is assigned by the server in the first acknowledge sent from the server to the client.

Session establishment. Depending whether the session layer is used as an end-to-end connection or via a gateway, there will be one session on the client

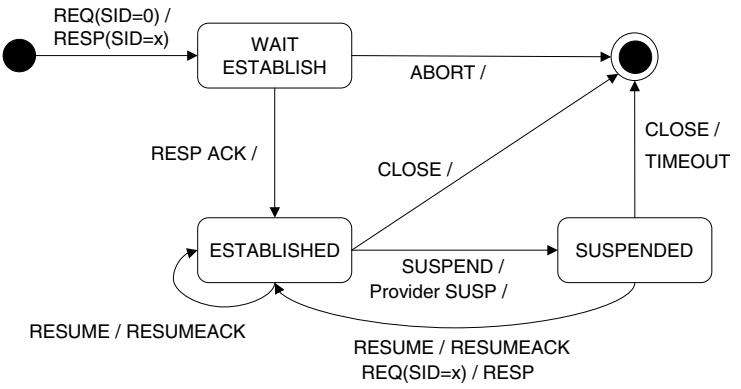


Fig. 3. Session state chart at server side

associated with every server the client uses or otherwise only one session between the client and the gateway. In the second case all GIOP connections from the client are tunneled over the single session between client and gateway. The session is always initiated by the client. To avoid the overhead of connection establishment (like three-way-handshake in TCP) the session is established implicitly during the first request from the client to a server. The server will create a new session on receipt of such a request and set the state of this session to `WAIT_ESTABLISH` (see Fig. 3). Together with the reply the server transmits a newly assigned session id back to the client, which has to be used in further transactions to identify the session. The client will set the state of the newly created session to `ESTABLISHED` after receiving the reply to the first request (including the session id). If the WTP acknowledgement to this reply is lost, the server will destroy the session, but the client will assume, that the session is still open. The client will not find out that the session is invalid before it sends a new request to the server which is answered with an WTP `TR-Abort(User)` by the server.

The tuple (session id, server name) identifies the session on client side. At the server the session id alone is unique. In the case of a transport disconnection during the first WTP transaction, the session establishment fails and has to be retried by the client (after regaining network connectivity).

Suspending a session. Figure 4 shows a state chart defining the session states at client side. At the client there are two possible events which cause a session to change into the `SUSPENDED` state. A session may be suspended either by the application (explicit suspend) or by the underlying transport layer (implicit suspend):

explicit suspend: A session-aware application may suspend a session explicitly by calling the `suspend` operation of the session. The session entity will send a

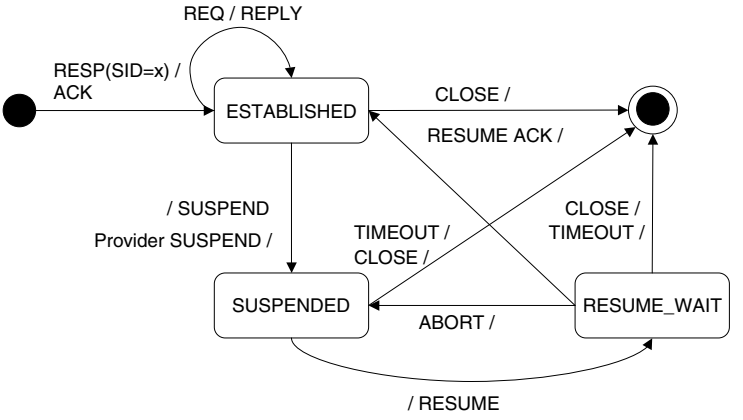


Fig. 4. Session state chart at client side

Suspend message to the server and change into `SUSPENDED` state. The session entity will continue accepting incoming replies but instead of handing them to the application (CORBA ORB), the reply will be stored internally. On the other hand requests from the ORB will be rejected during `SUSPENDED` mode. The session will remain suspended until explicitly resumed by the application.

implicit suspend: A session may be implicitly suspended in the case of detection of longer network disconnections (possibilities to detect disconnections are discussed further down). In this case the session entity will continue accepting requests from the ORB but store them internally instead of sending them. All synchronous method invocations by the ORB will remain blocked. Session-aware applications are notified by the session entity of the suspend event. The session-aware application may as a response to such a notification ask the session entity to stop accepting requests and, therefore, unblock any synchronous calls (returning an error). The client will try to resume the session by itself transparently to the application.

On the server side the session is suspended either by receiving a Suspend message from the client or also by the detection of network disconnection of the client. The server caches available but unsent or unacknowledged replies while the session is in `SUSPENDED` state (see Fig. 5). Once the session is reestablished by the client, the client can poll for missing replies. Thus, the retransmission of already sent requests is not necessary. The server may close suspended sessions after a specific amount of time to avoid keeping open "zombie" sessions which will not be resumed e.g. because the client has crashed. This timeout should be rather long to allow the session to survive long disconnections of the client.

Detection of network disconnection. Generally speaking, a disconnection is a state in which no WDP datagrams can be exchanged between the client and

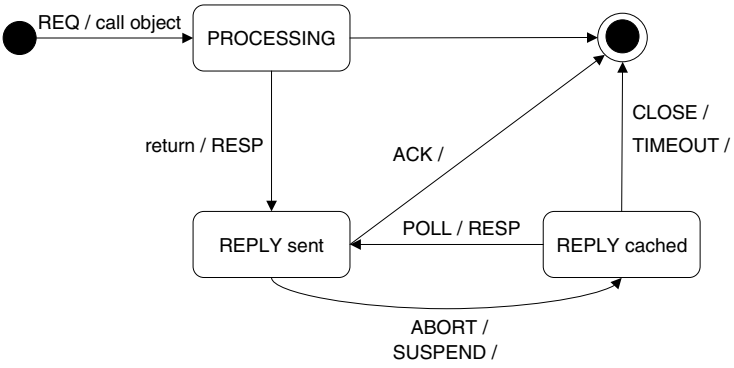


Fig. 5. Server side state chart of a request

server. We assume that these disconnections are caused by the loss of network connectivity of the client (e.g., because of loss of radio coverage).

The session entity on the server side may detect the unreachability of the client when trying to send results to the client via WTP transactions (class 1 or 2). The WTP entity will not get acknowledgements for these transactions and, therefore, abort the transaction and signal the Abort to the session layer. The session will change into `SUSPENDED` mode. If there are no outstanding results to be sent by the server, the disconnection will not be detected. Because of the passive role of the server this is not a problem. It is the responsibility of the client to resume a session, and the server will simply assume that there was a disconnection when it receives a Resume message.

On the client side the detection of network disconnection is critical. Unfortunately there are situations in which the session entity can not detect the disconnection directly by events from the WTP layer. That is the case when the WTP entity has sent a request (as an WTP invoke, see Sec. 3.5), received a "hold on" acknowledgement from the WTP server and then waits for the result. If during that period the network is disconnected, and no more requests are sent by the session entity, the WTP entity will not abort the transaction, because the disconnection is not detected. After a "hold on" message the client waits for an infinite time to receive a result. If on the other hand the network is disconnected before the WTP entity receives the acknowledgement, it will (after some retransmissions and timeouts, compare Fig. 6) abort the transaction. The session will be suspended by this Abort.

There are two ways to solve these problems:

1. A component outside the session layer may signal the network disconnection to the session entity. This component may be the operating system or a management entity of the WAP protocol stack (see Sec. 2.5).
2. A "hold on" message on the session layer may be introduced, which is sent from the server to the client in periods where no other messages (results or

acknowledgements) are sent. The time interval after which such messages are sent could be increased each time after sending a "hold on" message to reduce transmission costs in networks, where the amount of transferred data is charged (like in GPRS networks in Germany). The client would expect a message (result, acknowledgement, "hold on") after these time intervals. The missing of a message would cause the client to presume a disconnection and to suspend the session implicitly.

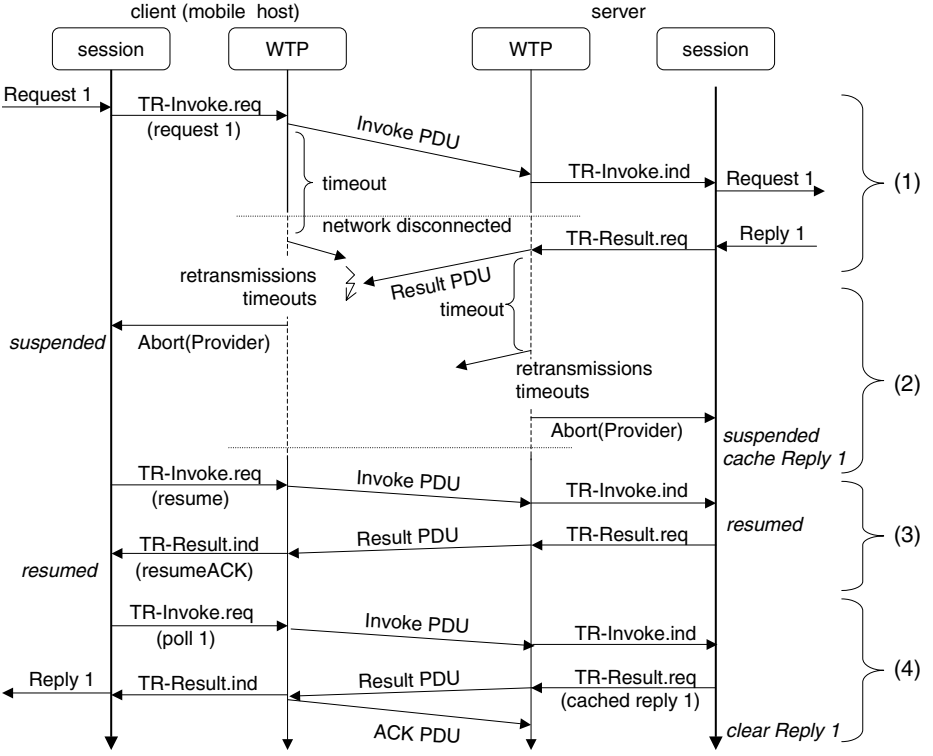


Fig. 6. Session suspend and resume

Resuming a session. The client tries to resume the session by building a new network connection and sending a Resume message to the server. Because the session is independent of a specific transport connection, the session may be resumed over a different bearer. The question which bearers are available and which have network connectivity is outside the scope of the session layer. Again the operating system or WAP management entities may provide such information or an approach as presented in [8] may be used to guard the availability of network devices as well as the network connectivity of these devices. The server

acknowledges the resuming of the session by returning a ResumeACK message to the client. Once the session state on the client is changed to `ESTABLISHED`, stored requests are sent and missing replies are polled from the server.

Figure 6 shows an example sequence of exchanged messages for suspending and resuming a session. At the beginning the session is already established (by request number 0, not shown in the diagram) and the next request shall be sent. The request is transmitted in a WTP `invoke PDU` to the server and the session layer hands the request to the CORBA ORB, which returns the reply (1). The client was disconnected from the network right after sending the WTP `invoke PDU`. The client tries to retransmit the WTP message, because it is not acknowledged by either a "hold on" acknowledgement or implicitly by a WTP `result PDU`. After several retransmissions the WTP entity aborts the WTP transaction by which the session at client side is implicitly suspended (2). The same occurs on the server which tries to retransmit the WTP `result PDU`, because it is not acknowledged by the client. The WTP entity also aborts the transaction, the session at server side is implicitly suspended, too. The server caches the reply. After regaining network connectivity, the client session entity sends a Resume message to the server, which responds with a ResumeACK message and changes back into the `ESTABLISHED` state (3). On receipt of the ResumeACK message, the client starts to poll for every missing reply. In this case, Reply 1 is missing, therefore, the client sends a PollReply message for Reply 1 to the server. The server responds with a CachedReply message containing the requested Reply 1. After the CachedReply message is acknowledged by the client, the server destroys the cached reply (4).

3.5 Mapping onto WTP

As stated above, we use WTP as the reliable transport protocol which is used by the session layer to send and receive requests and replies, respectively. WTP directly supports the request/reply communication model and is, therefore, more suited than byte-streaming protocols like TCP. The session layer maps the GIOP 1.1 messages directly onto WTP transactions as shown in Table 1.

Table 1. Mapping of GIOP messages onto WTP

GIOP msg	WTP primitive	trans.class	Initiator
Request	TR-Invoke	2	client
Reply	TR-Result	2	server
LocateRequest	TR-Invoke	2	client
LocateReply	TR-Result	2	server
CancelRequest	TR-Invoke	1	client
MessageError	TR-Invoke	1	client
MessageError	TR-Result	2	server
CloseConnection	TR-Invoke	1	server
Fragment	TR-Invoke	1	client/server

Whether transaction class 1 or 2 of WTP is used to send a GIOP *Request* (or *LocateRequest*) depends on the three different types of CORBA calls:

oneway: If the request is a oneway request (no response expected), always transaction class 1 is used (transaction class 0 could be used as well, because the CORBA standard does not demand the reliable transmission of oneway requests).

synchronous: If a response is expected, transaction class 2 is used for synchronous calls.

deferred: IIOP implementations usually use a synchronous invocation in the ORB to transmit these calls and, thus, transaction class 2 is used for deferred synchronous calls.

Besides the GIOP messages the session layer uses a few control messages to suspend, resume and close a session.

Table 2. Mapping of session control messages onto WTP

session msg	WTP primitive	trans.class	Initiator
Suspend	TR-Invoke	0	client
Resume	TR-Invoke	2	client
ResumeACK	TR-Result	2	server
CloseSession	TR-Invoke	1	client/server
PollReply	TR-Invoke	2	client
CachedReply	TR-Result	2	server

In the following we will discuss facts concerning the length in bytes of a GIOP message sent over WTP. The WTP standard defines an optional feature for segmentation and reassembly (SAR) of large messages (WTP invokes and results) which exceeds the MTU (maximum transfer unit: the maximum number of bytes, that can be sent in one packet) of the underlying bearer. The reason is, that many of the bearers used by WTP (like IP or GSM SMS) have a "lost one lost all" approach concerning SAR [9]. If the WTP implementation supports SAR, selective retransmission of lost segments is used to minimize the number of resent bytes. Because WTP supports no more than 256 segments in one invoke (or result) message, the size of an invoke sent by the session layer should not exceed $256 \times MTU$. Either the session layer can ensure this by introducing SAR at session layer or in our case with sending GIOP messages, the ORB can be instructed to divide large requests into fragments and send them as GIOP fragments.

4 Implementation Details

The implementation of the session layer is done in Java using ORBacus and Jannel. Both implementations had to be modified to work together. Where ORBacus

has to be modified to support a transaction oriented protocol like WTP, there was no publicly available Java implementation of the client side protocols of WAP. Most of this functionality had to be implemented by ourselves.

4.1 CORBA-Details

We use ORBacus by OOC as a CORBA implementation [16]. It is available with source code and is published under the ORBacus Royalty-Free Public License free of charge for non-commercial use. It includes the Open Communication Interface (OCI). This interface is below the GIOP layer where new mappings on transport protocols can be plugged in the ORB.

The OCI is designed for the use of bytestream-oriented transport protocols. As the session layer provides a message-oriented interface, we use a slightly adapted version of the OCI. This new interface preserves the structure of data sent and received by the ORB, which are GIOP messages. This is necessary to map the GIOP messages onto WTP-transactions. The identification of GIOP messages in form of request ids is also handed through this new interface to the session. We adapted the ORB to transparently use the new OCI interface with the session protocol and the old one with other protocols (like IIOP).

```
module wtiop {
  const unsigned long TAG_WTIOP_IOP = 4;
  const unsigned short IP = 1;

  struct Version {
      octet major;
      octet minor;
  };

  struct ProfileBody {
      Version wtiop_version;
      unsigned short bearerType;
      string host;
      sequence<octet> address;
      unsigned short port;
      sequence<octet> object_key;
  };
};
```

Fig. 7. IDL of WTIOP

In order to include a new PlugIn it is necessary to describe the address format in IDL. If the server supports multiple bearers we use multiple TaggedProfiles (one for each bearer) in a CORBA IOR (interoperable object reference). Right now only bearers providing IP are supported which may be used on a local

wireless LAN or wireless WAN such as GSM or GPRS. Figure 7 shows the IDL file describing the address format. To identify the WTIOP profile of an object we defined a tag with a value of four because it is not used otherwise. For official use it is necessary to get a tag assigned from the OMG. Every server needs a globally unique hostname (which may be a DNS name). This is used together with the session id to identify a session at client side. The port number refers to the WDP port number the server uses, and the address field contains the address in a bearer dependent encoding. This may be an IP address (or DNS name) for bearers providing IP, but could also contain e.g. a telephone number for other bearers. The `bearerType` field defines the bearer for which the address is valid.

The notification of session-aware applications is implemented by using the callback mechanism of the so called *Info* objects defined in the OCI. With these objects, applications can register callback methods which will be invoked when the particular event occurs. The standard OCI already defines callbacks for connection and disconnection events. These will be called by the session layer in case of session establishment and closing a session, respectively. We extended the *Info* objects with the possibility for applications to register additional callback methods which will be invoked in case of suspending and resuming the session.

4.2 WAP-Details

For our implementation of WTIOP we needed an implementation of the WAP protocols, specifically of the WDP and WTP protocol layers. Unfortunately we could not find a complete free open source implementation of these protocols. With Kannel ([11]) there is an open source project building an open source WAP gateway. Because Kannel's design is influenced by its use as a gateway, it is not well suited to reengineer a wap-stack implementation from its source code. Jannel is a Kannel port to Java by Empower Interactive, Inc. which better fulfilled our needs. Also because we use the Java version of ORBacus and wanted to implement the session layer in Java as well, the Java implementation was preferred. But the Jannel implementation only contained the server side implementation of the WAP protocols. Thus, we implemented the client part of WTP by ourselves (mainly added the client state tables of WTP). Beside that, we had to make some minor changes to the original Jannel implementation to enable the use of WTP with our own session layer instead of WSP.

5 Related Work

There are several other approaches which deal with the mobility of clients and their wireless attachment to existent networks e.g. by supporting continuous communication during periods of mobility.

The end-to-end approach to host mobility presented in [21] allows to migrate one end of an active TCP connection to a different IP address. This approach as well as our design assumes that normally it is the client that changes its network

attachment point (the IP address) and, therefore, no update of the location of the client has to be conducted, only existing connections must be migrated. If a server changes its location, the paper proposes to use a dynamic DNS update to reflect this change. However, there is no mechanism included to signal the event of connection-migration to the application which uses the TCP-connection and, furthermore, longer network disconnections cause an abort of the TCP-connection as in the original TCP (as a result of a TCP timeout). Another paper [22] by the same authors states that there is a need for notification of mobility-related events to applications and to hide longer periods of disconnection by introducing a session layer, but currently no implementation is available. Similar end-to-end approaches are used for migrating Java-Sockets [13] and TCP [18]. Split-connection proxies are used in MSOCKS [12] and OMIT [7]. However, all of them do not provide solutions to handle long disconnections or add a substantial overhead.

[8] introduces a system of dynamic network reconfiguration which monitors the availability of network interfaces and their status of physical connection. This enables applications to choose or change the network to be used for their communication based on availability, costs and provided services like throughput or latency. A similar system is needed by our session layer to be able to choose a bearer when resuming.

6 Conclusion and Future Work

This paper introduced a generic session layer based on WAP for distributed applications in the nomadic environment. This session layer relieves distributed application programmers from dealing with complex connection establishment and recovery in the wireless and mobile environment. Although this session layer is independent of any given architecture for distributed applications we showed its functionality and usefulness within a CORBA environment.

The session layer we conceived and implemented fulfills the requirements identified in Sec. 3.1. The implicit session establishment during the first request avoids the overhead of explicit session setup messages. Network disconnections are detected by the session layer even in cases in which the WTP protocol will not detect them. Both session-aware and legacy CORBA applications are fully supported. Session-aware applications may itself control the state of the session and must therefore be notified whenever an external session state modification has occurred. This is done via a special callback interface. But legacy applications will also benefit from the session layer, because network disconnections and reconnections are transparently handled. Without the session layer an error would occur in the case of a disconnection. Automatic reconnection and resuming of the session in these cases is provided. The lost reply problem introduced by disconnections or network handovers are solved transparently to the CORBA application by polling missing replies from the server after resuming the session.

Nevertheless, there still is some work to be done. First of all, the session layer shall also be evaluated through measurements on different systems. Furthermore,

session establishment and recovery should also consider different transport service providers (in addition to bearers providing IP). Hence, dynamic network reconfiguration should be possible.

Finally, the session layer will be an integral part of our π^2 architecture [20], a framework supporting nomadic computing.

References

[1] Ajay Bakre and B.R. Badrinah. I-TCP: Indirect TCP for Mobile Hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems ICDCS-15*, May 1995.

[2] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP Performance over Wireless Networks. In *Proceedings of the 1st ACM International Conference on Mobile Computing and Networking (MOBICOM'95)*, 1995.

[3] E. Brewer and R. Katz. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications*, 5(5):8–24, October 1998.

[4] K. Brown and S. Singh. M-TCP: TCP for mobile cellular networks. *ACM Computer Communications Review*, 27(5):19–43, October 1997.

[5] R. Caceres and L. Iftode. Improving the Performance of Reliable Transport Protocols in Mobile Computing Environments. *IEEE Journal on Selected Areas in Communication*, 13(5):850–857, June 1995.

[6] J. Currey, K. Jin, K. Raatikainen, S. Aslam-Mir, and J. Korhonen. Wireless access and terminal mobility in corba. OMG Document telecom/2001-02-01, Object Management Group OMG, February 2001. Revised Submission to RFP telecom/99-05-05.

[7] Andreas Fieger and Martina Zitterbart. Migration support for indirect transport protocols. In *Proceedings of the International Conference on Universal Personal Communications*, San Diego, California, October 1997.

[8] Jon Inouye, Jim Binkley, and Jonathon Walpole. Dynamic network support for mobile computers. In *Proceedings of the Third ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, Budapest, Hungary, September 1997.

[9] Inprise Corporation and Highlander Engineering Inc. Wireless access and terminal mobility. ftp://ftp.omg.org/pub/docs/telecom/2000-05-05.pdf, May 2000.

[10] Iso/iec is 7498: Information processing systems – open systems interconnection – basic reference model. International Standard, 15. Oktober 1984.

[11] Kannel: Open source WAP and SMS gateway. http://www.kannel.org/, 2001.

[12] D. Maltz and P. Bhagwat. Msocks: An architecture for transport layer mobility, 1998.

[13] Tadashi Okoshi, Masahiro Mochizuki, Yoshito Tobe, and Hideyuki Tokuda. Mobilesocket: Session layer continuous operation support for java applications. Technical report, Graduate School of Media and Governance, Keio University, October 1999.

[14] Object Management Group (OMG). Telecom Domain Task Force: Request for Information (RFI) - Supporting Wireless Access and Mobility in CORBA. ftp://ftp.omg.org/pub/docs/telecom/98-06-04.pdf, June 1998.

[15] Object Management Group (OMG). CORBA/IIOP Specification Version 2.3.1. ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf, October 1999.

[16] Object Oriented Concepts (OOC). http://www.ooc.com, 2001.

[17] Charles Perkins. *IP Mobility Support/IP Encapsulation within IP*, October 1996. RFC 2002+2003.

[18] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. Technical Report TR-CS-97-08, Canberra 0200 ACT, Australia, 1997.

[19] Rainer Ruggaber and Jochen Seitz. A transparent network handover for nomadic CORBA users. In *Proceedings of the 21st International Conference on Distributed Computing Systems ICDCS-21*, Phoenix, Arizona, USA, April 2001.

[20] Rainer Ruggaber, Jochen Seitz, and Michael Knapp. $I\!I^2$ - a Generic Proxy Platform for Wireless Access and Mobility in CORBA. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'2000)*, pages 191–198, Portland, Oregon, USA, July 2000.

[21] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th International Conference on Mobile Computing and Networking (MobiCom)*, August 2000.

[22] Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek. Reconsidering internet mobility. In *Proc. 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[23] Wireless Application Protocol Forum (WAP-Forum). http://www.wapforum.org/, 2000.

Middleware for Reactive Components: An Integrated Use of Context, Roles, and Event Based Coordination

Andry Rakotonirainy[1], Jaga Indulska[2], Seng Wai Loke[3], and Arkady Zaslavsky[4]

[1] CRC for Distributed Systems Technology
Level 7, General Purpose, University of Queensland 4072, Australia
andry@dstc.edu.au
[2] School of Computer Science and Electrical Engineering
The University of Queensland 4072, Australia
jaga@csee.uq.edu.au
[3] School of Computer Science and Information Technology
RMIT University, GPO Box 2476V, Melbourne VIC 3001, Australia
swloke@cs.rmit.edu.au
[4] School of Computer Science and Software Engineering
Monash University, Caulfield VIC 3145, Australia
A.Zaslavsky@csse.monash.edu.au

Abstract. The proliferation of mobile devices and new software creates a need for computing environments that are able to react to environmental (context) changes. To date insufficient attention has been paid to the issues of defining an integrated component-based environment which is able to describe complex computational context and handle different types of adaptation for a variety of new and existing pervasive enterprise applications. In this paper a run-time environment for pervasive enterprise systems is proposed. The associated architecture uses a component based modelling paradigm, and is held together by an event-based mechanism which provides significant flexibility in dynamic system configuration and adaptation. The approach used to describe and manage context information captures descriptions of complex user, device and application context including enterprise roles and role policies. In addition, the coordination language used to coordinate components of the architecture that manage context, adaptation and policy provides the flexibility needed in pervasive computing applications supporting dynamic reconfiguration and a variety of communication paradigms. [1]

1 Introduction

Pervasive (ubiquitous) environments are characterised by an immense scale of heterogeneous and ubiquitous devices which utilise heterogeneous networks and

[1] The work reported in this paper has been funded in part by the Co-operative Research Centre Program through the Department of Industry, Science and Tourism of the Commonwealth Government of Australia

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 77–98, 2001.

computing environments that communities use across a variety of tasks and locations. The pervasive environment is an environment in which components, in spite of this heterogeneity, can react to environmental changes caused by mobility of users and/or devices by adapting their behaviour, and can be reconfigured (added or removed dynamically). Pervasive environments need to offer a variety of services to a diverse spectrum of entities. Also, services can appear or disappear, run-time environments can change, failures may increase, and user roles and objectives may evolve.

Adaptability and reactivity is the key to pervasive computing. Pervasive environments require the definition of a generic architecture or toolkit for building and executing reactive applications. Reactive behaviours are triggered by events; in many cases the trigger is not a single event, but a possibly complex composition of events. These behaviours exist in many domains and are very useful for e-business applications (stock market, sales alerts, etc.) There are already existing research and industry approaches which address many adaptation problems.

However, no integrated architecture that is open, flexible and evolvable yet exists. This situation makes it impossible to test, within a single architectural framework, concepts such as (i) rich context description, (ii) generic approach to decisions about adaptability methods, (iii) dynamic definition of communication paradigms, (iv) dynamic reconfiguration of the architectural components, (v) incorporation of enterprise aspects like roles of participants and policies governing roles.

Most existing pervasive architectures have been influenced by a specific type of application, or by the type of underlying technology, thereby resulting in a lack of generality (see Section 2). In order to avoid building a monolithic architecture we aim to provide generality, and to design an architecture that separates clearly the computation aspect from the coordination aspect. The computation aspect is in turn separated into dedicated services modelled as context, adaptation and policy managers. These aspects are often merged into a monolithic framework preventing flexibility and extensibility. Architecture Description Languages (ADL) have been proposed as modelling notations to support architecture-based development [19]. Their ability to scale to large systems and pervasive environments are yet to be proven [23]. To coordinate events of autonomous components without re-writing these components, we defined a rule and event based coordination script, which has a high-level declarative meaning in terms of overall structure of the architecture to specify cross-component interactions.

In this paper we describe the m3 architecture and its Run-Time Environment (m3-RTE). The basic concepts of the framework were presented in [14]. This paper presents extensions and detailed applications using the proposed architecture. The structure of this paper is as follows. Section 2 provides both brief characteristics of related work and their evaluation. Section 3 gives a high level description of m3-RTE. Section 4 describes the design requirements and modelling concepts used in the proposed architecture. Section 5 characterises the architecture and describes the functionality and interfaces of its main components. It also describes interactions between components. Section 6 discusses

several issues related to the implementation of the architecture (m3-RTE Run Time Environment) including coordination of events, descriptions of the current status of the architecture prototype and its main components. Section 7 shows examples of applications using the services of m3-RTE. Section 8 discusses the benefits of m3-RTE. Section 9 concludes the paper and mention future works.

2 Related Work

Sun[tm] Open Net Environment [26] (Sun ONE) provides the means for deriving context: information about a person's identity, role, and location, about security, privacy, and permissions - all the things that make services smart. In Sumatra [1], adaptation refers to resource awareness, ability to react and privilege to use resources. Sumatra provides resource-aware mobile code that places computation and data in response to changes in the environment. Rover [17] combines architecture and language to ensure reliable operations and is similar to Sumatra in the sense that it allows loading of an object in the client to reduce traffic. It also offers non-blocking RPC for disconnected clients. Bayou [8] takes a different adaptability approach by exposing potentially stale data to the application when the network bandwidth is poor or down. Bayou is similar to Coda [24] in that respect. Coda is a distributed file system that supports disconnected operation for mobile computing. Mobiware [3] and Odyssey [20] are more focused on adaptability measurement. Mobiware defines an adaptation technique based on a utility-fair curve. This micro-economic based method allows the system to dynamically adapt on the QoS (Quality of Service) variation of the data link layer network. Odyssey's approach to adaptation is best characterised as application-aware adaptation. The agility to adapt is defined as a key attribute to adaptive systems. It is based on control system theory where output measurements are observed from the input reference waveform variation. TSpaces [15] provides group communication services, database services, URL-based file transfer services, and event notification services. It is a networked middleware for ubiquitous computing. TSpaces can be succinctly described as a network communication buffer with database capabilities. JavaSpaces Technology [11] is similar to TSpaces and is a simple unified mechanism for dynamic communication, coordination, and sharing of objects between Java technology-based network resources like clients and servers. In a distributed application, JavaSpaces technology acts as a virtual space between providers and requesters of network resources or objects. PIMA, Aura, Limbo and Ensemble address issues about architecture as m3 does. Limbo [7] and Ensemble [22] are concerned with the mobile architecture and the programming model. Limbo offers an asynchronous programming model (tuple space) and defines an architecture for reporting and propagating QoS information that might be related to the system. Adaptability is achieved by filtering agents. Ensemble's architecture consists of a set of protocol stacks which are micro-protocol modules. Modules are stacked and re-stacked in a variety of ways to meet application demands. Adaptation is done transparently to the application. The lowest layer tries to adapt first. If it

can't, then it passes the notification to the layer above. This repeats until, eventually, the application itself has to reconfigure. PIMA [2] is a framework that focuses on providing a device-independent model for pervasive applications that facilitates application design and deployment. It provides functionalities such as an application adaptation enabler, and dynamic application apportioning to services. Aura [28] is a task-driven framework that helps the user to gather information about available services, selecting suitable services to carry out tasks and binding them together. A Service Coordination Protocol controls adaptability mechanisms required when the environment changes. Open-ORB [4] and dynamicTao [18] use reflection mechanisms to address dynamic configuration and adaptation of components. The two approaches make use of a reflective CORBA ORB that gives program access to meta information that allows the program to inspect, evaluate and alter the current configuration of components. In [10] is presented a set of requirements for future mobile middleware which support coordinated action between multiple adaptations triggered by multiple contexts.

As can be seen, the above work addresses specific types of adaptability for particular middleware and hence, caters for only specific kinds of contextual information. The cited works propose adaptation to the quality of service provided by the network or the Operating System and do not provide an open, evolvable architecture enabling the use of a broad range of concepts related to adaptation of enterprise applications. Furthermore, dynamic coordination of heterogeneous components, dynamic definition of communication paradigms and context awareness are not addressed in the existing solutions.

3 High Level Description of the m3 Architecture

In this section we first present a high level view of the m3-RTE (Run Time Environment). The m3-RTE can be considered as an ORB (OMG Object Request Broker) or an agent that adapts the delivery of information (protocol and content) with the use of context (user profile, server profile, etc), adaptation and policy services (see Figure 1). The m3-RTE has a programming interface that allows the system programmer to specify relevant contexts of interest, adaptation rules, policy rules and coordination within m3-RTE. Clients can access servers transparently. Servers are wrapped by m3-RTE so that all in/out-coming messages are filtered and coordinated by the m3-RTE coordination service.

Neither the user application nor the server do not need to know about the internal mechanism of m3-RTE: the adaptation of the request (protocol, content) to be delivered to a particular application can be done transparently. A Tickertape application is presented in Section 7 that demonstrates such an approach. Tickertape messages are pushed to applications based on subscriptions. m3-RTE was programmed to change dynamically the interaction protocol and the content of information delivery based on context (e.g. device capability, MIME content) and policy (e.g. level of security) without touching the client front end.

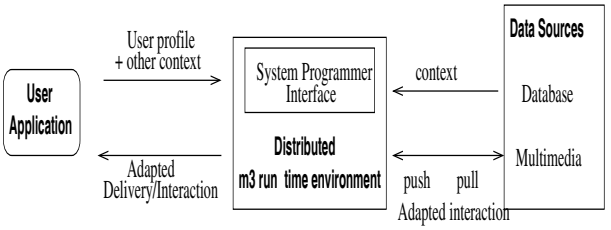


Fig. 1. m3-RTE located between the front end and back end

4 m3 Design Requirements and Modelling Concepts

In m3-RTE, we seek a balance in providing a high level model of a bounded pervasive environment (e.g., an enterprise) in terms of context, roles and events, and supporting dynamic lower level behaviours - such as reactivity and changes in configuration of components. More specifically, the design of the m3-RTE was driven by the following requirements:

1. Enterprise focus: As enterprise applications are complex and their complexity increases if they operate in pervasive environments. We need abstractions that capture the purpose, scope and policies of the system in terms of behaviour expected of different roles within the system by other entities (roles) within the enterprise. e.g the CEO is authorized to increase the salary of Project Leaders.
2. Reactivity/proactivity: reactivity is an important design guideline of the m3 architecture: the architecture should allow easy programming of reactions to context changes. The changing context of a component determines its interactions with other components and the service it can offer. Proactivity is the ability of an m3-RTE to foresee changes that might occur and prepare for those. This ability is crucial for successful adaptation and is achieved by generating sets of events to compensate for fluctuations in the system's behaviour and the surrounding environment. Patterns of behaviours and a history of previous executions are used for proactivity support.
3. Open dynamic composition: the system must support inter-operation with open components such as legacy applications or new components. New components may exhibit new types of interactions which go beyond the currently common but very static and asymmetric Remote Procedure Call (RPC). Dynamic composition (plug and play) configuration and customisation of services is essential in a pervasive environment as lack of resources to carry out a task may require a partial/complete reconfiguration of services.

Requirement (2) is addressed by using the event notification concept (publish/subscribe) as a core building block of the m3 architecture. Requirement (1) and (3) are addressed by integrating a policy manager inspired by RM-ODP enterprise policy specification and RM-ODP modelling concepts into the m3 architecture [16] . The three basic modelling concepts inspired by RM-ODP [16]

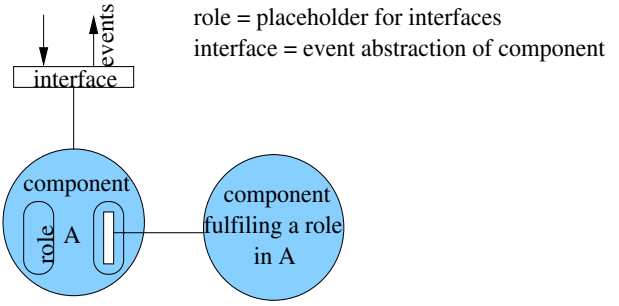


Fig. 2. m3 architecture modelling concepts

are components, their roles and the events to which components can react. These basic modelling concepts are illustrated in Figure 2.

- **Event** is an atomic modelling concept from which interaction protocols such as message passing, RPC, streams and other communication paradigms can be expressed. We assume that interactions between all components are event based.
- **Component** is a core modelling concept. It is a software abstraction. In object-oriented terminology, a component is a set of interacting objects (or even a single object).
- **Roles**: m3 focuses on RM-ODP enterprise specifications which define the purpose, scope and policies for a system in terms of roles played, activities undertaken, and policy statements about the system [16]. An enterprise role is an identifier for a behaviour (e.g. Director). A role is an abstraction and decomposition mechanism. It focuses on the position and responsibilities of a component within an overall system to carry out a task. An example of the Director's task is *supervise_subordinates*.
  A role cannot exist without a component. A role is a placeholder that can be fulfilled by one or more interfaces. Roles provide abstraction required for enterprise modelling.Roles can be added/removed dynamically during the lifetime of a component.

5 m3 Architecture

The modelling concepts in Section 4 are used to define the m3 architecture. The architecture is organised into three layers as shown in Figure 3:

- Coordination layer: a coordination manager that executes a coordination script written in MEADL (Mobile Enterprise Architecture Description Language)
- Dedicated manager layer consisting of three fundamental parts of pervasive and reactive computing environments. They are the Context, Adaptation and Policy managers.

– Distributed Service layer: services such as notification and security. This layer also includes network protocols.

The following subsections describe the functionalities of these three layers. It also gives examples of interaction within the m3 Run Time Environment (m3-RTE).

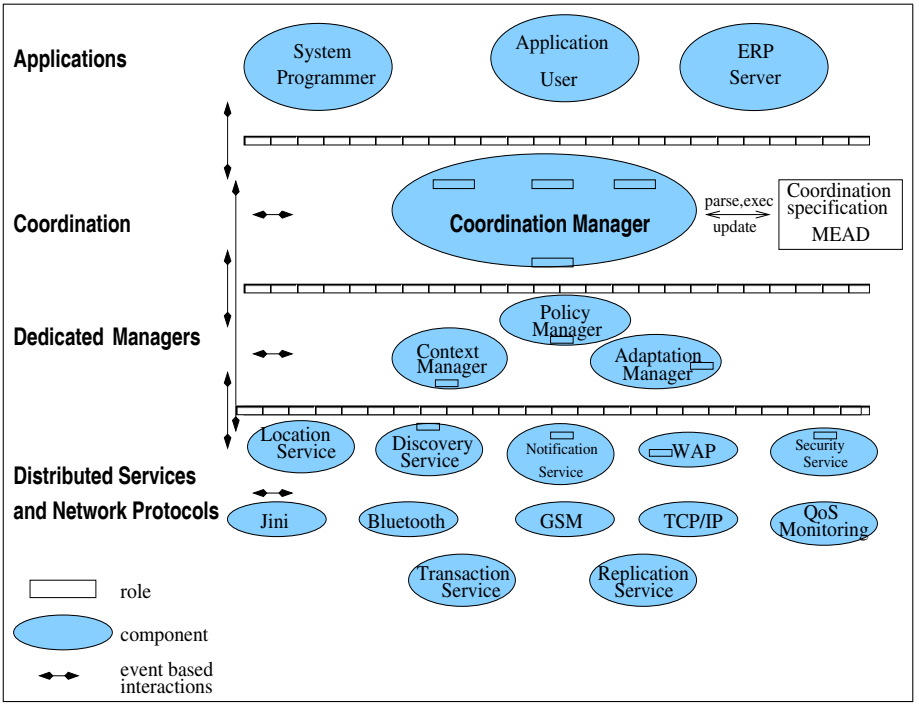


Fig. 3. Internal components of the m3 Architecture

5.1 Coordination Layer

The need for increased programmer productivity and rapid development for dynamic changes within complex systems are the main motivations that led to the development of coordination languages and models. Coordination-based concepts such as Architecture Description Languages (ADL) provide a clean separation between individual software components and their interaction within the overall software organisation. This separation makes large applications more tractable, more configurable, supports global analysis, and enhances reuse of software components.

The coordination layer is the core of the m3 architecture. The coordination focuses on concepts necessary to manage the implications of mobility and reactivity for enterprise-wide activities. For example the coordination layer should

be able to re-configure itself if a policy manager and/or context manager is unreachable or cannot be instantiated in a small device. The coordination consists of two parts (*i*) specifying with MEADL the interaction between enterprise roles and (*ii*) executing MEADL scripts within a coordination manager.

Coordination specification with MEADL. MEADL specifies the coordination of events. It is a coordination script that separates the behaviour specification (functionalities) of components from the communication needed to coordinate such behaviours. MEADL specifications provides means to configure dynamically the interaction between components, to change dynamically the communication paradigm and to provide session management.

We take a pragmatic (not formal description techniques) and minimalist (simple) approach to specify coordination. The script offers two main advantages: (*i*) interaction between components can be changed dynamically, and (*ii*) coordinated components can be added or removed dynamically.

Events belong to roles, and therefore, MEADL specifies the coordination of roles. Roles can be fulfilled by users, ERP(Enterprise Resource Planning) servers, network protocols or dedicated managers. A role's description is the interface of tasks that a component can provide. A role's duty is described in a set of input/output events. The duty of roles is to perform a set of tasks. The interface to tasks corresponds to the interface of components.

The m3 architecture is a reactive architecture and this is reflected in the use of the reaction rule syntax `on event` as first class construct of MEADL.

`on event` $R_j : E_i$ waits for the occurrence of a set of events E_i from a defined set of roles R_j. The value of E_i parameters is checked with a `WHERE` clause. The relevant context is also checked with `in context` or `out context` clauses.

Code 5.1 MEADL EBNF syntax

```
rule    : "on" pexpr [guard] "{" actions "}"
pexpr   : "event" role "(" [ident(","ident)*] | "(" pexpr ")"
          | pexpr op pexpr
guard   : "not"* ( "in" | "out" ) "context" string
actions : [action(";"action)*]
role    : ident ":" ident
op      : "where" | "and" | "or" | "eq" | "neq"
action  : ( "call" | "emit" ) ident "(" [expr ("," expr)*] ")"
```

The MEADL syntax is summarised in Code 5.1. An example of a MEADL sentence is shown in Code 5.2. Code 5.2 specifies a rule that waits for the occurrence of event `http:get(balance, account)` from the role `Director`. If the current context is secure and the balance is more than 10,000 then it emits a new event featuring the new balance then calls a function `foo`.

Code 5.2 Example of MEADL specification

```
ON EVENT Director:http:get(balance, account)
        WHERE (balance > 10,000)
        IN CONTEXT ‘‘secure_environment’’
    {
      EMIT employee_chat_line(‘‘DSTC has’’,balance);
      CALL foo(account, 23);
    }
```

Coordination Manager. The coordination manager is an engine that executes a MEADL script. MEADL scripts are transformed into XML [27]. Then, the coordination manager coordinates the events based on the generated XML specification. The use of XML is a provision for the future use of different coordination mechanisms (e.g. workflow engine) and language independent coordination.

The coordination manager coordinates the other components by sending/receiving events. To allow rules to affect other rules (a form of reflection) and to allow dynamic change of reaction rules, the Coordination Manager offers the following methods:

- `Reaction-Id create-react-rule (spec)` create a reaction rule
- `void delete-react-rule(reaction-Id)` delete a reaction rule
- `void map (event, operations)` : map an event to a set of operations
- `void inhibit (reaction-Id)`: inhibit a reaction rule
- `void reactivate(reaction-Id)`: reactivate an inhibited reaction rule

5.2 Dedicated Managers Layer

The middle layer of the architecture is composed of three dedicated managers which provide the three major functionalities. We identified them as necessary for modelling the enterprise pervasive environment (and thus, for building applications for such environments). The three managers are context, adaptation and policy managers. The Context Manager (CM) provides awareness of the environment and feeds the Adaptation Manager (AM) with context information in order to enable selection of an appropriate adaptation method. The Policy Manager (PM) ensures that the overall goals of the roles involved are maintained despite adaptation and that a change of roles is followed by appropriate adaptation.

The use of these managers is invisible outside the m3 architecture. They are autonomous components with well defined interfaces that interact together to provide the appropriate level of service to applications using the m3-RTE. Only the coordination manager has an impact on the way the three dedicated managers interact.

Context Manager (CM). Components may move to other environments or some features of the current context may change (memory, disk space, user preference etc). Context consists of

> "any information that can be used to characterise the situation of an entity, where an entity can be a person, place, physical, or computational object." [9]

The aim of the CM is to provide context awareness. The design of a context aware system must contain at least three separate but complementary steps. They are:

1. Context description: The syntactical representation of semantic-free information relevant to an entity.
2. Context sensing: Related to gathering contextual information specified in the context description.
3. Context interpretation: Related to understanding perceived information.

The current version of CM focuses on bullet point 3. The CM is responsible for gathering context information and making it available to the other entities. The CM enables consistent addition, deletion and modification of the context (value, attribute and relationship between attributes) and the detection of context changes.

We identified two types of context called local and community context. A component can be aware of its own *local context* (e.g. the characteristics of the device on which it is running, location, role fulfilled by the user), or the contexts of all the components or roles (on the same device or on other devices) it is interacting with. We call the combination of the contexts of all the interacting applications *community context*. A community being the composition of components formed to meet an objective [16]. Awareness of local or community context (or changes in local or community context) can itself be local (i.e., only one application has the awareness) which we call *local awareness*, or community (i.e., all the applications are aware of the context change at roughly the same time either synchronously or asynchronously) which we call *community awareness*.

The CM represents context as value/attribute pairs using RDF (Resource Description Framework) graphs. RDF instances assign values to instances. It is these instances that describe context. RDF enables the definition of schemata which describe the gathered context as well as allowing the use of relationship operators between attributes (e.g cardinality) or constraint properties. A community context is modelled as relationship between different contexts (e.g relationship between context of different roles). The use of RDF enables us to use the work of the W3C CC/PP [5] WG which is defining a vocabulary for describing device capabilities, as well as protocols to transmit these device profiles from the client. The interfaces provided by our context manager are:

- `ContextValue getContext (Id,Path,Callback)`: Get the context associated with Id
- `void addContextListener(Path, Callback)`: Register to be notified when context changes take place. Path restricts the scope of interest.
- `ContextAttr listContextAttr (Id,Path)`: Get the context vocabulary associated with context Id.

- `boolean ExistsContextAttr(Id,Path)`: Checks if the attribute pointed to by Path exist in context Id
- `boolean incontext(component, ctx)` Checks if a component is within a context
- `boolean outcontext(component, ctx)` Checks if a component is outside a context

Adaptation Manager (AM). The AM enables the m3-RTE to react to changes context. The AM makes a decision about adaptation if context changes and installs/invokes an appropriate adaptability method. The AM is able to incorporate various types of adaptation.

Adaptation in response to context changes can be either local (i.e. only one application adapts independently from the others) or community (i.e. all the applications adapt together in some coordinated manner). We call these two forms of adaptation *local adaptation* and *community adaptation* respectively. These six dimensions are summarized below.

	local	community
context	local context	community context
context-awareness	local awareness	community awareness
adaptation	local adaptation	community adaptation

Most work to date has focused on local adaptation of an application in response to local awareness of local context [10]. For instance, there is not much work on other aspects such as local adaptation in response to local awareness of community context (i.e., an application adapts according to its knowledge of the contexts of the other applications on other computers it is interacting with. Our approach to tackle community adaptation is to specify dependencies (chain) of execution between adaptation rules. At this stage our dependency is sequential (A then B) but we plan to have concurrent triggering of adaptation rules. A dependency is modelled as a graph where nodes are adaptation rules. The AM prevents the formation of cycles in the graph as it represents a ping-pong effect.

The AM specification is based on `Event Condition Action` (ECA) rules. `Event` is an event that might trigger `Action`. `Conditions` are conditions related to the event. Events are often generated by the context manager. `Action` is a reference to an adaptation action. Such rules support multimedia (continuous) and operational (discrete) adaptation and decouple the rules from the actual implementation of the adaptation. It also supports the specification of application aware adaptation [20] and application transparent adaptation [24,3] by respectively giving the control and implementation of the (A)ction of an ECA rule to the application or to the m3-RTE (see Section 7.1).

The Adaptation Manager provides the following interfaces:

- `Boolean AddECARule(ECARule r)`: Adds an adaptation rule. The rule r is added to the Adaptability Manager.
- `Boolean RemoveECARule(ECARuleID rid)`: Removes an adaptation rule.

- `ECARuleIDSet FindECARules(Criteria c)`: Retrieves an adaptation rule. A criterion c (e.g. keywords to search in rule descriptions or event descriptions) is used to search for ECA rules. The set of ECARuleIDs whose criteria match c is returned. This set could be empty if there are no rules found.
- `Result Adapt(Event e, Parameter p)`: Invokes the Adaptability Manager which will make an adaptation choice.
- `ECARuleIDSet findECARules (Event e)`: This function queries the rules on which an event e can be possibly executed. The function returns a set of ECARuleID, which can be used to select appropriate rules for execution.
- `Boolean Depend(ECARule a, ECArule b, seq | conc)` create a dependency between the execution of the two rules. The Dependency can be parallel or sequential. Return false if a cycle of dependency is detected. A cycle might generate an infinite loop.
- `result ap-aware(condition, function)` call function each time condition is true, this function is mapped to AddECARule.

Policy Manager (PM). The Reference Model for Open Distributed Processing (RM-ODP) Enterprise Language [16] comprises concepts, rules and structures for the specification of a community, where a community comprises objects (human, software, or combinations thereof) formed to meet an objective. The Enterprise Language focuses on the purpose, scope and policies of the community, where an enterprise policy is a set of rules with a particular purpose related to the community's purpose. We see such an approach as providing useful concepts for high-level modelling of a distributed system in general and pervasive systems in particular. The goals of roles can change due to a lack of resources required to achieve a task, for example. This implies that policies can be altered and changed dynamically depending on the context. The policy manager ensures that policy specification described as *Obligation, Prohibition and Permission* associated with enterprise roles are respected in pervasive environments. An example of policy specification is `Prohibited (Role, Community, Action, Time)`. It specifies that a `Role` in a `Community` is `Prohibited` to do `Action` during `Time`.

The policy manager enables consistent addition, deletion and modification of policies. The Policy Manager is also responsible for maintaining consistency and resolving conflicts between roles objectives. (e.g. a role cannot be Authorised and Prohibited to do an action at the same time.). It also prioritises adaptation rules and context of interest based on objectives.

The Policy Manager provides the following services:

- `Boolean CheckPolicy(CommunityID, Roles,URI)`: Check policy
- `Boolean AddPolicy(CommunityID, Roles,URI)`:Add policy rules
- `Boolean RemovePolicy(CommunityID, Roles,URI)`:Remove policy rules
- `PolicyRuleIDSet RetrievePolicy(CommunityID, Roles,URI, criteria)`: Retrieve policy rules based on the criteria.

5.3 Service Layer

A dynamic and open environment is characterised by the frequent appearance of new services and network protocols. A unified mechanism is required to access services and network protocols. This layer wraps services and network protocols to fit into m3 modelling requirements. The wrapping allows upper layers to interact with the service layer using the event-based paradigm. This layer provides:

- at least the notification and discovery services. All the services fulfill a role, accessed as components using event notification service. The discovery service communicates via the notification service to the rest of the m3-RTE.
- a universal interface to the upper layers that enables the use of communication protocols such as SMTP, HTTP, TCP/IP or security protocols transparently. The protocol independent interface is similar to the Linda tuple space model [15,11]. It has generic interfaces `in(from,data,attr)` and `out(to,data,attr)` to communicate with components.

5.4 Interactions between Components

All components of the m3-RTE have to register to a service discovery server as illustrated in Figure 4. Components use a lookup service operation to get the reference of a registered service and interact with it. The reference is used to join the service. The `join` request will generate a unique channel ID in which peers will interact. The in/out operations are used to exchange data.

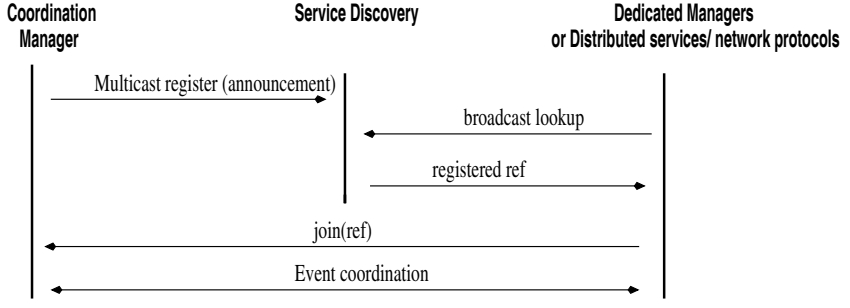


Fig. 4. m3-RTE initialisation

The discovery service registers only active tasks (services). A JINI like lease mechanism is used to achieve this.

Once all the components are initialised and registered, the m3-RTE can receive calls from an application. Figure 5 illustrates an example of a m3-RTE processing of an `http(GET(URL))` request from an application. Note that the coordination manager was programmed as illustrated which is not a mandatory

interaction (e.g we can change the coordination script and avoid using the policy manager). The Coordinator Manager receives the request and forwards it to the PM to check authorisations. If authorisations are granted then the request is sent to the AM. In this script, the AM is constantly updated by the CM about the context of the application. The AM selects an adaptation rule based on context and the adaptation action is executed. The reply is sent directly to the coordination manager which forwards it to the application.

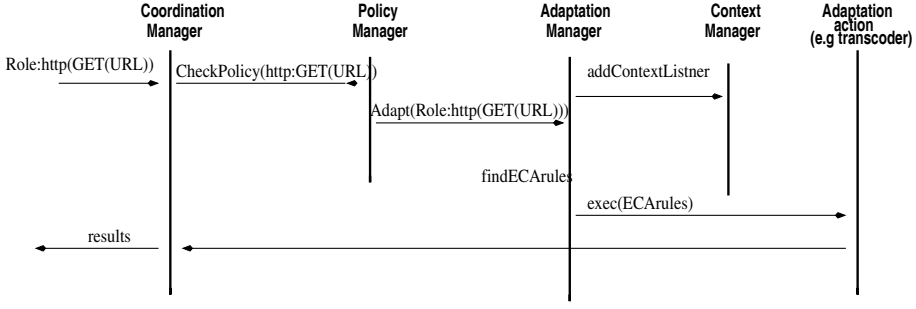


Fig. 5. Interaction between m3-RTE components

The python Code 7.1 relieves the user application (AP) from having to poll the value of the bandwidth. The AM will do it on behalf of the AP. Note that the code 7.1 does not prevent the AM, CM or PM to take actions provided that it has been programmed by the system programmer. For example Application transparent adaptation has to be programmed explicitly in MEADL which coordinates the other three managers.

6 m3-RTE Implementation Issues

This section shows how the Coordination Manager uses the Elvin [6] notification service to implement the execution of MEADL/XML scripts.

6.1 The Use of Elvin

Using point-to-point communication models such as RPC (JAVA-RMI) leads to rather static and non-flexible applications. Event notification reflects the dynamic nature of mobile applications. Interaction components are symmetrically decoupled. Such decoupling and content-based addressing features are the key to the scalability of a mobile platform. Event notification is the building block of the m3-RTE. The concept consists of sending a message to an intermediate system to be forwarded to a set of recipients. The benefits are: reduced dependencies between components and as a result greater flexibility. DSTC Elvin [6]

takes this simple concept and extends it to provide a routing service which is dynamic, scalable, and applicable for a wide variety of communication needs.

Elvin is a notification/messaging service with a difference: rather than messages being directed by applications from one component to another (address base), Elvin messages are routed to one or more locations, based entirely on the content of each message. Elvin uses a client-server architecture for delivering notifications. Clients establish sessions with an Elvin server process and are then able to send notifications for delivery, or to register to receive notifications sent by other components. Clients can act, anonymously, as both producers and consumers of information within the same session. The Elvin subscription language allows us to form complex regular expression.

Elvin is used to exchange events within the m3-RTE. MEADL expressions are parsed and transformed into Elvin subscriptions using Python [21]. Elvin's data is encoded in W3C Simple Object Access Protocol (SOAP 1.1) [25]. Such a choice provides for the use of different communication protocols in the future.

6.2 m3-RTE Prototype

The goal of the m3-RTE prototype is to create a test bed run-time environment for our current and future concepts in pervasive computing. The prototype includes the following managers:

Coordination Manager.

Code 6.1 summarises how the MEADL specification 5.2 is transformed into an XML specification. The XML expression in Code 6.1 is parsed with a Python [21] XML parser. Python Elvin subscription expressions are then generated from the XML parsing.

Context Manager. Currently, the context manager is able to manage device descriptions and location information in an RDF (Resource Description Framework) graph and location information for users and devices. This information has to be gathered from a variety of physical sensors (location devices, e.g. GPS, badges) and logical sensors (extracting/extrapolating location information from operating systems and communication protocols). A location service has been prototyped to gather location information from a variety of sensors, to process this information and convert it to a common format before submitting it to the CM. Our prototype comprises agents processing location data from several location devices and from the Unix and Windows operating systems. Its architecture allows easy incorporation of new sensor agents.

Adaptation Manager. Several adaptability methods have been prototyped including insertion of a variety of filters, migration of objects from resource poor sites to the static network and sending the result of the operation back to the object's client, restriction of an object interface (e.g. allowing access to a subset of operations only) and adaptation of Web content [12].

Policy Manager. The Policy Manager extends context information by adding descriptions of roles and policies associated with roles. Our Policy Manager prototype is in an early stage of development and is currently able to manage

Code 6.1 XML representation of MEADL

```
<rule>
       <trigger>
           <relation rel="and">
               <event name="http:get">
                   <argument ref="#action" />
                   <role="Director" />
               </event>
               <relation rel=&gt>
                    <argument ref="#action" />
                    <literal type="string">10,000</literal>
               </relation>
           </relation>
       </trigger>
       <guard>
           <not>
               <in context="secure_environment" />
           </not>
       </guard>
       <reaction>
           <call function="foo">
               <argument ref="#action" />
           </call>
       </reaction>
 </rule>
```

simple security oriented policies such as Permission and Prohibition. We use concepts from MEADL to specify simple policy rules. The following example shows an authorisation to the role Director to do an action if the Director belongs to the DSTC community and the context of the role is in the Head Quarters.

Code 6.2 Example of Policy specification

```
ON EVENT Director:action
        WHERE (Director.community = 'DSTC')
        IN CONTEXT ``HQ''
    {
     AUTHORIZE(action)
    }
```

7 Examples of Applications Using m3-RTE

The first example demonstrates the ability of m3-RTE to support application aware adaptation according to the terminology used in [20]. The second shows an application "transparent" adaptation.

7.1 Application Aware Adaptation

This section shows an example of the use of interfaces of m3-RTE managers. It shows code that implements the concept of application aware adaptation using m3-RTE in Python. AM and CM modules are imported and the ap-aware (application aware) method of the AM is called with "Callback" as parameter, Callback is a function that belongs to the application. The Callback function will be called once the bandwidth is `CM.bandwidth < 33`. The ap-aware expression is mapped directly onto an Elvin notification expression.

Code 7.1 Application aware adaptation

```
import AM  /* import Adaptation Manager */
import CM  /* import Context Manager context variable such as bandwith */
main()  /* main of the client AP */
  AM.ap-aware(CM.bandwith < 33 ,Callback)

def Callback()
   print ‘‘AP callback adaptation function ’’
```

7.2 Tickertape

This application shows how the m3 architecture adapts the Tickertape user interface rendering and the communication protocol according to the device capabilities and user preferences provided by the CM.

Fig. 6. Unix X Windows

Tickertape is an existing application that gives visual scrolling display of Elvin (event) traffic (see Fig 6). It is a tailorable window that can be used to both produce and consume notifications as desired. It displays notifications that the user subscribers to. In this prototyping, we use it for chat-style messages. Scrolling messages can have a MIME attachment that the user can view with a mouse click.

The demonstration consists of delivering the relevant Tickertape notifications with the appropriate communication protocol to four *existing* applications installed in three devices (Palm, Solaris Ultra10, Nokia7110). Note that this

example is not just a transcoding exercise as communication protocols and components involved in the delivering changes significantly for each adaptation.

We programmed the m3-RTE as follows. The CM has an abstract description of the class of devices such as Palm, Solaris Ultra10, Nokia7110 that contains their capabilities (e.g. WAP 1.0, WAP 1.2, touch-screen, Netscape). The AM has the adaptation rules that select a required component to exchange Elvin messages with the appropriate device (e.g. selection of proxies, WAP gateways). The MEADL script executed by the Coordination Manager forwards the request to the AM and is specified as follows:

Code 7.2 MEADL specification of Tickertape application

```
ON EVENT Project_leader:Ticker-connect(Elvin-server)
        IN CONTEXT ‘‘Palm’’ or ‘‘Nokia7710’’ or ‘‘Solaris’’
    {
      EMIT Adapt(Project_leader:Ticker-connect(Elvin-server));
    }
```

The list of subscribed channels for roles is known by the CM from the user profile. The content of Elvin messages to be forwarded and the communication protocols to be used are chosen by the AM. As a result Tickertape can be adapted in four different ways. When a user uses:

- a Unix (Solaris Ultra 10) workstation, the user sees tickertape message as in Figure 6. It uses X11. Messages and associated MIME type scroll in the bar. The Elvin protocol uses TCP.
- a non WAP enabled palm device, the user sees Figure 7(a). The communication protocol is a phone line connected to a modem and then to the server via HTTP. The Palm user interface cannot scroll messages. The associated MIME types are not transfered. The user has to click the receive button to check the latest update due to the absence of any push mechanism. A proxy component is involved in the interaction. The proxy provides persistence for Elvin by remaining subscribed on behalf of clients even while the client is off-line. When clients reconnect, stored notifications are delivered by the proxy to the clients.
- WAP 1.1 enabled 7710 Nokia Mobile phone, the user sees Figure 7(b). We use Kannel 1.0 (open source) to connect mobile devices to WAP servers. WAP (Wireless Application Protocol) is used to provide services on mobile phones by enabling mobile devices to function as simple web browsers. Kannel 1.0 implements the WAP 1.1 specification gateway to connect a phone line to the Elvin server. The WAP optimised protocols are translated to plain old HTTP by a WAP gateway. At the time we were writing the Nokia 7710 prototype there was no WAP browser that supports the push mechanism. Therefore the user had to press on a receive button to read tickertape messages. Note that MIME types are not sent.

– WAP 1.2 browser, the user sees Figure 7(c). We extend an open source WAP browser (ja2wap) in order to have the first WAP browser supporting the push mechanism compliant with the WAP forum specification. The browser automatically displays tickertape messages and the user does not need to press a button to load messages. MIME type attachments are not sent.

(a) Palm

(b) Nokia 7710 WAP 1.1

(c) Palm- WAP 1.2 - push

Fig. 7. Tickertape on mobile devices

8 Benefits of the m3 Architecture

The described examples allowed testing of some of the m3 architecture goals. The focus of the Tickertape example has been on the "plumbing" required to integrate multiple forms of data delivery to the application level. The m3-RTE prototype emphasises openness and generality of context description and management. The openness of the m3 architecture is demonstrated by the fact that we only added less than ten lines of code into the MEADL and AM to plug in a new component that enables tickertape viewing on different devices[2]. Such conciseness also demonstrates the expressiveness of our coordination language MEADL. Note that the adaptation mechanism can be more complex. For example some sensitive tickertape messages must not be sent to particular roles due to their physical location. Hence, m3-RTE features a universal approach to adaptation which can support a reasonable set of adaptation mechanisms such as discrete, continuous and community adaptation.

One of the main strengths of the coordination language MEADL is that it simple, role-based and can be translated into XML. Furthermore the use of an event based publish/subscribe concept as a core modelling concept allows

[2] This doesn't include the code required to write the actual components that implement the new communication and protocols which is a different engineering effort.

dynamic definition and management of event coordination between the Context Manager, Adaptation Manager, Policy Manager and other component based applications. The use of a publish/subscribe protocol eases the deployment of a distributed version of the m3 architecture. To our knowledge no middleware architecture that integrates and coordinates context, policy and adaptation to support reactive components exists.

9 Conclusion and Future Work

This paper describes an open architecture used as a toolkit to build adaptable enterprise applications for pervasive environments. It leverages heterogeneous devices, existing mobility related standards and protocols. We have identified context, adaptation and policy management as fundamental elements of pervasive computing applications, and shown how our architecture integrates these three notions. The use of a role-based component model and MEADL coordination gives the m3 architecture the ability to provide dynamic plug and play, and yet effectively coordinate enterprise components. The use of Elvin as the main communication paradigm within the m3 architecture enables a loose form of integration of components (or roles), permitting architectural reconfigurability as reactions to the ever-changing contexts of applications or occurrences of (a)synchronous events in a pervasive environment. A different way of building applications is also proposed, where the behaviour of the application is explicitly dependent on separately specified relevant context changes, required adaptations and behaviour policies. An application is therefore constructed by defining (and mixing and matching) these elements rather than by traditional monolithic application building.

As a next step, work is being carried out on both an extension to the prototype and on further adaptive and context-aware applications [13] in order to fully capture and test the concepts introduced by the m3 architecture to support pervasive computing environments. We also are distributing the architecture presented in Figure 3. Each node (or device) will have its own set of dedicated managers (context, policy and adaptation managers) which communicate to coordinate adaptations across several nodes. This permits a set of applications running on possibly different devices to adapt and react cooperatively. Global policies would then be needed which are disseminated to the nodes and integrated with the local policies. We also plan to evaluate the performance of the m3 architecture and compare it with conventional adaptive middleware.

References

1. Acharya, A. Ranganathan, M., Saltz,J. "A language for Resource-Aware Mobile Programs" *Mobile Object Systems: Towards the Programmable Internet, pages 111-130. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.*
2. Banavar, G., Beck, J., Gluzberg,E., E., Munson, J., Sussman, J. and Zkowski, D. "Challenges: An application Model for Pervasive Computing" *6th Proc annual Intl. Conference on Mobile Computing and Networking MOBICOM 2000, Boston August 2000*

3. Bianchi,G., Campbell, A.T, Liao, R. " On Utility-Fair Adaptive Services in Wireless Networks" *Proc of the 6th Intl Workshop on QoS IEEE/IFIP IWQOS'98 Napa Valley CA, May 1998*
4. Blair,G,. Blair,L.,Issarny, V,. Tuma, P,. Zarras, A,. The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms. *Middleware 2000 Proc LNCS 1795 - IFIP/ACM NY, USA, April 2000*
5. Composite Capabilities/Preference Profiles CC/PP - W3C - `http://www.w3.org/Mobile/CCPP/`
6. Arnold. D., Segall, B., Boot,J., Bond,A., Lloyd,M. and Kaplan,S Discourse with Disposable Computers: How and why you will talk to your tomatoes, Usenix Workshop on Embedded Systems (ES99), Cambridge Massachusetts, March 1999 also `http://elvin.dstc.edu.au/`
7. Davies, N. , Friday, A.Wade, S. and Blair, G. "A Distributed Systems Platform for Mobile Computing" *ACM Mobile Networks and Applications (MONET), Special Issue on Protocols and Software Paradigms of Mobile Networks, Volume 3, Number 2, August 1998, pp143-156*
8. Demers,A., Petersen, K.,Spreitzer, M., Terry,D., Theimer, M., Welch,B. "The Bayou Architecture: Support for Data Sharing among Mobile Users" *Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, December 1994, pages 2-7.*
9. Anind K. Dey. "Enabling the Use of Context in Interactive Applications" *Doctoral Consortium paper in the Proceedings of the 2000 Conference on Human Factors in Computing Systems (CHI 2000), The Hague, The Netherlands, April 1-6, 2000, pp. 79-80.*
10. C. Esftratiou, K. Cheverst, N. Davies, A. Friday, An Architecture for the Effective Support of Adaptive Context-Aware Applications, in Proc. of 2nd International Conference on Mobile Data Management, Hong-Kong, January 2001. Lecture Notes in Computer Science, Vol 1987.
11. Eric Freeman, et al JavaSpaces(TM) Principles, Patterns and Practice *The Jini(TM) Technology Series June 1999* also `http://www.sun.com/jini/specs/jsspec.html`
12. K. Henricksen, and J. Indulska., "Adapting the Web Interface: An Adaptive Web Browser", Proceedings Australasian User Interface Conference 2001, *Australian Computer Science Communications, Volume 23, Number 5, 2001.*
13. K. Henricksen, J. Indulska and A. Rakotonirainy "Infrastructure for Pervasive Computing: Challenges", Workshop on Pervasive Computing and Information Logistics at Informatik 2001, Vienna, September 25-28, 2001.
14. J. Indulska, S.W. Loke, A. Rakotonirainy, V. Witana, A.Zaslavsky "An Open Architecture for Pervasive Systems" The Third IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems September 2001 Krakow.
15. Wyckoff, P., McLaughry,S. W., Lehman, T. J. and Ford,D. A. "TSpaces" *IBM Systems Journal, August 1998* also `http://www.almaden.ibm.com/cs/TSpaces/`
16. Information Technology - Open Distributed Processing - Reference Model - Enterprise Language (ISO/IEC 15414 — ITU-T Recommendation X.911) July 1999
17. Joseph A., Kaashoek F. "Building reliable mobile-aware applications using the Rover toolkit" *MOBICOM '96. Proceedings of the second annual international conference on Mobile computing and networking, pages 117-129'*
18. Kon, F. et al Monitoring, Security, and Dynamic Configuration with dynamicTAO Reflective ORB *Middleware 2000 Proc LNCS 1795 - IFIP/ACM NY, USA, April 2000*

19. Medvidovic N, Taylor "A Framework for Classifying and Comparing Architecture Description Language " *Proc Software engineering Notes, ESEC/FSE'96 - LNCS Vol 22 number 6 November 1997*
20. Noble, B., Satyanarayanan, M., Narayanan, D. Filton J.E, Flinn J.,Walker K. , "Agile Application Aware Adaptation for Mobility" *16th ACM Symposium on Operating System Principles 1997*
21. Python programming Language `http://www.python.org`
22. Renesse,v-R. Birman, K., Hayden,M,., Vaysburd, A,., Karr, D. "Building Adaptive systems using Ensemble" *Cornell University Technical Report, TR97-1638, July 1997.*
23. Rakotonirainy A., Bond A., Indulska,J,. Leonard.D. SCAF: A simple Component Architecture Framework. *Technology of Object-Oriented Languages and systems TOOLS 33 - June 2000 - IEEE Computer Society - Mont St Michel France*
24. Satyanarayanan, M. The Coda Distributed File System *Braam, P. J. Linux Journal, 50 June 1998*
25. Simple Object Access Protocol (SOAP) 1.1 `http://www.w3.org/TR/SOAP/`
26. Sun One brings mobile intelligence to the wireless world `http://www.sun.com/2001-0710/feature/`
27. Extensible Markup Language (XML) 1.0 `http://www.w3.org/XML/`
28. Want, Z. and Garlan D., "Task-Driven Computing". *Technical Report, CMU-CS-00-154, School of Computer Science CMU May 2000*
29. Wireless Application Protocol - WAP Forum Specifications `http://www.wapforum.com/what/technical.htm`

Experiments in Composing Proxy Audio Services for Mobile Users

Philip K. McKinley, Udiyan I. Padmanabhan, and Nandagopal Ancha

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA
{mckinley,padmana3,anchanan}@cse.msu.edu

Abstract. This paper describes an experimental study in the use of a composable proxy framework to improve the quality of interactive audio streams delivered to mobile hosts. Two forward error correction (FEC) proxylets are developed, one using block erasure codes, and the other using the GSM 06.10 encoding algorithm. Separately, each type of FEC improves the ability of the audio stream to tolerate errors in a wireless LAN environment. When composed in a single proxy, however, they cooperate to correct additional types of burst errors. Results are presented from a performance study conducted on a mobile computing testbed.

1 Introduction

The large-scale deployment of wireless communication services and advances in portable computers are quickly making "anytime, anywhere" computing into a reality. One class of applications that can benefit from this expanding and varied infrastructure is collaborative computing. Examples include computer-supported cooperative work, computer-based instruction, command and control, and mobile operator support in military/industrial installations. A diverse infrastructure enables individuals to collaborate via widely disparate technologies, some using workstations on high-speed local area networks (LANs), and others using wireless handheld/wearable devices.

Collaborative applications differ widely in their quality-of-service requirements and, given their synchronous and interactive nature, they are particularly sensitive to the heterogeneous characteristics of both the computing devices and the network connections used by participants. One approach to accommodating heterogeneity is to introduce a layer of adaptive middleware between applications and underlying transport services [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]. The appropriate middleware framework can help to insulate application components from platform variations and changes in network conditions. Moreover, a properly designed middleware framework can facilitate the development of *new* applications through software reuse and domain-specific extensibility.

We are currently conducting a project called *RAPIDware* that addresses the design and implementation of middleware services for dynamic, heterogeneous

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 99–120, 2001.

environments. A major goal of the RAPIDware project is to develop adaptive mechanisms and programming abstractions that enable middleware frameworks to execute in an autonomous manner, instantiating and reconfiguring components dynamically in response to the changing needs of client systems. Moreover, the RAPIDware methodology is intended to apply not only to communication protocols, but also to fault tolerance components, security services, and reconfigurable user interfaces.

For RAPIDware to achieve these goals, it must be possible to compose and reconfigure middleware services at run time. Such adaptability is particularly important in *proxy servers*, which are often used to mitigate the limitations of mobile hosts and their wireless connections [17,18,19,20,9,12,21]. Adopting the terminology of the IETF Task Force on Open Pluggable Edge Services (OPES) [22], we view RAPIDware proxies as composed of many *proxylets*, which are functional components that can be inserted and removed dynamically at run time without disturbing the network state. Moreover, it should not be necessary to compile these components into the proxy code *a priori*, but instead they should be mobile components that are uploaded into proxies at run time.

Earlier in the RAPIDware project, we designed a composable proxy framework based on detachable Java I/O streams [23]. The framework enables proxylets such as filters and transcoders to be dynamically inserted, deleted, and reordered on extant data streams. As such, detachable streams provide the "glue" needed to support the dynamic composition of proxy services. In this paper, we demonstrate how this framework can be used to combine two independent forward error correction (FEC) proxylets, one using block erasure codes [24], and the other using the GSM 06.10 encoding algorithm designed for wireless telephones [25]. Separately, each type of FEC improves the ability of the audio stream to tolerate errors in a wireless LAN environment. However, by simply plugging both proxylets into the framework, they cooperate to correct additional types of burst errors occurring on the wireless network.

The remainder of the paper is organized as follows. In Section 2, we discuss the RAPIDware project and its foundations. Section 3 reviews the design and operation of detachable streams. Section 4 describes the individual operation of the two audio proxylets and their combined functionality when coupled in a single proxy server. Section 5 presents results of an experimental study on a mobile computing testbed. Section 6 discusses related work on composable proxy services. Section 7 presents our conclusions and discusses future directions for the RAPIDware project.

2 Background

The RAPIDware project evolved from our prior work on Pavilion [26], an object-oriented framework to support synchronous web-based collaboration. Like other collaborative frameworks, Pavilion can be used in a default mode, in which it operates as a collaborative web browser [27]. In addition, Pavilion enables a developer to construct new multi-party applications by reusing and extending existing components: browser interfaces, communication protocols, a leadership

protocol for session floor control, and a variety of proxy servers. Pavilion uses proxy servers for several tasks: transcoding and filtering of data streams to reduce bandwidth and load on mobile clients [28], data caching for memory-limited handheld devices [29], forward error correction for real-time isochronous communication [30,31] and reliable data delivery on wireless networks [21].

The RAPIDware project extends Pavilion by developing programming abstractions and supporting mechanisms to *automate* the instantiation and reconfiguration of middleware components, such as proxy services, in order to accommodate resource-limited hosts and dynamic network conditions. A key principle in RAPIDware is to separate adaptive middleware components from non-adaptive, or *core*, middleware services, thereby facilitating dynamic reconfiguration of components at run time. Towards this end, we are designing an extensible set of adaptive components, which we refer to as *raplets*. Figure 1 depicts a simple example of the intended relationship among raplets and other components in a collaborative application. Shown are three types of systems connected by several data streams. Two of the systems use proxy servers to transcode or otherwise modify data prior to its transmission on wireless links.

RAPIDware uses two main types of raplets, *observers* and *responders*, to accommodate heterogeneity and adapt to variations in conditions. The observers collectively monitor the state of the system. When an observer detects a relevant event, the observer either instantiates a new responder or requests an extant responder to take appropriate action to address the event. Example events include changes in the quality of a network connection, disparities among collaborating devices, and changes in user/application preferences or policies. Responder raplets are programmed to handle such events by instantiating new components or modifying the behavior of a communication protocol or user interface.

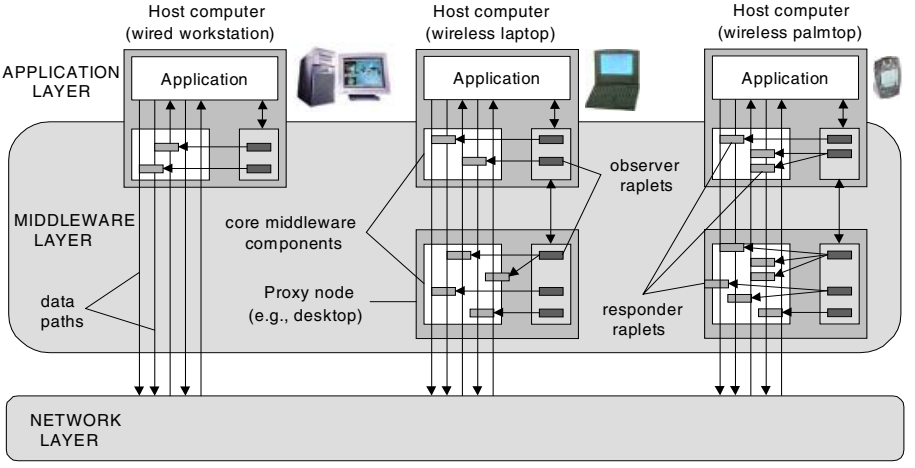


Fig. 1. Configuration of RAPIDware adaptive middleware components.

Besides RAPIDware, several other projects have addressed the issue of configurable and adaptive proxy services for mobile hosts [9,12,32]; these designs enable filters and transcoders to be configured at run time in order to match the capabilities of users devices and networks (see Section 6). If all possible proxylets are known to the proxy *a priori*, then this task is relatively simple. However, it may be difficult to predict the possible variations and sources of proxy services that will be needed. In RAPIDware, we seek to create a framework that enables third-party proxylets to be authenticated and dynamically inserted into an existing proxy. We focus on "lightweight" proxies, typically executed on workstations and other hosts accessible to the mobile user. To manipulate data streams sent to and from clients systems, we require mechanisms that enable the stream to be disconnected and redirected to another piece of code, without compromising the integrity of the data or that of the network state. Next, we review the operation of detachable Java I/O streams, which provide this functionality.

3 Framework Overview

To illustrate the intended operation of RAPIDware proxy components, let us consider the example configuration shown in Figure 2. Suppose that a proxy server is instantiated to support one or more mobile users, connected via a wireless LAN, who are participating in a collaborative session with other users on the wired network. Among other duties, such a proxy might receive live audio/video streams on the wired network, pass them to proxylets that transcode them into a lower bandwidth formats, and forward the new streams on the wireless network. We use the term *filter* to refer to this particular subclass of proxylet, which includes the audio FEC proxylets described in this paper.

Now let us assume that the user wants to maintain the connection as she moves from her office (near the wireless access point) to a conference room down the hall. In such environments, the packet loss characteristics can change dramatically over a distance of only several meters [21]. When losses rise above a given level, the RAPIDware system should insert an FEC filter into such

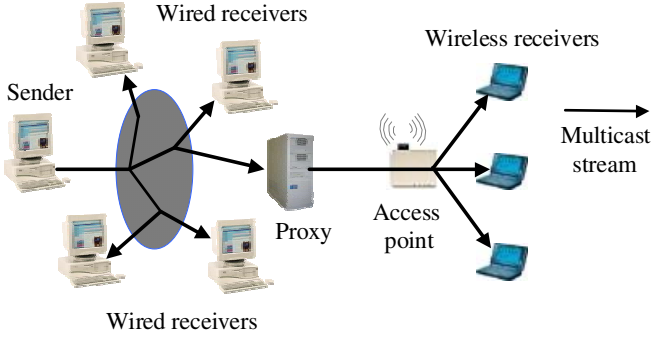


Fig. 2. Proxy configuration for nodes on a wireless LAN.

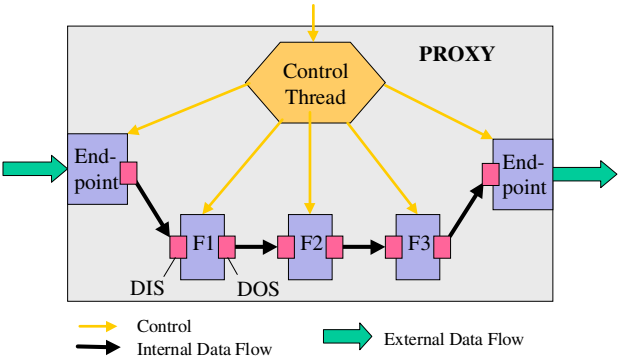


Fig. 3. Composition of proxy filters.

data streams in order to make them stream more resilient to losses. However, the insertion should not disturb the connections to the data sources. Moreover, since the FEC filter may be data-specific (e.g., placing more redundancy in I frames than in B frames in MPEG streams [33]), we need to consider the format of the stream in order to start the FEC filter at the proper point in the stream. Finally, it is possible that the application itself was dynamically downloaded to the mobile host, in which case the associated filter for low-bandwidth connections may not have been known to the proxy in advance. In this case, the filter would need to be fetched from a repository and instantiated on the proxy.

To support such functionality, we began by designing objects and interfaces to enable filters to be inserted, deleted, and chained together [23]. Figure 3 depicts the resulting software structure of a RAPIDware proxy and its operation on a single data stream. The proxy receives and transmits the stream on *EndPoint* objects, which encapsulate the actual network connections. Each EndPoint has an associated thread that reads or writes data on the network, depending on the configuration of the EndPoint. A ControlThread object is responsible for managing the insertion, deletion, and ordering of the filters associated with the stream. In this example, the proxy comprises three filters, F1, F2, and F3. The key support mechanisms are detachable stream objects, namely, DetachableInputStream (DIS) and DetachableOutputStream (DOS). The DIS and DOS are used for all communication among filters, and between filters and EndPoints. The DIS and DOS can be stopped (paused), disconnected, and reconnected, enabling the dynamic redirection and modification of data streams. Now, let us briefly describe the main classes that make up the framework.

DetachableOutputStream and DetachableInputStream. These classes are based on the the Java PipedOutputStream and PipedInputStream classes, respectively. DetachableOutputStream extends the base java.io.OutputStream class, and DetachableInputStream extends the base java.io.InputStream class. In addition to overriding many of the base class methods, we have also included additional state variables and methods to implement the functionality needed to support com-

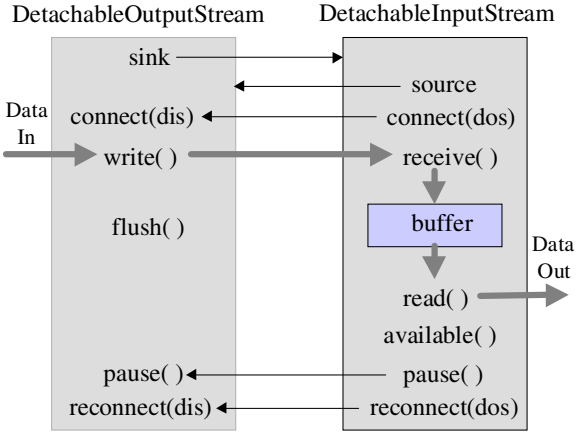


Fig. 4. Configuration of detachable streams.

posable filters. Figure 4 illustrates the relationship between a DetachableOutputStream (DOS) and a DetachableInputStream (DIS). The connect() method is used to associate a specific output stream with a specific input stream. Among other initializations, the connect() method sets DOS.sink and DIS.source variables so as to identify the other half of the connection. The DIS.connect() method simply calls the DOS.connect() method, which actually does this initialization.

As with their piped counterparts, the data written to the stream is buffered at the DIS side. An invocation of the DOS.write() method results in a call to the DIS.receive() method, which places the data in the buffer. The DIS.available() method returns the number of bytes currently in the buffer, and data is retrieved from the buffer using the DIS.read() method. The DOS.flush() method can be used to force any buffered output bytes to be written out, and notifies any readers that bytes are waiting in the pipe. Unlike the PipedOutputStream and PipedInputStream classes, both the DOS and DIS can be temporarily paused and reconnected to other streams. The pause() method has to be called before any actual disconnection and switching of the data stream. The method blocks attempts to write to the buffer and ensures that all the data has been read from the buffer. It also sets flags indicating that the two sides are no longer connected. Once the pause method returns, the reconnect() method can be used to attach the DIS and DOS to other filters. If the buffer has not yet emptied, the caller is suspended until the buffer becomes empty. The reconnect() method checks whether the call is valid (not still in the connected state) and then mimics the actions of the connect() method in setting several global variables.

ControlThread. This class is used to manage the configuration of filters on a given stream supported by the proxy. The class maintains the Filter Vector, a dynamic array that holds references to the currently configured filters. The class implements methods to insert and delete filters from the Filter Vector, as well as methods that allow the ControlManger class to query about the available filters

and the methods they support. The ControlThread receives commands from across the network, either from the mobile client, from an application server, or from the control manager.

Filter and FilterContainer. The base class for proxylets is Detachable.Proxylet. Any proxylet that is to be used in the framework needs to extend this base class. The Proxylet class extends the Thread class and thus is inherently runnable. The Filter class extends the base class and is meant to be further extended by all proxy filters that are to be run in the proposed framework; see Figure 5. The author of a filter would write the functional code as the run() method of the filter. The Filter class contains a DIS and a DOS object, along with their corresponding standard references, called *DIS* and *DOS*. The ControlThread uses these references to manipulate the stream connections. A group of methods (e.g., setDIS, setDOS, getid) is used to establish references to the DIS and DOS in the filter code itself. The FilterContainer class is used to hold an array of Filter objects. This functionality is required when new filters are uploaded into the framework from remote hosts. The FilterContainer class has methods to obtain the number of Filters available and an enumeration method to return a String enumeration of the Filter objects names.

EndPoint. These are extensions of Filters that are instantiated by the ControlThread for providing input and output services to the framework. If the I/O is network-based, then the EndPoint objects would be a EndPointSocketReader and EndPointSocketWriter. If the I/O is a non-network stream then we would

```
public class Filter extends Proxylet implements Serializable, Cloneable {
    String idString = new String("NA/");  // the filter identifier

    // The DIS, which will be manipulated by the ControlThread
    public final DetachableInputStream DIS = new DetachableInputStream();

    // The DOS, which will be manipulated by the ControlThread
    public final DetachableInputStream DOS = new DetachableOutputStream();

    private int objid = -1;  // An object id for convenience

    // The setDIS and setDOS methods allow on to set up their own
    // references names to the actual DIS and DOS
    public DetachableInputStream setDIS(){
        return(this.DIS);    }

    public DetachableOutputStream setDOS(){
        return(this.DOS);    }
```

Fig. 5. Excerpt from the Filter class.

use an EndPointStreamReader and EndPointStreamWriter. Each EndPoint contains an active thread that handles I/O to and from the proxy. Combined with the ControlThread, two EndPoints comprise a "null" proxy, that is, one that simply forwards data without modifying it. Upon insertion of a filter between the EndPoints, the stream is redirected through the new code.

ControlManager. As described earlier, observer and responder components provide adaptive functionality in RAPIDware, based on predefined user preferences and device/network descriptors. To test the behavior of RAPIDware filters and related components, however, we found it useful to develop a user interface that can be used to manage RAPIDware-based collaborative sessions. The ControlManager class has a Swing-based GUI designed for this purpose. Based on responses to queries, the ControlManager constructs a graphical representation of the state of the proxy, including the current configuration of filters, based on the methods available in the ControlThread. This design enables the same ControlManager to be used with different types of proxies. The current ControlManager GUI is shown in Figure 6. Although it is rather primitive and used only for our testing, we intend to improve it as the RAPIDware project progresses. The interface displays the current configuration of proxy filters and has pull-down menus and dialog text boxes that allow an administrator to insert and remove filters at specified locations in a given stream. The ControlManager uses serialization to deliver new filters to the proxy, as they are requested.

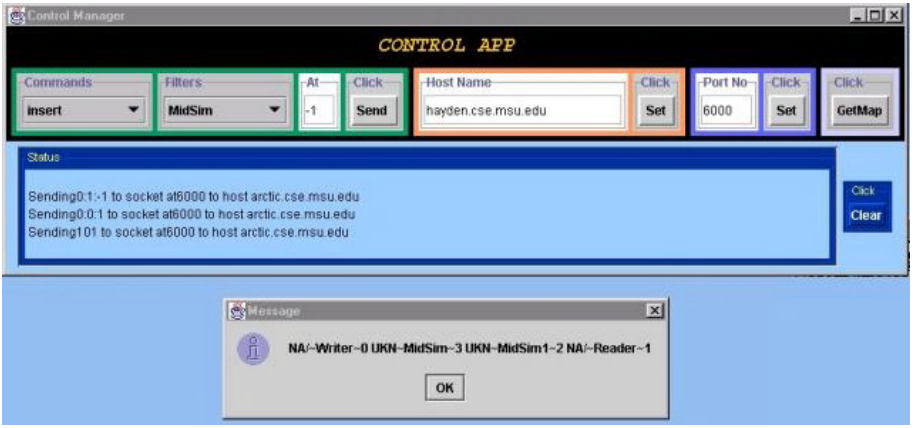


Fig. 6. Screen capture of ControlManager.

4 The Audio Filters and Their Composition

To evaluate the operation and performance of the RAPIDware DIS/DOS proxy framework, as well to determine which new features are needed, we are constructing several filters and other proxylets for use in the framework. For example, we are porting proxy services from Pavilion (such as video transcoding and reliable

muliticasting services) to the new framework, and we are developing new proxylets related to security management and handoff of applications among different graphical displays. As part of this testing, we have developed two different audio FEC filters, each of which can be independently inserted in a running audio stream. Moreover, we note that chaining together the two filters can provide a level of error correction beyond what either filter can provide separately. We begin by discussing the packet loss characteristics of wireless LANs (WLANs) and their effects on interactive audio streaming, followed by details of the two audio FEC filters and a description of their combined operation.

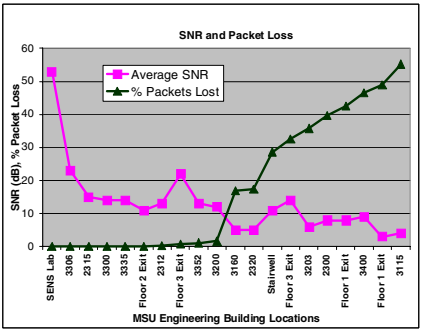
Characteristics of Wireless LANs. The performance of group communication services, such as audio multicasting for collaborative applications, is affected by four main characteristics of WLANs. First, the packet loss rates are highly dynamic and location-dependent [21]. Figure 7(a) demonstrates this behavior by plotting the relationship between signal-to-noise ratio (SNR) and packet loss rate during a short excursion within range of the wireless access point in our laboratory. The results demonstrate the highly variable loss rate that can occur in such environments, and the SNR values quickly drop below the level of 20 dB that is typically considered acceptable.

Second, the loss characteristics of a WLAN are very different from those of a wired network. In a wired domain, losses occur mainly due to congestion and subsequent buffer overflow. In wireless domain, however, losses are more commonly due to external factors like interference, alignment of antennae, ambient temperature, and so on. Figures 7(b) and (c) show example burst error distributions for two locations near our laboratory, where our wireless access point is located. Location 1 is just outside our laboratory, and location 2 is approximately 25 meters down a corridor. In both cases, while some large bursts occur, many are very short, and most "burst" errors comprise a single packet loss. To minimize the loss rate in terms of bytes, smaller packets are preferred, as shown in Figure 7(d). Apparently, the errors within larger packets are relatively localized, so that by sending several smaller packets, some number of them will be received successfully, whereas the larger packet would be lost.

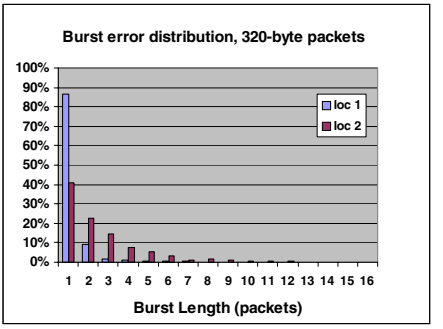
Third, the 802.11b CSMA/CA MAC layer provides RTS/CTS signaling and link-level acknowledgements for unicast frames, but not for multicast frames. The result is a higher packet loss rate as observed by applications using UDP/IP multicast, as opposed to UDP unicast. Figure 8 demonstrates this behavior for a typical location just outside our laboratory. As a result, multicast transmissions require more redundancy from the application level.

Fourth, since the wireless channel is a shared broadcast medium, it is important to minimize the amount of feedback from receivers. Simultaneous responses from multiple receivers can cause channel congestion and burden the access point, thereby hindering the forward transmission of data. Thus, it is desirable to use proxy-based FEC instead of proxy-based retransmissions for real-time communication such as interactive audio streams.

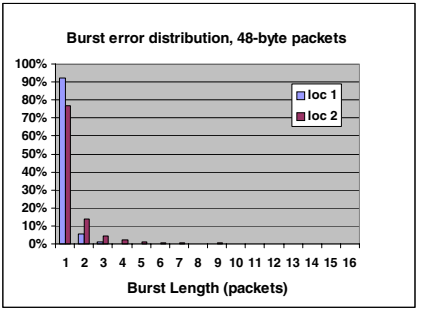
Audio Filter Using Block-Oriented FEC. Our first audio filter uses an FEC mechanism that can be applied to any data type. It recovers packets that have



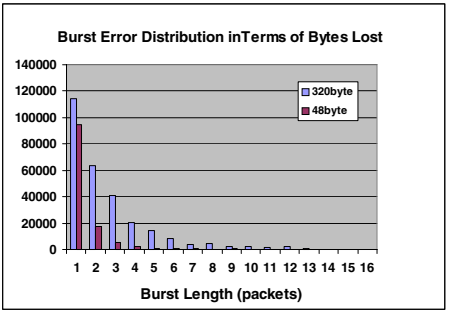
(a) packet loss vs. SNR



(b) burst distr., 320-byte packets



(c) burst distr., 320-byte packets

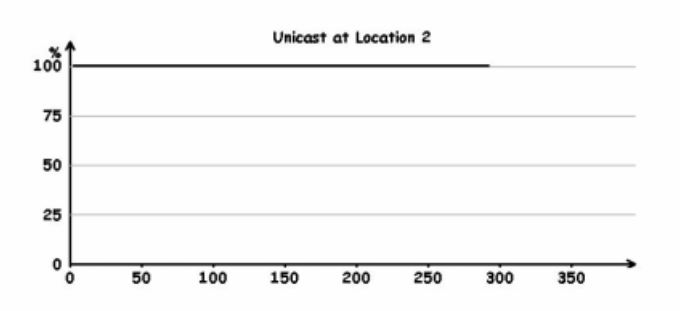


(d) 320-byte vs. 48-byte packets

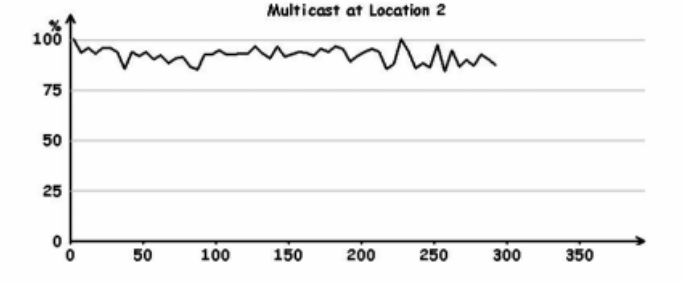
Fig. 7. Typical characteristics of a WLAN channel.

been "erased" due to an error detected by the CRC check in the data link layer. As shown in Figure 9, an (n, k) *block erasure code* converts k source packets into n encoded packets, such that any k of the n encoded packets can be used to reconstruct the k source packets [34]. In this paper, we use only *systematic* codes, which means that the first k of the n encoded packets are identical to the k source packets. We refer to the first k packets as *data* packets, and the remaining $n-k$ packets as *parity* packets. Each set of n encoded packets is referred to as a *group*. The advantage of using block erasure codes for multicasting is that a single parity packet can be used to correct independent single-packet losses among different receivers [24]. These codes are lossless, in that a successful decoding produces exactly the original data.

Recently, Rizzo [24] studied the feasibility of software encoding/decoding for packet-level FEC, using a particular block erasure code called the Depending on the values of k and $n-k$, Rizzo showed that this code can be efficiently executed on many common microprocessors. Rizzo's public domain FEC encoder/decoder library [24] is implemented in C and has been used in many projects involving



(a) UDP unicast reception rate



(b) UDP/IP multicast reception rate

Fig. 8. Sample packet loss traces for UDP/IP unicast and multicast.

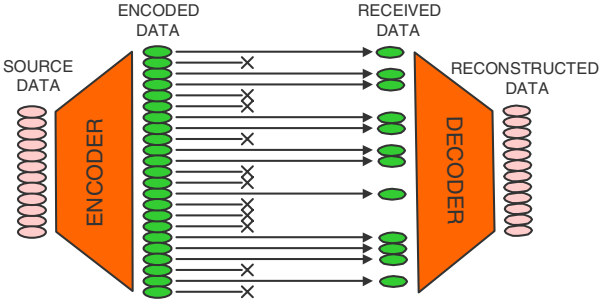


Fig. 9. Operation of FEC based on block erasure codes.

multicast communication [35,36,37,38,39], including our own prior studies [21, 30,31]. Given the emphasis on portability and code mobility in the RAPIDware project, however, we decided to switch to an open-source Java implementation of Rizzo's FEC library, available from Swarmcast [40]. In general, we found the Swarmcast library to provide a convenient interface and good performance. Although considerably slower than the C implementation, the Java version was able to satisfy real-time audio encoding and decoding requirements on systems with modest processing power.

Figure 10 shows the operational schematic of the major components of the audio application when this FEC filter is running. The audio recorder was built as a pure Java application making use of the Java Sound API. Specifically, the recording thread uses the javax.sound.* classes to read audio data from a workstation's sound card and send it to the proxy via the wired network. The encoding of the data is 16 bits per sample, PCM signed, at the standard rate of 8000 Hz over a mono channel. The audio receiver uses one thread to read data from the network and store it in a circular buffer. A second thread reads the data and uses the Java Sound API to play it.

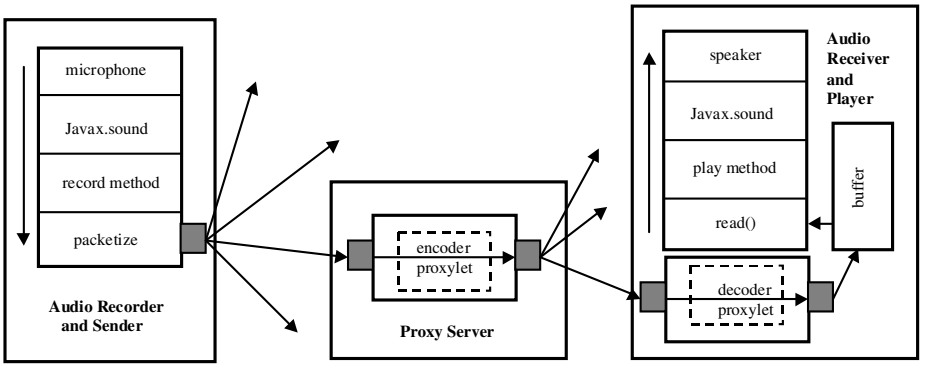
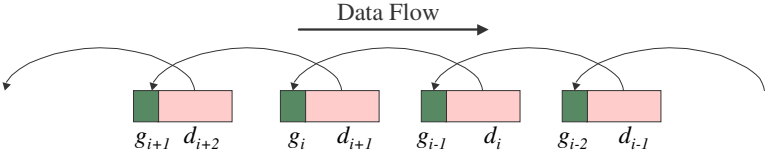


Fig. 10. Operational block diagram of the streaming audio configuration.

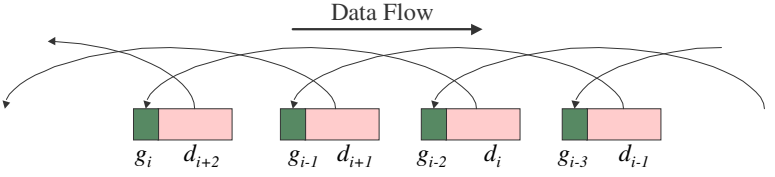
For data streams directed toward the client system, the encoder is instantiated on the proxy and the decoder on the client (the placement is reversed for outbound streams). The encoder and decoder filters have the same basic construction and simply invoke different methods in the Swarmcast FEC library. The encoder filter creates the FEC encoder by instantiating a class of the FECCodefactory's default FEC codec. The thread then collects k packets and constructs references to these. The FEC encoder is then called, which returns n packets contained in a new reference buffer. These packets are labeled with a group identifier and are written to the data stream. The decoder at the client requires any k packets in a given group in order to decode the original k data packets. The decoder thread thus reads up to k packets in a given group, after which additional packets are discarded. This data is passed to the decode

method of the FEC codec, which returns the k original data packets, which are forwarded to the client application.

GSM Audio Filter. While block-oriented FEC approaches are effective in improving the quality of interactive audio streams on wireless networks [30], the group sizes must be relatively small in order to reduce playback delays. Hence, the overhead in terms of parity packets is relatively high. An alternative approach with lower delay and lower overhead is *signal processing based FEC (SFEC)* [41, 42], in which a lossy, compressed encoding of each packet i is piggybacked onto one or more subsequent packets. If packet i is lost, but one of the encodings of packet i arrives at the receiver, then at least a lower quality version of the packet can be played to the listener. The parameter θ is the offset between the original packet and its compressed version. Figure 11 shows two different examples, one with $\theta = 1$ and the other with $\theta = 2$. As mentioned, it is also possible to place multiple encodings of the same packet in the subsequent stream, for example, using both $\theta = 1$ and $\theta = 3$.



(a) GSM encoding with $\theta = 1$



(b) GSM encoding with $\theta = 2$

Fig. 11. Different ways of using GSM encoding on a packet stream.

The SFEC RAPIDware filter that we developed uses GSM 06.10 encoding [25] for generating the redundant copies of packets. The full rate speech codec in GSM is described as Regular Pulse Excitation with Long Term Prediction (GSM 06.10 RPE-LTP). Although GSM is a CPU-intensive coding algorithm – it is 1200 times more costly than normal PCM encoding [42] – the bandwidth overhead is very small. Specifically, the GSM encoding creates only 33 bytes for a PCM-encoded packet containing up to 320 bytes (160 samples).

Our filter uses a freeware Java version of the GSM codec available from Tritonus (www.tritonus.org).

The filter can work in one of two ways. The first is to piggyback encoded data on subsequent packets, as shown in Figure 11. The second is to compute encodings for multiple packets and create a new packet of encoded data, to be inserted in the stream at a later point. This second method is useful when it is important to keep packet sizes small, as in a wireless LAN.

Combining the Audio Filters. Either filter can be inserted separately into an audio stream. However, we can also insert both filters, as shown in Figure 12 (in the figure and in the remainder of the paper, we'll refer to the filters simply as "GSM" and "FEC"). If the direction of the audio channel were reversed, then the encoders would reside on the client, and the decoders on the proxy.

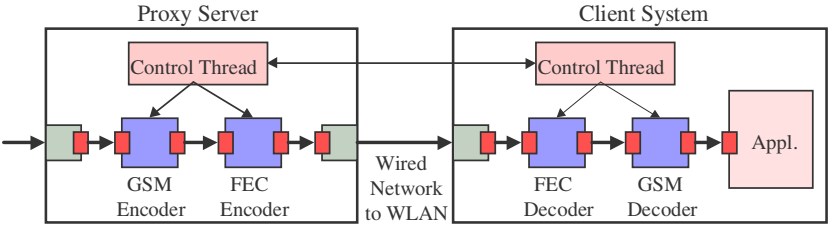


Fig. 12. Configuration of audio filters on proxy and client.

Figure 13 shows a particular example of the two encoders working together. The GSM encoder is configured to operate on a packet basis, computing a GSM encoding for every three data packets and inserting the new packet after $3 \times \theta$ data packets in the following packet stream. Each group of four packets (three data packets and one GSM packet) is forwarded to the FEC filter, which is configured to compute two parity packets using a (6,4) block erasure code. The (6,4) code can recover at most two lost packets per group, hence covering the most common packet loss cases, and does so without any loss in quality. Combining FEC and GSM code can tolerate relatively long isolated burst errors of up to $(\theta + 2)n - 2k$ packets, depending on the location of the lost packets relative to group boundaries. Of course, if GSM instead of FEC is used to reconstruct a packet, the quality of the resulting signal will be lower than that of the original.

5 Experimental Evaluation

In order to study the operation and performance of audio filters separately and in combination, we conducted a set of experiments on the mobile computing testbed in our Software Engineering and Network Systems (SENS) Laboratory. Results of using block erasure codes alone are described elsewhere [23,30], so here we concentrate on the GSM encoder and the combination of the two methods. Detailed evaluations are described in a companion technical report [43].

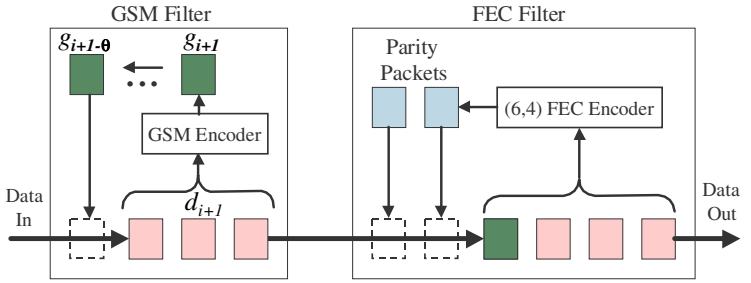


Fig. 13. Operation of combined audio filters.

Testing Environment. The mobile testbed currently includes conventional workstations connected by a 100 Mbps Fast Ethernet switch, three 802.11 WLANs (Lucent WaveLAN, Proxim RangeLAN2, and Cisco/Aironet), and several mobile handheld and laptop computer systems. All tests reported here were conducted on the Aironet WLAN, which uses direct sequence spread spectrum signaling and has a raw bit rate of 11 Mbps. We used both wired desktop PCs and wireless laptop PCs as participating stations in the experimental configuration depicted in Figure 14. The sender, proxy, and control manager were executed on dual-processor 400/450 MHz desktop workstations, while the mobile nodes were 300 MHz laptops equipped with Aironet network interface cards. The Aironet access point and the participating wired stations were located in our laboratory, while the locations of the mobile nodes were varied. Although the proxy multicasts the audio stream on the WLAN, here we report the results for a single receiver.

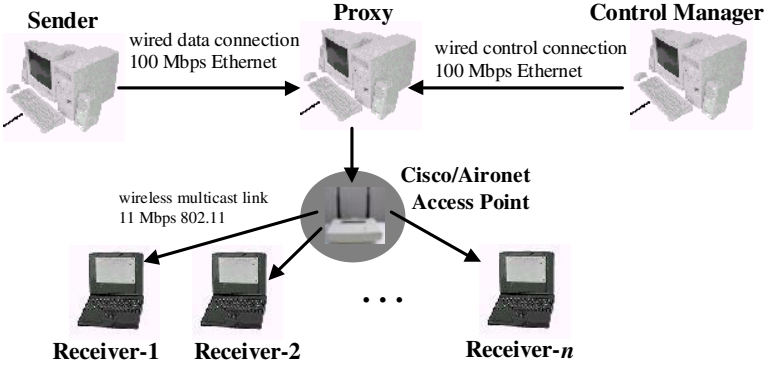
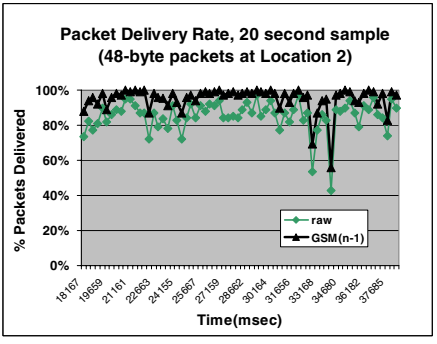


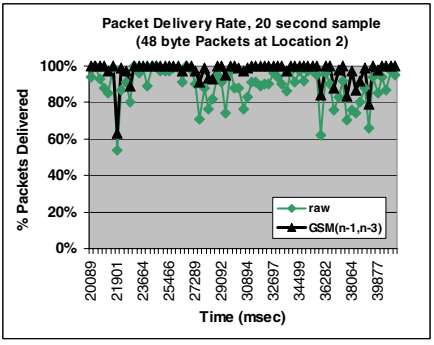
Fig. 14. Physical configuration of experimental components.

Initially, both the proxy and client are configured as "null" filters, as the Endpoints simply read and retransmit data. In an actual RAPIDware environment, observer threads at the client would monitor the packet loss rate and burst length distribution, and would instruct the ControlThread on the proxy when to insert the FEC or GSM filter. In order to control the testing, however, we used the Control Manager GUI to insert the filters manually.

GSM Experiments. We tested the GSM filter in isolation, setting θ to different values and using both single and double copies of the encoded data. In all cases, small 48-byte packets produced considerably better results than 320-byte packets, so we report only the former here; many additional results can be found in [43]. Figure 15 shows a sample of the results. In Figure 15(a), we placed a single encoded copy of each packet i in its successor packet $i+1$. In Figure 15(b), we placed one encoded copy in packet $i+1$ and one in packet $i+3$. Using multiple copies produces a clear advantage in terms of packet delivery rate.



(a) $\theta = 1$, 48-byte packets

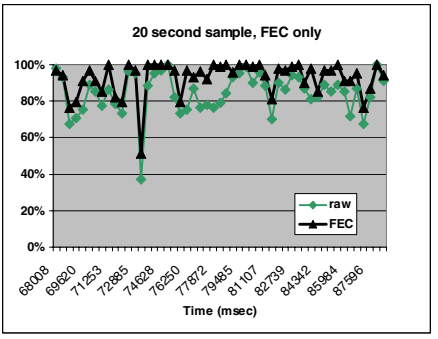


(b) $\theta = 1, 3$, 48-byte packets

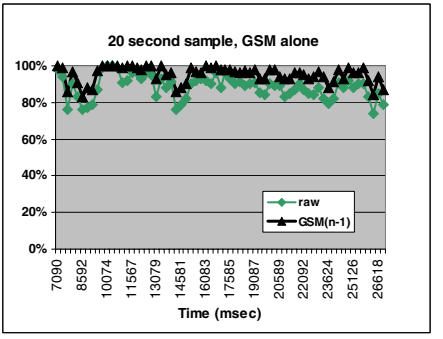
Fig. 15. Sample traces results for the GSM filter with 48-byte packets.

Figures 16(a) through (c), respectively, show sample traces of using the FEC(8,4) and GSM(n-1) filters, alone and in combination. The combination appears to be most effective in recovering data, and this result is confirmed by Figure 16(d), where we averaged the results of 5 two-minute runs and computed the packet delivery rate for each of the methods at a particular location in our building. By combining the two filters, we are able to reconstruct 97% of the audio data, even though the raw packet delivery rate at this location was only 83%.

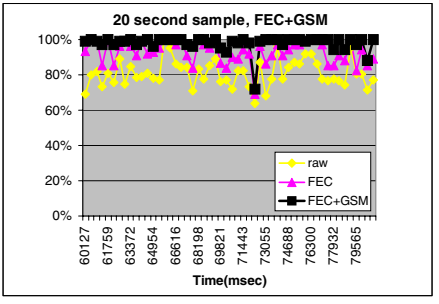
Finally, we conducted a set of experiments near the periphery of the wireless cell (Location 3), where large burst errors are more frequent. In these tests, we again combined the FEC filter with the GSM filter, but we configured the latter to use two values of θ, both 1 and 3, which enables it to correct longer burst



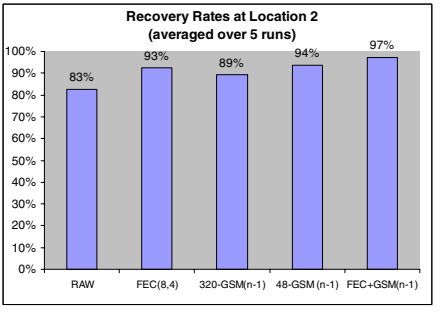
(a) FEC(8,4) alone



(b) GSM(n-1) alone



(c) FEC(8,4)+GSM(n-1)



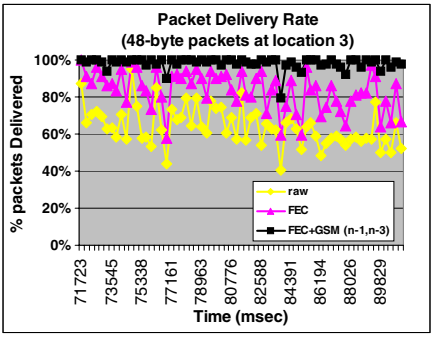
(d) performance comparison

Fig. 16. Effects of FEC, GSM filters and their composition.

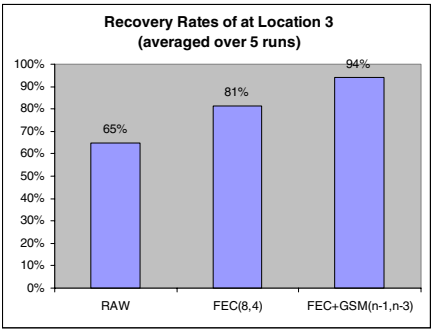
errors. Figure 17(a) shows a short sample trace. Despite the fact that the raw delivery rate sometimes falls below 50%, the combination of filters is extremely effective in recovering the lost data. Figure 17(b) shows the average results of 5 two-minute runs. At this location the raw delivery rate was only 65%, FEC alone raised the rate to 81%, and the GSM filter further improved the rate to 94%. The 13% GSM improvement over FEC alone compares with only a 4% improvement at Location 2. We conclude that, while both types of filters improve the quality of the audio channel, near the cell periphery, they are almost equally important. These results demonstrate the utility of being able to compose two different proxylets easily and dynamically within a single proxy framework.

6 Related Work

In recent years, numerous research groups have addressed the issue of adaptive middleware frameworks that can accommodate dynamic, heterogeneous infras-



(a) sample trace



(b) performance comparison

Fig. 17. Effects of FEC and GSM filters near the wireless cell periphery.

tructures. Examples include CEA [1], MOOSCo [2], BRAIN [3], Squirrel [4], Adapt [5], MOST [6], AspectIX [44], Limbo [45], MASH [10], TAO [11], MobiWare [12], MCF [13], QuO [14], MPA [9], Odyssey [15], and DaCapo++ [16] Rover [7], BARWAN [32], and Sync [46]. These projects have greatly improved the understanding of how middleware can accommodate device heterogeneity and dynamic network conditions, particularly in the area of adaptive communication protocols and services. Indeed, several of these projects address dynamic configuration of proxy and/or stream functionality. In the remainder of this section, we discuss four such projects and their relationship to RAPIDware.

In the MobiWare project [12], "mobile filters" can be dispatched to various nodes in the network, or to hosts, in order to achieve bandwidth conservation. Apparently, these filters are established only during handoff from one network to another. The detachable streams discussed herein could be used to extend this functionality so that filters could be reorganized at any time.

In the Adapt project at Lancaster [5], CORBA is extended to support open bindings, which enable manipulation and reconfiguration of communication paths through the use of object graphs. This powerful mechanism could be used directly to implement dynamically composable proxy services. In contrast to a CORBA-based design, we sought in this subproject to determine the minimal level of functionality needed to provide dynamic composition of communication stream components. We offer detachable Java I/O streams as a possible solution.

The Berkeley TranSend proxy is based on the TACC model [8], in which *workers* are the active components of the proxy. The TACC server enables workers to be chained together in a manner similar to Unix pipes. Details of the implementation are not available. However, the project focuses on proxies built atop highly available parallel workstation clusters, whereas RAPIDware proxies are intended to be lightweight, on-demand proxies established dynamically on one or more idle workstations available to the user.

The Stanford Mobile People Architecture (MPA) [9] is designed to support person-to-person reachability through the use of *personal proxies*. A key component of the personal proxy is the use of *conversion drivers*, which are configured dynamically to match the capabilities of the user's device and network. The RAPIDware project seeks to develop a general framework for the design of such software. In this case, the detachable stream mechanism and filter container class may provide a useful mechanism for composing such drivers and facilitating their dynamic loading and unloading from across the network.

Finally, we emphasize that this paper has described only a small part of the RAPIDware project. A given external event, such as a sudden decrease in quality on a wireless link, can affect not only communication protocols, but also middleware components associated with fault tolerance, security, and user interfaces. The overall goal of the RAPIDware project is to develop an integrated methodology for middleware adaptability that encompasses not only communication services, but also security policies, fault tolerance actions, and user interface reconfiguration. We will report developments in these areas in future papers.

7 Conclusions and Future Work

In this paper, we have described the use of a detachable Java I/O stream framework to support composition of proxy services. We reviewed the design of the framework, which enables proxylets to be dynamically inserted, deleted, and reordered. Addition and removal of proxy code is achieved without disturbing existing network connections. We demonstrated the use of the framework to support the instantiation and composition of two different audio FEC filters, one using block erasure codes and the other using the GSM 06.10 encoder, and we evaluated their performance on a WLAN testbed. The main contribution of this work is to show that independent proxylets can be composed and can cooperate synergistically, given the proper supporting proxy framework.

Our continuing work in this area addresses several issues: porting of additional proxies to the RAPIDware framework; development of a rules engine to characterize the "composability" of proxylets; and application of RAPIDware concepts to intrusion detection, fault tolerance, and user interfaces handoff. Given the increasing presence of wireless networks in homes and businesses, we envision application of the proposed techniques to improve performance of collaborative applications involving users who roam within a wireless environment.

Further Information. A number of related papers and technical reports of the Software Engineering and Network Systems Laboratory can be found at the following URL: `http://www.cse.msu.edu/sens`.

Acknowledgements. The authors would like to thank Arun Mani, Suraj Gaurav, Peng Ge, and Chiping Tang for their contributions to this work. This work was supported in part by the U.S. Department of the Navy, Office of Naval Research under Grant No. N00014-01-1-0744. This work was also supported in part by U.S. National Science Foundation grants CDA-9617310, NCR-9706285, CCR-9912407, and EIA-0000433.

References

1. Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O., Spiteri, M.: Generic support for distributed applications. IEEE Computer **33** (2000) 68–76
2. Miranda, H., Antunes, M., Rodrigues, L., Silva, A.R.: Group communication support for dependable multi-user object-oriented environments. In: SRDS Workshop on Dependable System Middleware and Group Communication (DSMGC 2000), Nürnberg, Germany (2000)
3. Burness, L., Kassler, A., Khengar, P., Kovacs, E., Mandato, D., Manner, J., Neureiter, G., Robles, T., Velayos, H.: The BRAIN quality of service architecture for adaptable services. In: Proceedings of the PIMRC 2000, London (2000)
4. Kramp, T., Koster, R.: A service-centered approach to QoS-supporting middleware (Work-in-Progress Paper). In: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England (1998)
5. Fitzpatrick, T., Blair, G., Coulson, G., Davies, N., Robin, P.: A software architecture for adaptive distributed multimedia applications. IEE Proceedings - Software **145** (1998) 163–171
6. Friday, A., Davies, N., Blair, G., Cheverst, K.: Developing adaptive applications: The MOST experience. Journal of Integrated Computer-Aided Engineering **6** (1999) 143–157
7. Joseph, A.D., Tauber, J.A., Kaashoek, M.F.: Mobile computing with the Rover toolkit. IEEE Transactions on Computers: Special issue on Mobile Computing **46** (1997)
8. Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A.: Adapting to network and client variation using active proxies: Lessons and perspectives. IEEE Personal Communications (1998)
9. Roussopoulos, M., Maniatis, P., Swierk, E., Lai, K., Appenzeller, G., Baker, M.: Person-level routing in the mobile people architecture. In: Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems, Boulder, Colorado (1999)
10. McCanne, S., Brewer, E., Katz, R., Rowe, L., Amir, E., Chawathe, Y., Coopersmith, A., Mayer-Patel, K., Raman, S., Schuett, A., Simpson, D., Swan, A., Tung, T., Wu, D., Smith, B.: Toward a common infrastructure for multimedia-networking middleware. In: Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97), St. Louis, Missouri. (1997)
11. Kuhns, F., O'Ryan, C., Schmidt, D.C., Othman, O., Parsons, J.: The design and performance of a pluggable protocols framework for object request broker middleware. In: Proceedings of the IFIP Sixth International Workshop on Protocols For High-Speed Networks (PfHSN '99), Salem, Massachusetts (1998)
12. Angin, O., Campbell, A.T., Kounavis, M.E., R.R.-F.M. Liao: The Mobiware toolkit: Programmable support for adaptive mobile networking. IEEE Personal Communications Magazine, Special Issue on Adapting to Network and Client Variability (1998)
13. Li, B., Nahrstedt, K.: A control-based middleware framework for quality of service adaptations. IEEE Journal of Selected Areas in Communications **17** (1999)
14. Vanegas, R., Zinky, J.A., Loyall, J.P., Karr, D.A., Schantz, R.E., Bakken, D.E.: QuO's runtime support for quality of service in distributed objects. In: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, England (1998)

15. Noble, B.D., Satyanarayanan, M.: Experience with adaptive mobile applications in Odyssey. Mobile Networks and Applications **4** (1999) 245–254
16. Stiller, B., Class, C., Waldvogel, M., Caronni, G., Bauer, D.: A flexible middleware for multimedia communication: Design implementation, and experience. IEEE Journal of Selected Areas in Communications **17** (1999) 1580–1598
17. Badrinath, B.R., Bakre, A., Marantz, R., Imielinski, T.: Handling mobile hosts: A case for indirect interaction. In: Proc. Fourth Workshop on Workstation Operating Systems, Rosario, Washington, IEEE (1993)
18. Zenel, B., Duchamp, D.: Intelligent communication filtering for limited bandwidth environments. In: Proc. Fifth Workshop on Hot Topics in Operating Systems, Rosario, Washington (1995)
19. Chen, L., Suda, T.: Designing mobile computing systems using distributed objects. IEEE Communications Magazine **35** (1997)
20. Chawathe, Y., Fink, S., McCanne, S., Brewer, E.: A proxy architecture for reliable multicast in heterogeneous environments. In: Proceedings of ACM Multimedia '98, Bristol, UK (1998)
21. McKinley, P.K., Mani, A.P.: An experimental study of adaptive forward error correction for wireless collaborative computing. In: Proceedings of the IEEE 2001 Symposium on Applications and the Internet (SAINT-01), San Diego-Mission Valley, California (2001)
22. Yang, L., Hofmann, M.: OPES architecture for rule processing and service execution. Internet Draft draft-yang-opes-rule-processing-service-execution-00.txt (2001)
23. McKinley, P.K., Padmanabhan, U.I.: Design of composable proxy filters for mobile computing. In: Proceedings of the Second International Workshop on Wireless Networks and Mobile Computing, Phoenix, Arizona (2001)
24. Rizzo, L.: Effective erasure codes for reliable computer communication protocols. ACM Computer Communication Review (1997)
25. Degener, J., Bormann, C.: The gsm 06.10 lossy speech compression library and its applications (2000) available at `http://kbs.cs.tu-berlin.de/~jutta/toast.html`.
26. McKinley, P.K., Malenfant, A.M., Arango, J.M.: Pavilion: A distributed middleware framework for collaborative web-based applications. In: Proceedings of the ACM SIGGROUP Conference on Supporting Group Work. (1999) 179–188
27. McKinley, P.K., Barrios, R.R., Malenfant, A.M.: Design and performance evaluation of a Java-based multicast browser tool. In: Proceedings of the 19th International Conference on Distributed Computing Systems, Austin, Texas (1999) 314–322
28. Arango, J., McKinley, P.K.: VGuide: Design and performance evaluation of a synchronous collaborative virtual reality application. In: Proceedings of the IEEE International Conference on Multimedia and Expo, New York (2000)
29. McKinley, P.K., Li, J.: Pocket Pavilion: Synchronous collaborative browsing for wireless handheld computers. In: Proceedings of the IEEE International Conference on Multimedia and Expo, New York (2000)
30. McKinley, P.K., Gaurav, S.: Experimental evaluation of forward error correction on multicast audio streams in wireless LANs. In: Proceedings of ACM Multimedia 2000, Los Angeles, California (2000) 416–418
31. Ge, P., McKinley, P.K.: Experimental evaluation of error control for video multicast over wireless LANs. In: Proceedings of the Third International Workshop on Multimedia Network Systems, Phoenix, Arizona (2001)

32. Katz, R. H., Brewer, E. A., et al.: The Bay Area Research Wireless Access Network (BARWAN). In: Proceedings Spring COMPCON Conference. (1996)
33. Xu, D., Li, B., Nahrstedt, K.: Qos-directed error control of video multicast in wireless networks. In: Proceedings of IEEE International Conference on Computer Communications and Networks. (1999)
34. McAuley, A.J.: Reliable broadband communications using burst erasure correcting code. In: Proceedings of ACM SIGCOMM. (1990) 287–306
35. Rizzo, L., Vicisano, L.: RMDP: An FEC-based reliable multicast protocol for wireless environments. ACM Mobile Computer and Communication Review **2** (1998)
36. Nonnenmacher, J., Biersack, E.W., Towsley, D.: Parity-based loss recovery for reliable multicast transmission. IEEE/ACM Transactions on Networking **6** (1998) 349–361
37. Huitema, C.: The case for packet level FEC. In: Proceedings of IFIP 5th International Workshop on Protocols for High-Speed Networks (PfHSN'96). (1996) 110–120 INRIA, Sophia Antipolis, France.
38. Gemmell, J., Schooler, E., Kermode, R.: A scalable multicast architecture for one-to-many telepresentations. In: Proceedings of IEEE International Conference on Multimedia Computing Systems. (1998) 128–139
39. Kermode, R.: Scoped Hybrid Automatic Repeat ReQuest with Forward Error Correction (SHARQFEC). In: Proceedings of ACM SIGCOMM. (1998) Vancouver, Canada.
40. Swarmcast: Release notes for Java FEC v0.5. `http://www.swarmcast.com` (2001)
41. Podolsky, M., Romer, C., McCanne, S.: Simulation of FEC-based error control for packet audio on the Internet. In: Proceedings of IEEE INFOCOM'96, San Francisco, California (1998)
42. Bolot, J.C., Vega-Garcia, A.: Control mechanisms for packet audio in Internet. In: Proceedings of IEEE INFOCOM'96, San Francisco, California (1996) 232–239
43. McKinley, P.K., Padmanabhan, U., Ancha, N.: Performance evaluation of audio FEC on wireless LANs. Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, in preparation (2001)
44. Hauck, F., Becker, U., Geier, M., Meier, E., Rastofer, U., Steckermeier, M.: AspectIX: A middleware for aspect-oriented programming. Technical Report TR-I4-98-06, Computer Science Department, Friedrich-Alexander-University, Erlangen-Nürnberg, Germany (1998)
45. Blair, G.S., Davies, N., Friday, A., Wade, S.P.: Quality of service support in a mobile environment: An approach based on tuple spaces. In: Proceedings of the 5th IFIP International Workshop on Quality of Service (IWQoS'97), New York (1997) 37–48
46. Munson, J., Dewan, P.: Sync: A system for mobile collaborative applications. IEEE Computer **30** (1997) 59–66

Thread Transparency in Information Flow Middleware*

Rainer Koster[1], Andrew P. Black[2], Jie Huang[2], Jonathan Walpole[2], and Calton Pu[3]

[1] University of Kaiserslautern, koster@informatik.uni-kl.de
[2] OGI School of Science and Engineering, Oregon Health and Science University, {black,jiehuang,walpole}@cse.ogi.edu
[3] Georgia Institute of Technology, calton@cc.gatech.edu

Abstract. Applications that process continuous information flows are challenging to write because the application programmer must deal with flow-specific concurrency and timing requirements, necessitating the explicit management of threads, synchronization, scheduling and timing. We believe that middleware can ease this burden, but middleware that supports control-flow centric interaction models such as remote method invocation does not match the structure of these applications. Indeed, it abstracts away from the very things that the information-flow centric programmer must control.
We are defining Infopipes as a high-level abstraction for information flow, and we are developing a middleware framework that supports this abstraction directly. Infopipes handle the complexities associated with control flow and multi-threading, relieving the programmer of this task. Starting from a high-level description of an information flow configuration, the framework determines which parts of a pipeline require separate threads or coroutines, and handles synchronization transparently to the application programmer. The framework also gives the programmer the freedom to write or reuse components in a passive style, even though the configuration will actually require the use of a thread or coroutine. Conversely, it is possible to write a component using a thread and know that the thread will be eliminated if it is not needed in a pipeline. This allows the most appropriate programming model to be chosen for a given task, and existing code to be reused irrespective of its activity model.

1 Introduction

The benefit of middleware platforms is that they handle application-independent problems transparently to the programmer and hide underlying complexity. CORBA or RPC, for instance, provide location transparency by hiding message passing and marshalling. Hiding of complexity relieves programmers from

* This work is partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, by NFS award CDA-9703218, and by Intel.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 121–140, 2001.

tedious tasks and allows them to focus on the important aspects of their applications.

The way that a middleware platform can hide complexity without hiding power is to provide higher-level abstractions that are appropriate for the supported class of applications. In order to choose a suitable abstraction, it is necessary to make some assumptions about the functionality that typical applications require. For example, a common abstraction provided by current middleware is the client-server architecture and request-response interaction, where control flows to the server and back to the client.

However, this model is inappropriate for an emerging class of information-flow applications that pass continuous streams of data among producers and consumers. Building these applications on existing middleware requires programmers to specify control-flow behaviors, which are not key aspects of the application. Moreover, existing middleware has inadequate abstractions for specifying data-flow behaviors, including quality of service and timing, which *are* key aspects of the application.

We propose a new middleware platform for information-flow applications that is based on a producer-consumer architectural model and the Infopipe abstraction. Infopipes simplify the task of building distributed streaming applications by providing basic components such as pipes, filters, buffers, and pumps [2, 28]. Each component specifies the properties of the flows that it can support, including data formats and QoS parameters. When stages of a pipeline are connected, flow properties for the composite can be derived, facilitating the composition of larger building blocks and the construction of incremental pipelines.

The need for concurrently active pipeline stages introduces significant complexity in the area of thread management that can be hidden in the middleware. Hence, our platform frees the programmer from the need to deal with thread creation, destruction, and synchronization. Moreover, the actual control flow is managed by the middleware and is decoupled from the way pipeline components are implemented, be they active or passive objects. We call this approach *thread transparency*. It simplifies programs and allows reuse of infopipe components. In the same way that RPC systems automatically generate code for parameter marshalling and message handling, our middleware handles thread management and generates glue code that allows Infopipe components to be reused in different activity contexts.

Section 2 describes the Infopipe middleware platform we are developing. Thread transparency is discussed in Section 3. Section 4 describes the current implementation. Related work is summarized in Section 5 before the conclusions in Section 6.

2 Infopipe Middleware

The Infopipe abstraction has emerged from our experience building continuous media applications [6, 36, 16, 13]. Currently we are building a middleware

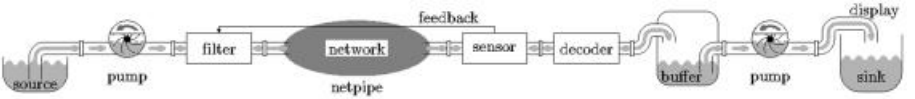


Fig. 1. Infopipe example

framework in C++ based on these concepts. On top of this platform we are reimplementing our video pipelines to facilitate further experimentation.

2.1 Overview

Infopipes let us build information flow pipelines from pre-defined components in a way that is analogous to the way that a plumber builds a water flow system from off-the-shelf parts.

The most common components have one input port and one output port. Such pipes can *transport* information, *filter* certain information items, or *transform* the information. *Buffers* provide temporary storage and remove rate fluctuations. *Pumps* are used to keep the information flowing. Corresponding to these roles each port has a appropriate *polaritiy*. Pumps have two ports with *positive* polarity, one pulling items from upstream and one pushing them downstream. Buffers, in contrast, have two *negative* ports being pulled from or pushed into. Filters and transformers have two ports of opposite polarity [1, 2]. *Sources* and *sinks* have only one port, which can be either positive or negative.

More complex components have more ports. Examples are *tees* for splitting and merging information flows. Splitting includes splitting an information item into parts that are sent different ways, copying items to each output (multicast), and selecting an output for each item (routing). Merge tees can combine items from different sources into one item or pass on information to the output in the order in which it arrives at any input.

In combining components of a pipeline it is important to check the compatibility of supported flows and to evaluate the characteristics of the composite Infopipe. Each basic or composite Infopipe has a *Typespec* that describes the flows that it supports. Typespecs provide information about supported formats of data items, interaction properties such as the capability of operating in push or pull mode, and ranges of QoS parameters that can be handled.

Transport protocols can be integrated into the Infopipe framework by encapsulating them as *netpipes*. These netpipes support plain data flows and may manage low-level properties such as bandwidth and latency. Marshalling filters on either side translate the raw data flow to and from a higher-level information flow. These components also encapsulate the mapping of QoS properties, which is described in more detail in Section 2.4.

In building an Infopipe an application developer needs to combine appropriate filters, buffers, pumps, network pipes, feedback sensors and actuators as well as control components. To facilitate this task, our framework provides a

set of basic components to control the timing and a feedback toolkit for adaptation control [8]. Components for processing specific types of flow need to be developed by application programmers, but can easily be reused in various applications. For instance, developers of video on demand, video conferencing, and surveillance tools can all use any available video codec components.

Figure 1 shows a simple video pipeline from a source producing compressed data to a display. At the producer side frames are pumped through a filter into a netpipe encapsulating a best-effort transport protocol. The filter drops frames when the network is congested. The dropping is controlled by a feedback mechanism using a sensor on the consumer side. This lets us control which data is dropped rather than suffering arbitrary dropping in the network. After decoding the frames, they are buffered to reduce jitter. A second pump controlling the output timing finally releases the frames to the display.

2.2 Interaction

With respect to polarity, there are three classes of components:

- *Positive components* have only positive ports and cause information to flow in the pipeline. Pumps and active sources and sinks belong to this class.
- *Negative components* have only negative ports. Buffers and passive sources and sinks belong to this class.
- *Neutral components* have positive and negative ports. They do not initiate any activity but may pass it on to other components. Common components with one positive and one negative port belong to this class.

The processing of the information items is driven by a thread that originates from a positive component as shown in Figure 2. Negative and neutral objects can be implemented as objects with methods that are called through a negative port and may call out through a positive port, making inter-object communication particularly efficient. Because pumps originate the threads, they regulate the timing of the data flow and can themselves be controlled by timers or feedback mechanisms. Each thread is responsible for calling through all the neutral pipeline stages as far as the next negative components up- or downstream. Hence, Pumps encapsulate the interaction with the underlying scheduler.

Besides exchanging data items, Infopipe components can exchange control messages. These messages are used to represent local interaction between adjacent components as well as global broadcast events. As an example of local

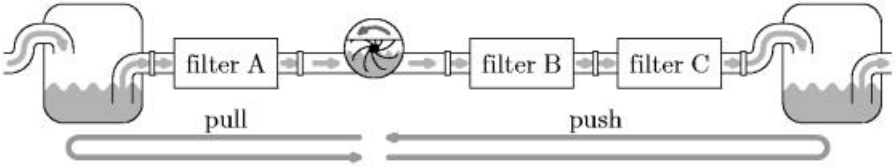


Fig. 2. Polarity

interaction, consider an MPEG-decoder that passes on decoded video frames but must still keep them as reference frames. Communication between the decoder and downstream components must be used to determine when the shared frames can be deleted. Another case is a video resizing component that needs to be informed by the video display whenever the user changes the window size. Control interaction between remote components of a pipeline includes communication between feedback sensors, controllers, and actuators. Other events such as user commands to start or stop playing need to be broadcast to potentially many components. While control events to adjacent components can easily be sent directly, we use a simple event service to facilitate global distribution of control events.

The current approach to handling control events is based on the assumption that handling these events does not require much time. Hence, there is no explicit control of timing and buffering of these events and their handlers are executed with higher priority than potentially long-running data processing.

2.3 Infopipe Typespecs

The ability to construct composite pipes from simpler components is an important feature of the Infopipe platform. Automatic inference of flow properties, glue code for joining different types of components, and automatic allocation of threads help the application programmer and simplify binding protocols for setting up an Infopipe.

A Typespec describes the properties of an information flow. Typespecs are extensible and new properties can be added as needed. Undefined properties may be interpreted as meaning either *don't know* or *don't care* as discussed below. The following list describes some parts of a Typespec.

- The *item type* describes the format of the information items and the flow.
- The *polarity* of ports in the information the flow determines whether items are pushed or pulled. Polarity is represented in the Typespec by assigning each port a positive or negative polarity. A positive out-port will make calls to the `push` method of the downstream components, while a negative out-port has the ability to receive a `pull`. Correspondingly, a positive in-port will make calls to `pull`, while a negative in-port represents the willingness to receive a `push`. With this representation, ports with opposite polarity may be connected, but an attempt to connect two ports with the same polarity is an error.
  Some components do not have a fixed polarity. For example, filters can operate in push or pull mode, as can chains of filters. These components are given the polymorphic polarity $\alpha \rightarrow \overline{\alpha}$. When one port is connected to a port with a fixed polarity, the other port of the filter or filter chain acquires the opposite "induced" polarity [2, 9].
- A third property specifies the *blocking behavior* if an operation cannot be performed immediately. For instance, if a buffer is full, the push operation can either be blocked or can drop the pushed item. Likewise, if a buffer is empty, a pull operation can either be blocked or return a nil item.

- While push and pull are the only data transmission functions, *control events between connected components* may be needed to exchange meta-data of the flow. The capability of components to send or react to these control events is included in the Typespec to ensure that the resulting pipeline is operational.
- *QoS parameters* may include video frame rates and sizes, latency, or jitter. While processing a flow with specific values for these parameters requires elaborate resource management and binding protocols, QoS parameters may provide valuable hints to the rest of the pipeline even if guarantees are not available. For instance, feedback mechanisms can trade one quality dimension for another, for instance, trade frame rate for timely delivery, which again can be reflected in the Typespec.
- For distributed pipelines, the *location* indicates that a flow, from a source or to a sink, for instance, must be produced or consumed at a particular node.

Properties can originate from sources, sinks, and intermediate pipes. Sources typically supply one or more possible data formats along with information on the achievable QoS. Likewise, sinks support certain data formats and ranges of QoS parameters reflecting user preferences. Hence, source properties indicate what can be produced, sink properties indicate what the user likes to consume.

If for any stage in a pipeline a Typespec for an input or output port is given, Typespecs for other ports can be derived from that information. The derived Typespecs may support only a subset of the flow types in the given Typespec, reflecting restrictions imposed by that stage. These restrictions might originate because the stage supports only pull-interaction, fewer data types, or a smaller range for a QoS parameter. Moreover, stages can add or update properties.

Because of this incremental nature of Typespecs, we do not associate a fixed Typespec with each component, but let each pipeline component transform a Typespec on each port to Typespecs on its other ports. That is, the component analyzes the information about the flow at one port and derives information about flows at other ports. These Typespec transformations are the basis for dynamic type-checking and evaluation of possible compositions.

2.4 Distribution

Any single protocol built into a middleware platform is inadequate for remote transmission of information flows with a variety of QoS requirements. However, different transport protocols, can be easily integrated into the Infopipe framework as *netpipes*. These netpipes support plain data flows and may manage low-level properties such as bandwidth and latency. Marshalling filters on either side translate the raw data flow to a higher-level information flow and vice-versa. These components also encapsulate the QoS mapping, translating between netpipe properties and flow-specific properties.

In addition to netpipes, the Infopipe platform provides protocols and factories for the creation of remote Infopipe components. Remote Typespec queries also require a middleware protocol as well as a mechanism for property marshalling. The location itself can be integrated in the type checking by adding a location

Fig. 3. Distributed Infopipe

property that is changed only by netpipes. Finally, control events are delivered to remote components through the platform.

3 Transparent Thread Management

Different timing requirements and computation times at different stages of a pipeline require multiple asynchronous threads. Unfortunately, handling multithreading and synchronization mechanisms is difficult for many programmers and frequently leads to errors [26,33]. However, because the interaction between components in an Infopipe framework is restricted to well known interfaces, it is possible to hide the complexity of low-level concurrency control in the middleware platform. This is similar to the way in which RPC or CORBA hide the complexity of low-level remote communication from the programmer.

While some aspects such as timing behavior need to be exposed to the programmer, as described in Section 3.1, other aspects such as scheduler interfaces, inter-thread synchronization, wrappers and the adaptation of implementation styles can largely be hidden in the middleware platform, as described in the following three subsections.

3.1 Timing Control and Scheduling

Pumps encapsulate the timing control of the data stream. Each pump has a thread that operates the pipeline as far as the next negative component up- and downstream. Interaction with the underlying scheduler is also implemented in pumps. At setup, they can make reservations, if supported, according to estimated or worst case execution time of the pipeline stages they run. Moreover, they can select and adjust thread scheduling parameters as the pipeline runs.

From our experience building multimedia pipelines we can identify at least two classes of pumps. *Clock-driven pumps* typically operate at a constant rate and are often used with passive sinks and sources. Both pumps in Figure 1 belong to this category. Audio output devices that have their own timing control can be implemented as clock-driven active sinks. *Environment-sensitive pumps* adjust their speed according to the state of other pipeline components. The simplest version does not limit its rate at all and relies on other components to block the thread when a buffer is full or empty. More elaborate approaches adjust CPU allocations among pipeline stages according to feedback from buffer fill levels [31]. Another kind of environment-sensitive pump is used on the producer node of a

distributed pipeline [6, 36]. Its speed is adjusted by a feedback mechanism to compensate for clock drift and variation in network latency between producer and consumer.

The choice of the right pump depends on application requirements as well as the capabilities of the scheduler. While it is not yet clear to what extent pump selection and placement can be automated, pumps do hide thread creation and scheduling mechanisms. The programmer does not need to deal with these low-level details but can choose timing and scheduling policies by choosing pumps and by setting appropriate parameters.

If existing pumps do not provide the required functionality, it can be cleanly added by implementing new pumps. While a pump developer needs to deal with threads and scheduling, the pump encapsulates threading mechanisms similarly to the way that a decoder encapsulates compression mechanisms. In both cases, the complexity is hidden from the application programmers who use the new components.

3.2 Synchronization

Infopipe components need to process information (possibly from different ports) and control events. While information items and control events may arrive in any order, the middleware ensures synchronized access to shared data in its high-level communication mechanisms. The component developer does not need to deal with inter-thread synchronization explicitly, but just provides data processing and event handling functions. Hence, inter-thread synchronization is based on passing on data items and control events rather than on more error-prone primitive mechanisms such as locks and semaphores.

The pipeline components are implemented as monitors, also known as synchronized objects [4]: each component may contain at most one active thread at any time. However, we allow threads to be preempted because running functions such as video decoders non-preemptively can introduce unacceptable delay. A data processing function of one component is never called before the previous invocation completes or while a control event handler of the same component is running. Control events that arrive while data processing is in progress are queued and delivered as soon as the data processing is done. Note, however, that control events can be delivered while threads are blocked in a `push` or `pull`. Hence, the programmer needs to make sure that the component is in a consistent state with respect to control handlers when these operations are called.

3.3 Implementation Styles in Pipeline Components

In this section we discuss several styles with respect to activity that can be used in implementing pipeline components. The main distinction is between active objects that have an associated thread and passive objects that are called by external threads [4]. To make a clear distinction between the polarity of a component as introduced in Section 2.2 and its implementation style, we call implementations as active objects *thread-style components* and implementations

as passive objects *function-style components*. We focus on neutral components with one input and one output port, which are most common. As a simple example we use a defragmenter that combines two data items into one. The actual merging is performed by the function `y=assemble(x1,x2)`.

The middleware platform assumes components such as filters to be neutral, having a positive and a negative port. The external interface is an `item pull()` operation that can be called by downstream components and `void push(item)` operation that can be called by upstream components. Which of these is used in a particular pipeline component depends on the position of the component relative to pumps and buffers. Components between buffer and pump operate in pull mode, components between pump and buffer in push mode, as shown in Figure 2.

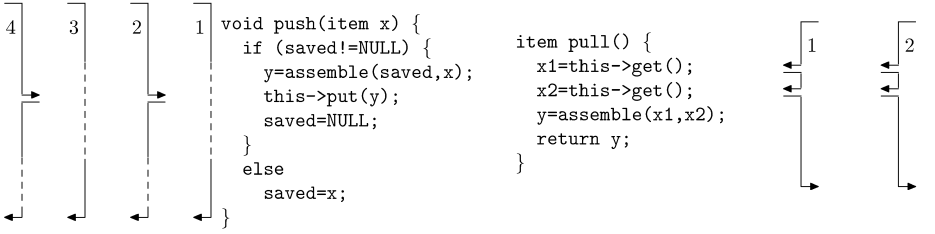


Fig. 4. Function-style defragmenter

To implement these components in function style, `push` or `pull` must be provided by the programmer. By convention, the programmer does not directly call `push` or `pull` methods on other components. He instead uses `put` and `get` methods, which are inherited from a base class provided by the middleware platform. In this case, the implementation of `put` and `get` is as follows:

```
void put(item x) {next->push(x);}
item get() {return prev->pull();}
```

For the defragmenter example, the `push` and `pull` methods are shown in Figure 4. Each numbered group of arrows shows the control flow for one call to the method that it annotates. In Figure 4b, each invocation of `pull` travels all the way through the code triggering two `get` calls and, hence, two `pull` calls to the upstream pipeline component. For `push` in Figure 4a every other call (2 and 4) causes a `put` and, hence, a downstream `push`. If no output item can be produced the call returns directly. This example shows that the `pull` operation for the defragmenter can be implemented more easily than `push`. The latter requires the programmer to explicitly maintain state between two invocations, which is done in this example using the variable `saved`. Conversely, for a fragmenter, `push` would be the simpler operation.

There are several reasons for integrating thread-style components that are written as active objects into this framework. One reason is the reuse of code from older pipeline implementations that used an active object model or implemented each stage as a process. Another reason is the flexibility that thread-style implementations provide. The programmer can freely mix statements for sending and receiving data items as is most convenient for a given component. Finally, more programmers are familiar with the thread-style model than with the function-style model.

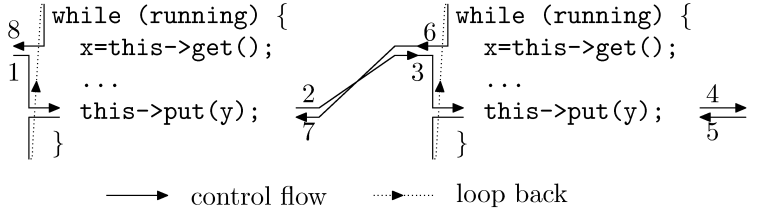
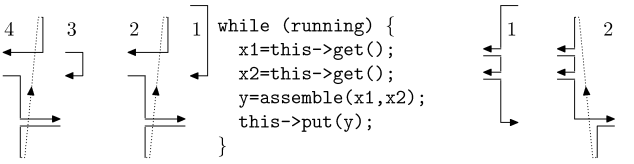


Fig. 5. Synchronous threads

The way to give these thread-style components the facade of a neutral component is to use coroutines, that is, threads interacting synchronously in such a way that they provide suspendable control flow but are not a unit of scheduling [5]. The communication mechanism between the threads does not buffer data; instead the activity travels with the data. All but one of the coroutines in a given set are blocked at any time. Figure 5 gives an example of two coroutines interacting in push mode. An item is pushed into the first component deblocking it from a `this->get` call (1). The component then processes the data and calls `this->put`, which passes the item to the next component (2), which deblocks from its `get` (3). It again does some processing and a `put` (4). When `put` returns (5), the control flow loops back to the `get` call. This blocks the second component (6) and unblocks the first component from its `put` (7). Finally the control flow reaches a `get` call again and returns to the upstream component (8).

The coroutine behavior described above is implemented by inheriting different `put` and `get` methods from appropriate superclasses. Consider a pipeline running in push mode. If the target of a `put` is a function-style component providing a `push`-method, then `put` can simply call `next->push`. However, if the target is a thread-style component, then `put` performs a switch to the coroutine of the target component, which is blocked in its `get` method. Pull mode is handled analogously.

Figure 6 shows an thread-style implementation of the defragmenter example. Here again, each numbered group of arrows denotes the control flow for one `push` call (in Figure 6a) or one `pull` call (in Figure 6b) to the component. When operating in push mode, upstream `get` calls block the defragmenter and each invocation executes from `get` to `get`. As an exception, the first `push` call invokes



a) push-mode b) pull-mode

Fig. 6. Thread-style defragmenter

```
while (running) {
  x=this->pull();
  this->put(x):
  }
```

a) Push-mode wrapper for `pull`

```
while (running) {
  x=this->get();
  this->push(x):
  }
```

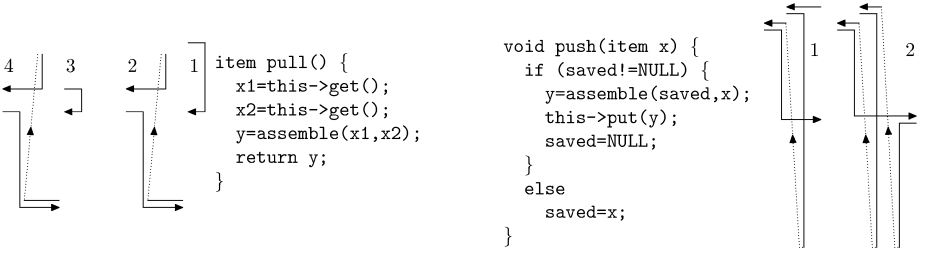
b) Push-mode wrapper for `push`

Fig. 7. Coroutine wrappers

the main function of the component, enters its loop, and satisfies the first call to `get`. Again, the pull mode works analogously.

The function-style implementation shown in Figure 4 has a major drawback. Components have to provide both a `push` and a `pull` operation that implement the same functionality. Alternatively, components could provide only one of these operations, but then could be used in either pull or push mode only, making building the pipeline more difficult. These restrictions can be avoided with middleware support that allows `push` functions to be used in pull mode and vice-versa. Our Infopipe middleware generates glue code for this purpose and converts the functions into coroutines as illustrated in Figure 7. Figure 8 shows the resulting control flow for the defragmenter example.

Note that the information flow is the same in Figures 4, 6, and 8. The number of incoming and outgoing arrows is the same for each invocation and for all three



a) push-mode b) pull-mode

Fig. 8. Function-style defragmenters, used other way

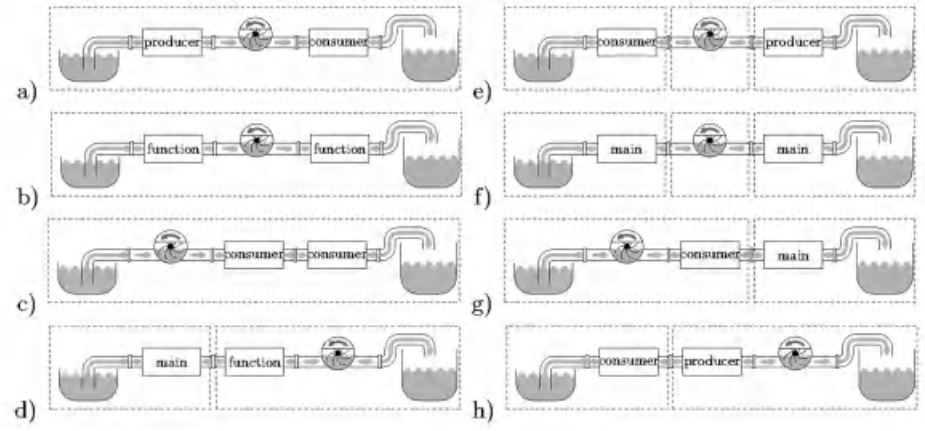


Fig. 9. Pipelines and coroutines

implementations. Every other push triggers a downstream push in Part a of the figure and every pull triggers two upstream pulls in Part b.

In the common case of a component that produces exactly one output for each input, an additional, particularly simple, implementation is possible. All the programmer needs to do is provide a conversion function: `item convert(item x)`. While the functionality is restricted by this one-to-one mapping, this type of component can easily be used in pull as well as push mode.

The `push` and `pull` methods are provided by the middleware:

```
void push(item x) {next->push(convert(x));}
item pull() {return convert(prev->pull());}
```

While we have used a defragmenter as an example, the different ways of implementing components that we have described also apply to fragmenters, decoders, filters, and transformers. By supporting all these styles, we provide flexibility in developing and reusing components, but for efficiency it is nonetheless important to avoid context switches and use direct calls whenever possible. Hence, the framework detects which components can share a thread and for which ones additional coroutines are needed. Figure 9 shows several pipelines between a passive source and a passive sink with the associated threads and coroutines depicted as dashed boxes. The same coroutine boundaries would apply to pipeline sections between two negative components. Altogether, there are four styles of neutral components. Thread-style implementations provide a thread-like main function. Function-style components are consumers implementing `push`, producers implementing `pull`, or are based on a conversion function. In push mode, consumers and conversion functions are called directly, and in pull mode producers and conversion functions are called directly. Otherwise, a coroutine is required. In each case, all threads operate synchronously as one coroutine set and the pump controls timing and scheduling in all components.

3.4 Complex Components

The behavior of components with more than two ports is more complex. Not all styles of implementation can be supported for all components. Sometimes the functionality of the component makes a particular style inappropriate. To see this, consider a switch with one in-port and two out-ports. Incoming packets are routed to one of the out-ports depending on the data in the packet. Now consider this switch in pull mode, that is, packets are pulled from either out-port. A pull request arrives at out-port 1 triggering an upstream pull-request at the in-port. Suppose that the incoming packet is routed to out-port 2. Now there is a pending call without a reply packet and a packet nobody asked for. Suspending the call would require buffering potentially many requests on out-port 1 and buffering packets at out-port 2 until all packets at out-port 2 are pulled. This approach leads to unpredictable implicit buffering behavior and complex dependencies. To avoid these problems the Infopipe framework generally allows only one negative port in a non-buffering component. However, there are exceptions. For instance a different type of switch may route the packet not according to the value of the packet, but based on the activity. A pull on either out-port triggers an upstream pull and returns the item to the caller. In this case, the out-ports must both be negative and the in-port must be positive. This component could not work in push mode.

4 Implementation

The development of the Infopipe middleware described in Section 2 is still in progress. We have implemented the activity-related functionality discussed in the previous section and part of the Typespec processing. A local video player has been built on top of it.

The platform is built on a message-based user-level thread package [12,13,15] implemented in C++. Each thread consists of a code function and a queue for incoming messages. Unlike conventional threads, the code function is not called at thread creation time but each time a message is received. After processing a message, the code function returns, but the thread is terminated only when the return value is -1. In this way, code functions resemble event handlers, but may be suspended waiting for other messages or may be preempted. Inter-thread communication is performed by sending messages to other threads, either synchronously if there remains nothing to do for a thread until a reply is received, or asynchronously whenever a reply is not needed immediately, or no reply is required at all. Network packets and signals from the operating system are mapped to messages by the platform, allowing all kinds of events to be handled by a uniform message interface.

The Infopipe platform creates a thread for each pump. If there is no need for coroutines in the section of a pipeline that is controlled by a particular pump, the thread calls the `pull` methods of all components upstream of the pump, then calls `push` with the returned item on the components downstream of the pump, and finally returns to the pump, which schedules the next pull. This is

the situation in configurations a), b), and c) in Figure 9. For configurations d), g), and h) there are two coroutines and for configurations e) and f) there are three coroutines associated with the pump. If such coroutines are needed, each of them is implemented by an additional thread of the underlying thread package. Their synchronous interaction is implemented on top of it.

Infopipe `push` and `pull` calls between coroutines and control events are mapped to asynchronous inter-thread messages. Although `push` and `pull` are synchronous to the Infopipe programmer, synchronous messages cannot be used, because then the thread would not be responsive to control events. Instead, the thread blocks waiting for either a control message or the data reply message. A control event is dispatched to the appropriate handler and then the thread blocks again. After receiving the reply message the code function of the thread is resumed. In this way the middleware implementation establishes synchronous communication of data items between coroutines, while control events can be handled even if the component is blocked in a `pull` or `push`.

The thread package supports scheduling by attaching priorities to threads as well as by attaching constraints to messages. In the latter case, the effective priority of a thread is derived by the scheduler from the constraint of the message that the thread is currently processing or, if the thread is waiting for the CPU, on the constraint of the first message in its queue. If no constraint is specified for the message, a static priority is used that is assigned to the thread when it is created. The package provides a priority inheritance scheme that modifies this behavior as necessary to avoid priority inversion, for instance, when a thread receives a message with a higher priority than that of the message it is currently processing.

In the Infopipe framework, message constraints are assigned by the pumps. Messages between coroutines inherit the constraint from the message received by the sending component, applying the constraint to the entire coroutine set. In this way, the pump controls the scheduling in its part of the pipeline across coroutine boundaries.

While other systems for concurrency such as μC++ provide coroutines directly [5], this message-based approach facilitates the processing of control events and the scheduling of concurrent activities according to different timing constraints [13].

The component developer indicates his choice of implementation style by inheriting from the appropriate base class and by overriding a `run` method for a thread-style component, a `push` method for a consumer, a `pull` method for a producer, and a `convert` method for a function-style component. Additionally, a handler for control events needs to be provided. For pipeline components that change the Typespec of flows the inherited implementation of the type query must be overridden.

Pipelines are configured by a high-level C++ interface. Composition and start of a simple video player could be implemented by

```
mpeg_file source("test.mpg");
mpeg_decoder decode;
```

```
clocked_pump pump(30);  // 30 Hz
video_display sink;
source>>decode>>pump>>sink;
main_channel.send_event(START);
```

If the components were not compatible, the composition operator >> would throw an exception. This simple example does not compensate for jitter caused by varying decoding times. The last line starts the pipeline by broadcasting a control event, to which the pump reacts. In a video player for complex presentations consisting of several streams, an additional control component would register for global control events such as START and in response dispatch START events to individual pipelines at the start time of their stream relative to the start time of the overall presentation.

A context switch between the user level threads takes about 1 μs; the time for a mere function call is two orders of magnitude shorter. Hence, the approach that we have presented in which threads and coroutines are introduced only when necessary is mostly important for pipelines that handle many control events or many small data items, such as a MIDI mixer. For these applications, and if kernel-level threads are used, allocating a thread for each pipeline component would introduce a significant context switching overhead.

5 Related Work

Some related work aims at integrating streaming services with middleware platforms based on remote method invocations such as CORBA. The CORBA telecoms specification [25] defines stream management interfaces, but not the data transmission. Only extensions to CORBA such as TAO's pluggable protocol framework [17] allow the efficient implementation of audio and video applications [24].

One approach for adding quality of service support to CORBA has been introduced by the QuO architecture [35]. It complements the IDL descriptions with specifications of QoS parameters and adaptive behavior in domain specific languages. From these declarative descriptions so called delegates are generated and linked to the client application in a similar way to that in which stubs are generated from an IDL. QuO, however, has not been built for streaming applications and interaction is based on remote method invocations.

A model for specifying flow quality and interfaces has been proposed as part of the MULTE project [29]. Compatibility and conformance rules are used for type checking and stream binding. This model is more formal, but less flexible, than our current approach using Typespecs.

Similarly to Infopipes, the Regis environment [20] separates the configuration of distributed programs from the implementation of the program components. The Darwin language is used to describe and verify the configurations. Components, which execute as threads or processes, are implemented in C++ with headers generated from Darwin declarations. While the Infopipe implementation

described here also uses C++ for pipeline setup, there are plans for developing an Infopipe Composition and Restructuring Microlanguage [28].

Open middleware platforms and communications frameworks such as OpenORB [3] and Bossa Nova [14] offer a flexible infrastructure that supports QoS-aware composition and reflection. While these frameworks do not provide specific streaming support, they can serve as a basis for building information flow middleware.

Event-based middleware such as Echo [7, 11] provides a type-safe and efficient way of communicating data and control information in a distributed and heterogeneous environment. A higher-level Infopipe layer can also be built on top of these platforms.

Ensemble [34] and Da CaPo [27] are protocol frameworks that support the composition and reconfiguration of protocol stacks from modules. Both provide mechanisms to check the usability of configurations and use heuristics to build the stacks. Unlike these frameworks for local protocols, Infopipes use a uniform abstraction for handling information flow from source to sink, possibly across several network nodes.

The *x-kernel* protocol architecture [10] associates processes with messages rather than protocols. In this way, messages can be shepherded through the entire protocol stack without incurring any context switch overhead. We support this thread-per-packet approach for Infopipe components that are implemented in a way that allows direct method calls. Alternatively, developers may choose to program in an thread-like style if this simplifies the program structure.

The Scout operating system [23] generalizes from the x-kernel by combining linear flows of data into *paths*. Paths provide an abstraction to which the invariants associated with the flow can be attached. These invariants represent information that is true of the path as a whole, but which may not be apparent to any particular component acting only on local information. This idea — providing an abstraction that can be used to transmit non-local information — is applicable to many aspects of information flows, and is one of the principles that Infopipes seek to exploit. For instance, in Scout paths are the unit of scheduling, and a path, representing all of the processing steps along its length, makes information about all of those steps available to the scheduler. This is similar to the way that a section of an Infopipe between two passive components is scheduled by one pump.

Structuring data processing applications as components that run asynchronously and communicate by passing on streams of data items is a common pattern in concurrent programming [e.g. 18]. *Flow-Based Programming* applies this concept to the development of business applications [22]. While the flow-based structure is well-suited for building multimedia applications, it must be supplemented by support for timing requirements. Besides integrating this timing control via pumps and buffers, Infopipes facilitate component development and pipeline setup by providing a framework for communication and threading.

The VuSystem [19] is a multimedia platform that has several similarities to Infopipes: applications are structured as pipeline components processing infor-

mation flows, there are interfaces for flow and control communication, and no particular real-time support from the operating system is needed. VuSystem, however, is single-threaded and timing and flow are controlled by the data processing components themselves. Infopipes, in contrast, support multiple threads, preemptive scheduling, and a choice of several programming styles for components and more elaborate consistency checks for pipeline setup.

For constructing streaming applications from components, there are also free and commercial frameworks [21, 32, 30]. GStreamer and DirectShow support setup of local pipelines without timing and QoS control. They provide services to automatically configure components for the conversion of data formats. GStreamer supports component function-style push and thread-style implementations, but does not have pumps to encapsulate timing control. RealSystem is a distributed framework that allows file source components to be used in servers as well as in local clients. The actual transmission is hardcoded into the RealServer and may be configured by adaptation rules.

6 Conclusions

Infopipes provide a framework for building information flow pipelines from components. This abstraction extends uniformly from source to sink. The application controls the setup of the pipeline, configuring its behavior based on QoS parameters and other properties exposed by the components.

The Infopipe platform manages concurrent activity in the pipeline and encapsulates synchronization in high-level communication mechanisms. To specify scheduling policies the application programmer needs only to choose appropriate pumps, which interact with the underlying scheduler and control the actual timing. Neutral components such as filters can be implemenented as active objects, passive consumers, passive producers, or conversion functions, whichever is most suitable for a given task, and existing code can be reused regardless of its implementation style with respect to threading. The Infopipe platform transparently handles creation of and communication between threads and coroutines. This is very much like the way in which CORBA transparently handles marshalling and remote communication.

We have implemented most middleware functionality for local pipelines. Using this platform, we have built several video processing components and configured a simple video player application. The supported functionality is being extended by distributed setup, resource reservations, and feedback mechanisms.

Acknowledgements. The Infopipe video player is based on a video pipeline built by Ashvin Goel and Charles 'Buck' Krasic using coroutines and POSIX threads. We would like to thank the anonymous reviewers for their help in improving this paper.

References

1. A. P. Black. An asymmetric stream communication system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 4–10, October 1983.
2. A. P. Black, J. Huang, and J. Walpole. Reifying communication at the application level. In *Proceedings of the International Workshop on Multimedia Middleware*. ACM, October 2001. Also available as OGI technical report CSE-01-006.
3. G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next-generation middleware. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 191–206. IFIP, September 1998.
4. J.-P. Briot, R. Guearraoui, and K.-P. Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3), September 1998.
5. P. Buhr, G. Ditchfield, R. Stroobosscher, B. Younger, and C. Zarnke. μC++: Concurrency in the object oriented language C++. *Software – Practice and Experience*, 20(2):137–172, February 1992.
6. S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A distributed real-time MPEG video audio player. In *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, volume 1018 of *Lecture Notes in Computer Science*, pages 142–153. Springer Verlag, April 1995.
7. G. Eisenhauer, F. Bustamante, and K. Schwan. Event services for high performance computing. In *International Conference on High Performance Distributed Computing (HPDC)*, August 2000.
8. A. Goel, D. Steere, C. Pu, and J. Walpole. Adaptive resource management via modular feedback control. Technical Report CSE-99-003, Oregon Graduate Institute, January 1999.
9. J. Huang, A. P. Black, J. Walpole, and C. Pu. Infopipes – an abstraction for information flow. In *ECOOP 2001 Workshop on The Next 700 Distributed Object Systems*, June 2001. Also available as OGI technical report CSE-01-007.
10. N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.
11. C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance computing. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.
12. R. Koster and T. Kramp. A multithreading platform for multimedia applications. In *Proceedings of Multimedia Computing and Networking 2001*. SPIE, January 2001.
13. R. Koster and T. Kramp. Using message-based threading for multimedia applications. In *Proceedings of the International Conference on Multimedia and Expo (ICME)*. IEEE, August 2001.
14. T. Kramp and G. Coulson. The design of a flexible communications framework for next-generation middleware. In *Proceedings of the Second International Symposium on Distributed Objects and Applications (DOA)*. IEEE, September 2000.
15. T. Kramp and R. Koster. Flexible event-based threading for QoS-supporting middleware. In *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*. IFIP, July 1999.
16. C. Krasic and J. Walpole. QoS scalability for streamed media delivery. Technical Report CSE-99-011, Oregon Graduate Institute, September 1999.

17. F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for object request broker middleware. In *Proceedings of the sixth IFIP International Workshop on Protocols for High-Speed Networks (PfHSN)*, August 1999.
18. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.
19. C. J. Lindblad and D. L. Tennenhouse. The vusystem: A programming system for compute-intensive multimedia. *IEEE Journal of Selected Areas in Communications*, 14(7):1298–1313, 1996.
20. J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5), September 1994.
21. Mircosoft. DirectX 8.0: DirectShow overview. `http://msdn.microsoft.com/library/psdk/directx/dx8_c/ds/0view/about_dshow.htm`, January 2001.
22. J. P. Morrison. *Flow-Based Programming : A New Approach to Application Development*. Van Nostrand Reinhold, July 1994.
23. D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI)*. USENIX, October 1996.
24. S. Mungee, N. Surendran, and D. C. Schmidt. The design and performance of a CORBA audio/video streaming service. In *HICSS-32 International Conference on System Sciences, minitrack on Multimedia DBMS and WWW*, January 1999.
25. OMG. CORBA telecoms specification. `http://www.omg.org/corba/ctfull.html`, June 1998. formal/98-07-12.
26. J. Ousterhout. Why threads are a bad idea (for most purposes), 1996. Invited talk given at USENIX Technical Conference, available at `http://www.scriptics.com/people/john.ousterhout/threads.ps`.
27. T. Plagemann and B. Plattner. CoRA: A heuristic for protocol configuration and resource allocation. In *Proceedings of the Workshop on Protocols for High-Speed Networks*. IFIP, August 1994.
28. C. Pu, K. Schwan, and J. Walpole. Infosphere project: System support for information flow applications. *ACM SIGMOD Record*, 30(1), March 2001.
29. H. O. Rafaelsen and F. Eliassen. Trading and negotiating stream bindings. In *Proceedings of the Second International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), LNCS 1795*, pages 273–288. IFIP/ACM, Springer, April 2000.
30. RealNetworks. Documentation of RealSystem G2 SDK, gold r4 release. `http://www.realnetworks.com/devzone/tools/index.html`, May 2000.
31. D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 145–158, February 1999.
32. W. Taymans. GStreamer application development manual. `http://www.gstreamer.net/documentation.shtml`, January 2001.
33. R. van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Proceeding of the 8th ACM SIGOPS European Workshop*, September 1998.
34. R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. Technical Report TR97-1638, Computer Science Department, Cornell University, 1997.

35. R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, September 1998.
36. J. Walpole, R. Koster, S. Cen, C. Cowan, D. Maier, D. McNamee, C. Pu, D. Steere, and L. Yu. A player for adaptive mpeg video streaming over the internet. In *Proceedings of the 26th Applied Imagery Pattern Recognition Workshop (AIPR-97)*. SPIE, October 1997.

Abstracting Services in a Heterogeneous Environment

Salah Sadou[1], Gautier Koscielny[2], and Hafedh Mili[3]

[1] Valoria Lab., Université de Bretagne Sud, France
Salah.Sadou@univ-ubs.fr
[2] LIFL Lab., U.S.T. Lille, France
Gautier.Koscielny@lifl.fr
[3] Département d'informatique, Université du Québec à Montréal, Canada
hafedh.mili@uqam.ca

Abstract. Applications often use objects that they do not create. In general, these objects belong to an execution environment and are used for some services (*server* objects). This makes applications strongly dependent on these objects and make them vulnerable to any modifications to these objects.
In this paper, we present a solution to this problem through the *service group* concept. A *service group* is an intermediary between the applications and the server objects. A *service group* is defined by the administrator of the shared services, for the entire set of client applications. A *service group* embodies a collection of signatures corresponding to the provided services and maintains the required associations between these signatures and the actual implementations of these services by the server objects. The client applications access the services through *service groups*, and are no longer directly related to servers, thus becoming more independent and better protected from modifications to the server objects. The *service group* will not only make it possible to pool disparate services in order to structure execution environment, but also to construct new services by composing existing ones.

Keywords: Distributed objects, Dynamic reuse, Disparate services, Environment structuring, Service compatibility, Service group.

1 Introduction

When building applications, we need various forms of reuse, i) the reuse of classes by creating objects from them, and ii) the reuse of *existing objects* in the system. An example of the second form of re-use is the printing daemon object that applications use when they need to carry out a printing job. Indeed, an object of this type manages the requests for printing jobs that come from various applications. It is an object that belongs to a common environment, shared by several applications.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 141–159, 2001.

What we call a common environment can be a machine or several machines on a local area network. In such an environment, there can be many such objects that provide shared services, and applications may be more or less tightly coupled to those objects.

We want to address the problem of using the services offered by these objects without being too dependent on the way those services are defined and implemented.

Two significant points arise from the statement of this problem:

Service use. Often we only reuse a part of an already built object. In this kind of situation, we are more interested in a subset of the services that it offers (its methods), than in its actual type (and full type). For instance, it is often the case that two objects, with two different types, offer equivalent services. In such a case of reuse, we must "abstract away" differences in types and focus only on the services that they offer.

Independence. We spoke about an abstraction on servers' types, but it will be necessary to go further: to have an abstraction on individual service signatures. Indeed, two similar services do not always have the same signature and if we want to replace one by the other, the call to the service, in our application, should not depend on what we consider to be non-essential aspects of a server's signature.

So, this problem is related to the evolution of service systems. In this paper, we propose a solution to the problem that makes a distinction between the definitions of services from the point of view of client applications, from the definitions of actual services as provided by actual servers. Naturally, we provide "service managers" with the means to describe *dynamically* the correspondence between these two definitions. This makes it possible to integrate a new server object in the system and to use it immediately thereafter by the existing applications. Further, we introduce the notion of a *service group*, which is a way to logically organize a disparate set of services.

In the next section we discuss related work in order to highlight the motivation of our approach. Section 3 presents the *service group* concept that provides means to define a set of needed services within the context of a particular environment. In section 4, we show the techniques used to establish correspondence between the client view and the server view of a set of services. We will discuss choices for the implementation of a *service group* in section 5, and conclude in section 6 by discussing directions for future research.

2 Related Work and Motivation

Our work touches on four separate research issues, dynamic service (re)use, typing relationships, encapsulation of multiplicity, and behavioral composition. We discuss research related to these four areas in order to highlight the motivation of our approach.

2.1 Dynamic Service Reuse

Typically, the way to implement dynamic service management and invocation involves going through intermediaries such as naming and trading services [23]. A naming service manages a hierarchical tree of name-object reference bindings. Although this approach enables an object to be designated by a unique machine-independent name, clients must know the *name* of the service beforehand. By contrast, a trading service allows clients to access services by function (however it may be described) rather than by name. More precisely, servers publish through the trader service the services that they offer. In turn, clients request services from the trader service by specifying the characteristics of those services. The trader matches the client criteria to the registered servers, and returns to the clients object references for those servers that match their criteria.

While the trading allows clients to be configured without prior knowledge of server objects, it has the following implications:

- *Strict typing requirements*. Service reuse remains complex due to the existence of different types representing the same service. Neither subtyping rules of *CORBA* and *RM-ODP* [21] nor service subtyping rules of the trading service are able to accommodate the variations in signatures (even the functionally equivalent ones) that our approach supports. Moreover, type evolution is only partially addressed in these platforms. Clearly, we need a more flexible approach to service conformance than the one offered by these approaches.
- *Server multiplicity*. Notwithstanding the problem of having servers of different types offer the same service, we also have the problem of several server *instances* that can offer the same service. Having to choose among these adds another complication to the use of these servers. There are two ways of handling this: we could either let the service manager (e.g. trader service) choose one, or let the client application choose among possible offers. The trader service does support an API for the addition, withdrawal, and selection of offers, and we may see an advantage in letting client applications choose among service offers. However, because traders "get out of the way" as soon as a client application gets a reference to a server, it becomes the responsibility of the client to ensure that the server they got is still active. If the server in question cannot be restarted for some reason, then it is the responsibility of the client application to go back to the trader for another server. This is a recurring problem that must be treated directly in the code of client applications, before *each* service invocation. Note finally that there is no transparency of access to the service invocation (access is done in two steps), leading to complexity when writing client code.

The notion of *service group* addresses both problems. First, we use the service conformance relationship (see section 4) to accommodate syntactic (renaming, reordering) and minor semantic differences (more parameters). Second, the *service group* object acts as a permanent interface between clients and servers, shielding clients from server life-cycle issues.

In [27], Singh *et al.* proposed an extension of a trading service, called a *facilitor*, that manipulates interfaces of servers, much like our approach. A *facilitor* reduces the coupling between clients and servers by hiding the number and the specifications of servers to the client. However, their approach relies on a complex agent communication language consisting of a vocabulary, a content language and a communication language.

2.2 Type Relationships

In interoperable systems, the operational interface of an object is formally specified in an interface definition language such as the *CORBA IDL* or the *DCE IDL* [1]. Type conformance in these languages is generally defined by subtyping,*i.e.* an instance of inclusion polymorphism which specifies a substitution rule between types [6]. Roughly speaking, a type τ must replace a type σ in each context where σ is expected. Thus, it allows services of one type to be substituted at run-time by instances of the subtype. The subtype relationship as defined by *CORBA* consists of pure extension, *i.e.* subtypes are obtained by *adding* operation signatures to super-types. With *RM-ODP*, a subtype may *refine* the signature of an operation of the super-type by widening (super-type) the type of an argument and narrowing the type of the result. The subtype relationship is reflexive, antisymmetric and transitive. On the contrary, our definition of type substitutability is a pre-order. It is a reflexive and transitive relation that is weaker than subtyping, *i.e.*, a subtype τ of a type σ is also coercively compatible with σ.

Several research efforts have addressed the problem of type compatibility and introduced other forms of type relationships (equivalence, conversion, service relations). In [4] and [20] a type manager is proposed that permits the addition and deletion of various compatibility relationships between types in a generic type repository. This repository consists of a type repository and a relationship repository. It provides operations for type matching and for various queries, including finding all the types that are compatible with a given type. In our approach, we want each *service group* to act as a repository of both roles and type conformance relationships between roles and service types; this repository is queried when new objects join a server group.

2.3 Encapsulation of Multiplicity

Several models have used the object group concept to make the fact that several servers may offer the same service (multiplicity transparency) transparent to clients [26,16]. Such models often consist of grouping active objects implementing a particular service so as to ensure service availability through replication. The *Gaggles* [2] model hides from clients the number of objects that implement a service. It is designed as a mechanism for invoking one out of a number of equivalent servers, in the same way that our approach does. However, *Gaggles* does not support the run-time definition and addition of services, like our *service groups* do.

Platforms such as *ANSA*[24], *GARF*[9] and *Electra*[17] propose stronger object group protocols than *Gaggles*. The *GARF* platform, for example, relies on the *Isis* [12] group communication toolkit, which provides various multicast primitives to ensure consistency between replicas in spite of failures and concurrency. Object groups are generally homogeneous, which means that all objects within a group are of the same type. The *Electra* model, like *DCOM* [19], enables the aggregation of different types of objects using a high-level interface. However, invoking a method specified in the group interface amounts to multicasting the invocation to *all* the members that implement the method. By contrast, our *service groups* provide flexible invocation mechanisms, supporting several types of protocols (one of, all of, the first one that terminates). Further, *service groups* support the behavioral composition of the available services.

2.4 Behavioral Composition

Behavioral composition refers to the composition of methods of different interacting objects. Helm *et al.* note in [11] that "patterns of communication within a behavioral composition represent a reusable domain protocol or programming paradigm" and propose *contracts* as constructs for the explicit specification of behavioral relationships between objects. A contract defines a set of communicating participants and their contractual obligations. It provides two operations for expressing a complex behavior in terms of simpler behaviors, refinement and inclusion. Contractual obligations consist of type obligations and causal obligations. In our approach, we consider only type conformance obligations to join a *service group*. However, contrary to contracts, *service groups* allows relaxed type conformance and accept the membership of different type specifications for a given role.

3 Service Group

The objectives of our approach may be summarized in four keywords: adaptation, pooling, transparent access, and service composition:

adaptation. It refers to the ability of applications to have their own expectations of services, and to match those expectations with the definitions of the actual services;

pooling. Refers to the convenience of having a single point of service for a set of related services, instead of having to bind to each server individually;

transparent access. To access a service, the client does not need to know the actual server that provides it. This enables us to interchange servers that provide a given service, without affecting the clients. Transparent access is made possible in part thanks to adaptation and pooling;

service composition. It refers to our ability to compose the services offered by actual servers to build new and more complex services. The composition of services involves the collaboration of existing servers, which is orchestrated by a *service group*.

The *service group* concept is useful to the extent that:

1. the needed services at hand are provided by generic objects that are not specific to a particular set of applications;
2. the same service may be offered by different objects, possibly of different type, or even with different invocation signatures;
3. to make the effort worthwhile, we assume that *several* applications share the same services.

3.1 Principles

Consider a situation where we have, on the one hand, a set of independently defined active objects that play the role of *servers*, and on the other, a set of client applications that have the same definitions (expectations) of the services they need. In other words, in the execution environment, there is only one definition for any given service, for the entire set of client applications. In order to shield client applications from an evolution in the way those services are implemented, we introduce a bridge between them in the form of a *service group*. For conceptual cohesion and tractability, we further assume that *service groups* pool services that are logically related such as printing services, database querying services, etc. for all of the client applications.

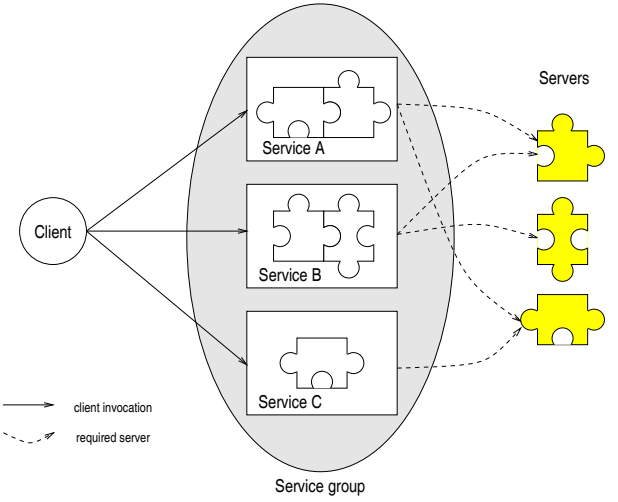


Fig. 1. Interaction of clients with a Service Group.

A *service group* is an intermediary between the applications (clients of the service) and the server objects (implementers of the service), see figure 1. A

service group is an active object defined by the administrator of the shared services, for the entire set of client applications. A *service group* embodies a collection of signatures corresponding to the provided services and maintains the required associations between these signatures and the actual implementations of these services by the server objects. Viewing a *service group* as an object, the various services it manages should not be viewed as *methods* of that object, in the OOP sense; in non-prototype OOP languages, the behavior of an object is specified at object creation time, and doesn't change after that. Services are better viewed as data held by a *service group*, making possible to add and remove services to a *service group* after it has been "created". A service invocation then is just a way to specify one item of that data; it's the responsibility of the *service group* object to translate a service invocation into actual method calls on actual server objects. In other words, a *service group* is a meta-level (reflective) object [18] centralizing the definitions and invocations of the behaviors of several other objects.

As suggested by figure 1, the services offered by a *service group* may either be implemented by a single service of server objects, or as a composition of services provided by several server objects.

3.2 Defining Service Groups

Simply stated, the definition of a *service group* consists of, 1) a description of the services provided to the clients, and 2) a description of the services required (needs) of the servers to provide them.

provided services a collection of signatures of services related to the same application domain;

needs a set of basic services/functionalities required from servers, to be able to provide **services**,

The needs are defined by method signatures and gathered in subsets representing *roles* to be played by members of the group (see figure 2). Servers join a *service group* to play a particular role. We distinguish between *role* and *role instance*. A role[1] is a generic service type that represents a set of object types providing similar services, whereas a role instance represents a binding to a particular server. A role is described by a name and a set of method signatures that define its behavior. It is instantiated at run-time by servers whose methods conform to those defined in the role (see section 4). Several instances of the same role can exist simultaneously, and independently. This means that the group can send the same request to any or several servers. Similarly, a server joining a *service group* can play (conform to) several roles. Thus, servers are dynamically bound to roles.

[1] A role may be considered as an object template and even if our model let server objects choose their roles, our approach is quite different from the role abstraction described in[15].

Further, note that an existing group may be bound to a role defined in another group, since it is also an object with a well-defined interface. A service group may represent a part-whole hierarchy[3] whose behavior may be composed in terms of other groups. Hence, roles may refer to service subgroups.

```
Group PrinterSG {
   role Printer {
     string printPs(Source doc, boolean twoSide);
   }
   role Translator {
     Source dviToPs(Source dviFile);
     Source psToRtf(Source psFile);
   }

   private Source dviToPs(Source dviFile) {
     return Translator.dviToPs(dviFile);
   }

   public string printDVI(in Source dvi,
                          in boolean twoSide) {
     Source ps = dviToPs(dvi);
     return printPs(ps, twoSide);
   }

   public string print(Source ps, boolean twoSide) {
     return Printer.printPs(ps, twoSide);
     }
}
```

Fig. 2. A printing service group definition.

A service, as shown in figure 2, may be a behavioral composition of *role methods*. A service may be declared either **public** or **private**. Only the public services of a role may be invoked by clients; private services are provided as a way of constructing more complex services to be offered by the *service group*. An example of a simple coordination is given by the *printDVI* service of figure2, which prints a document given its *dvi* format. A composite public service will not be available to the *service group* clients if one of the roles involved in the composition has not been instantiated (bound to an actual server).

New services may be supplied at run-time and added to the interface of a *service group* (see section 5). The *service group* shown in figure 2 is specified using our own extension to the *Java* language called *JGroup*. This extension isn't a new language. It's just a means to simplify the use of *service group* framework.

3.3 Service Composition

By gathering services related to the same application domain, a *service group* may offer new services by composing existing ones. This composition can be either sequential or parallel.

Sequential Composition. In the previous example, we defined a service called printDVI that prints *dvi* formatted documents directly. Based on the available servers, we know that we cannot print such documents directly: we have to transform them into a *postscript* format. Accordingly, we defined a role called *Translator* to reflect the fact that we need servers that can translate a *dvi* document into a *postscript* one, and defined the printDVI in terms of that role: to print a *dvi* document, the *service group* has to, first, translate it to *postscript* using the *Translator* role, and then, print the *postscript* document using the *Printer* role. The sequentiality of these operations is represented using the ";" operator. In the sequential composition, the invocation of a service on a role is synchronous: the first call blocks the composed service and must wait until the end of the first invocation to continue the computation.

Parallel Composition. We propose a mechanism of asynchronous invocation through the use of a parallel composition operator '||'. The expression of parallelism is thus explicit but the collection of results remains implicit.

The parallel composition combinator '||' acts as a separator between instructions (the same way the ';' sequential composition operator does). Consider, for example, the dviToRtf service defined in figure 3. This service consists (in part) of two invocations: an invocation on the translator t1 followed by an invocation on the translator t2. These two invocations are independent, and thus, can be executed in parallel. The results are collected later and the variables ps1 and ps2 represent implicit futures [7]. So, the execution of this service, will continue until it needs the values of these variables.

Because several objects can play the same role, an invocation can be multicast to several role instances. In this way, we provide a simple fault tolerance mechanism through replication [10]. For example, the new definition of the dviToPs service that is shown in figure 3 consists of invoking, asynchronously, two translator instances to perform the translation. The first result to be received is the one returned by the *service group*. The operation is performed even if only one server is bound to the translator role. In that case, the second part of the instruction is ignored.

Note, finally, that one can broadcast the invocation of a particular service to *all* of the servers that play that role. The psToRtf service illustrates this: it interacts with all the translators and returns the first available answer. If one needs several answers, servers may be called consecutively and asynchronously, as in the dviToRtf service definition.

```
Group PrinterSG {
    ...
   public bytes dviToRtf(in bytes dvi) {
     Translator t1, t2;
     bytes ps1, ps2;
     ps1 = t1.dviToPs(dvi) || ps2 = t2.dviToPs(dvi);
     ...
   }

   public bytes dviToPs(in bytes dvi) {
     Translator t1, t2;
     bytes ps = t1.dviToPs(dvi) || t2.dviToPs(dvi);
     return ps;
     }

   public bytes psToRtf(in bytes ps) {
     bytes ps = Translator||.psToRtf(ps);
     return ps;
   }
    ...
}
```

Fig. 3. Various parallel service compositions.

4 Conformance of Services

In order to play one role of a *service group*, a server must conform to the specification of that role, i.e. the set of method signatures that are specified in it. This means that for each method signature listed in the role, the server has (at least) one method that conforms to that signature. This section deals with the meaning of "conformance" in this context.

Thereafter, the concept of *type of service* refers to a particular invocation (method) signature, and to the expected behavior of the service. A strict interpretation of conformance requires that implementers of the service support a *method* that has the exact same signature and that produces the same behavior. In our case, we use a broader interpretation of conformance that makes it possible for a service offered by a group to be carried out by servers whose methods may have a different signature from that which is advertised by the server group. The type of a service, as defined by a role, may be regarded as a generic type of service [14]; it represents a *class* of services in the sense that they may have different types, provided that those differences are not 'essential'.

We consider the generic service type as the canonical representative of the set of types that are compatible in the sense of contravariance and *coercive compatibility*, defined further below. This enables us to use a service invocation protocol that factors out the commonalities between the interfaces of the various servers.

To illustrate this, we elaborate the printing service example. We assume that the users of a local area network (LAN) have several printers with *different* characteristics; some accept only simple postscript printing while others allow color and/or double-side printing. Figure 4 shows the interface definitions for the various printers.

```
interface Printer1 {
 ...
 void printPs1(Source doc, String status);
}

interface Printer2 {
 ...
 String printPs2(Source doc, boolean rv);
}

interface Printer3 {
 ...
 String printPs3(Source doc, boolean rv,
                 boolean color);
}
```

Fig. 4. Interfaces of several printing service servers.

The LAN manager wishes to unify the access to these various printers through a single definition of the printing service so that all the applications can use this unique definition of the printing service, while benefiting from all available printers. A solution is shown in the example of figure 2.

Before showing how this manager could define the links between his own definition of the printing service and those provided by the existing servers, we give some formal definitions of the relations that must exist between the two.

4.1 Some Formal Definitions

The behavior of a role is described, syntactically, by an interface that represents a set of generic service types. Several objects may conform to this definition while being completely unrelated in the type hierarchy. Relating a type to a role consists of matching a subset of its methods/services to the services defined in the role. This is done by applying a *coercive compatibility* relationship. The coercive compatibility relationship consists of a signature matching and a conformance to the role at hand.

Service Signature Matching. We must be able to define a matching procedure for each operation that is supposed to conform to the generic service

definition. Signature matching establishes a structural correspondance between the pair $\langle name, parameters \rangle$ of the operation and the pair $\langle name, parameters \rangle$ of the generic service. When the two operations have the same number of parameters, the correspondence is straightforward and only the order and the types of the parameters matter. Zaremski *et al.* in [28] proposed various flavors of relaxed match for functions defined in *ML*, including a reorder transformation that defines a permutation applied to a tuple of parameters. Using this transformation, we can reorder generic service type parameters to match operations parameters.

However, when the number of parameters is different, we must narrow the set of all appropriate inputs to the generic service type. For instance, the generic service type "*string print(Source doc, boolean twoSide)*" of figure 2, allows clients to print postscript documents in gray-scale and possibly in double-side while *printPs1* allows one side printing and *printPs3* allows color printing. Hence, *printPs3* matches *printPs* if its parameter *color* is bound to $false$ and *printPs1* matches *printPs* only for one-side printing.

Definition 1 (service signature matching).
Let $f : A \to B$ *and* $g : A' \to B'$ *be two services with:*
$A = \tau_1 \times \ldots \times \tau_n$, $B = \sigma_1 \times \ldots \times \sigma_k$,
$A' = \tau'_1 \times \ldots \times \tau'_m$, $B' = \sigma'_1 \times \ldots \times \sigma'_k$
and n, m, $k \geq 0$.
g matches f if:
(i) there exists a composition $\mathcal{T} \circ \mathcal{C} : A \to A_{\mathcal{T} \circ \mathcal{C}}$
of a coercion $\mathcal{C} : A \to A_{\mathcal{C}}$
and a reorder transformation $\mathcal{T} : A_{\mathcal{C}} \to A_{\mathcal{T} \circ \mathcal{C}}$
such that $A_{\mathcal{T} \circ \mathcal{C}} <: A'$, *and;*
(ii) $B' <: B$.

Coercion $\mathcal{C}$ is a structural transformation of the parameters of one operation f onto the parameters of another operation g. $\mathcal{C}$ is an injection onto a sub-domain $A_{\mathcal{C}}$ of the domain of g when f has less parameters than g, a projection onto another sub-domain $A_{\mathcal{C}}$ of the domain of g in the opposite case. In both cases $\mathcal{C}$ is a partial function that is not defined for some inputs belonging to the domain of f or g. In order for g to replace f, it must accept arguments of type greater than A, hence the contravariance rule on domains (condition (i)). Inversely, the result of the application of f may be passed as a parameter of another method. Thus, the result of the application of g must be able to be passed as a parameter to the same method hence the covariant rule on codomains (condition (ii)).

Conformance Relationship. Signature matching may be extended to a type, allowing the definition of a conformance relationship between a type and a role. It is a coerced behavior compatibility relationship similar to the one defined in *RM-ODP*[13]. A type τ conforms to a role ρ applying a coercion of the behavior of τ onto the behavior of ρ.

Definition 2 (role conformance).
Let τ *be a type and* ρ *be a role type. let* F_τ *and* F_ρ *be respectively, a subset of* τ *operations and a subset of* ρ *operations.*

τ conforms to ρ if for each operation $g_\tau \in F_\tau$, there exists one and only one $f_\rho \in F_\rho$ such that f_τ match g_ρ.

This definition may enable only a partial compatibility between a type and a role since matching may be applied to subsets of their interfaces.

4.2 Conformance Specification

A conformance specification describes how a type supports a role in a *service group* according to definitions 1 and 2. It is the designer of the *service group* who creates typing obligations within a *service group* in order to flesh out roles.

```
Printer1 conforms to PrinterSG::Printer {
   string printPs(Source doc, boolean twoSide)
     coerces {
       string printPs1(Source doc, String status);
      return status;       }
     where {
      twoSide in [FALSE];
     }
};

Printer2 conforms to PrinterSG::Printer {
   string printPs(Source doc, boolean twoSide)
     coerces {
       string printPs2(Source doc, boolean twoSide);
     }
};

Printer3 conforms to PrinterSG::Printer {
   string printPs(Source doc, boolean twoSide)
     coerces {
      string printPs3(Source doc, boolean color,
                                boolean twoSide);
     }
     where {
      color in [FALSE];
     }
}
```

Fig. 5. Example of a conformance declaration.

Figure 5 shows different conformance specifications for the *Printer* role of the printer *service group PrinterSG* (see figure 2). Conformance specifications are also defined in *JGroup*. Here we show how the three printing servers of figure 4 match the generic service *printPs*. In one case there is only a name mismatch;

in the other two, we also have parameter mismatch. For example, *printPs3* is compatible with *printPs* provided that the parameter *color* is set to false. To describe the link between a generic service parameter and the corresponding one from the server-provided service, we use the same parameter name in both signatures. If the server-provided service does not return a value (or the *desired* value), we use the "return" key word to return the desired value. That is the case of the *printPs1* service of the *Printer1* server, which uses a *status* parameter to indicate the status of the printing, but which doesn't return it explicitly; the coercion relationship forces the return of the *status* and ignores the actual return value of the service.

4.3 Conformance in Action

Once a conformance relationship is defined between a server type and a role, and once the definition is added to a *service group*, instances of the server type are authorized to join the *service group* in order to play that role. When a server joins a *service group*, its interface is inspected and compared to the type obligations that are stored in the group. Consequently, we may find zero, one or many role specifications that match its interface. In the case where the type of the server does not match any role specification, the server cannot join the group. Otherwise, for each role specification that the type of the server matches, the corresponding role is instantiated. A composite service will remain unavailable if one of its component roles has not been instantiated.

Specifications of conformance relationships are defined separately from the *service group* and may be provided at run-time. This approach allows the definition of new role conformance relationships when finding out about new services that are likely to match a role. The *service group* manager may also withdraw role definitions to reflect the disappearance of services or the creation of a new version of a service.

5 Implementation Choices

The reader may think that we can use *EJB app servers* [22] to implement the *service group* in order to manage resources. But, in *EJB* the pooling uses objects of the same type while *service group* allows the pooling of objects from different types.

Our main concern in implementing the *service group* concept was ease of use. As mentioned earlier, *service groups* are described by Java-like constructs illustrated in figure 2. We have developed a processor that compiles such definitions and generates Java classes. The *service group* itself is represented by a class. The behavior of a *service group* is implemented by *class* (static) methods, for several reasons, 1) to centralize (and share) the management of the various services, 2) to obviate the need for programmers to create/instantiate *service groups*, and 3) to make them aware of the fact that they are accessing a *shared* service that they don't own.

We implemented the *service group* as distributed objects to meet the needs for distributed applications. If the application is centralized, a simple proxy would be enough.

5.1 Using a Service Group

The example of figure 6 illustrates an invocation of one service of the *PrinterSG service group*. It consists of building an array of the service arguments and then, sending the request to the group through its *invokeService* class method; *invokeService* takes the name of the service and its argument list as parameters.

```
import  SG.printer.PrinterSG;
public class User{
 Source documentPs ;
 public String toPrint (boolean doubleSide) {
  String status;
  try {
    Object[] args = new  Object[2];
    args[0] = documentPs;
    args[1] = new Boolean(doubleSide);
    status =
    (String)PrinterSG.invokeService("printPS", args);
    } catch (Exception e) {
        System.err.println("printPS service error");
        }
  return status ;
  }
}
```

Fig. 6. Using a service group

For each *service group* described in the *JGroup* syntax, we generate a class by using a template that consists of the definition of the class methods representing the possible actions on the group. These methods are of two kinds:

public for methods destined to the clients/users of the *service group*. These methods include the invocation of services (as shown in figure 6), and querying actions to know about the existing services or roles, to check whether a service is active or not, etc.

protected for methods destined to the *service groups* manager. These methods manage services, roles, support the addition of new conformance relationships, and handle the registration (joining) of new servers.

5.2 Service, Role, and Server Conformance Specifications

As mentioned earlier, the specification of conformance relationships between servers (server types) and roles is provided separately, and is treated as data that *service groups* can add and remove at will during run-time. In the implementation, these specifications are provided in the syntax illustrated in figure 5, in a separate file. In fact, this is true of all the components of a *service group*: each specification, be it of a service, a role or a conformance relationship, is stored in a separate file, and compiled to produce a class. To add any of the three types of components to the *service group*, the group manager provide the *service group* with the class name so that it loads the corresponding class, creates an instance of it, an adds it to the appropriate part. To this end, the group manager has various external commands to manage existing groups (see figure 7. All these commands use the protected services of service groups.

---

addRole *groupName roleClassName*
removeRole *groupName roleClassName*
addService *groupName serviceName*
removeService *groupName serviceName*
addConformance *groupName conformanceName*
removeConformance *groupName conformanceName*
addServer *groupName serverRMIAdress*
removeServer *groupName serverRMIAdress*

Fig. 7. External commands for service group managing.

Note that the services and roles that are specified as part of the initial group definition (as opposed to defined separately) will also have classes generated for them. The difference with those that are defined externally (and added dynamically) is that they are created automatically as part of the construction (instantiation) of the group.

This approach has the advantage of making *service groups* sufficiently flexible to change their behavior during run-time, without requiring a shut-down or an interruption of service.

6 Conclusion and Future Work

To conclude this paper, we will point out the contributions of our work, its limitations, and discuss directions for future research.

6.1 Contributions

The contributions of our work do not reside in the proposal of a mediator or a facade pattern, which are well-known design patterns [8] but rather in tackling different issues, some of which are orthogonal:

Execution environment structuring. We argue that the proper structuring of the execution environment of applications is as important as the structuring of the applications themselves. A well structured environment simplifies the writing of applications. *Service groups* help in this structuring by providing a simple organization of disparate services. It unifies/factors the definitions of compatible services in order to simplify their use. The unification of the definition of a shared service for the purposes of several applications, creates a certain coherence between these applications. Because the unified definition does not depend directly on the existing server objects, the environment can evolve (change in the type or the implementation) without affecting the applications that use it.

Dynamic (re)use of services. By gathering services that are similar in function but different in protocol, we can, by the same token, hide the multiplicity of the *existing* implementations of the service (with possibly different types), and make it possible to add *new* implementations (also with different types) and let client programs take advantage of those implementations, even if they don't know their type. Our implementation goes one step further by supporting the addition of new *services*, enabling us to evolve the *service group* without shutting it down.

Composition of services. Through the JGroup extension to Java, we provide a simple mechanism for composing existing services in order to provide new ones. Thus, client applications can use services that no server alone could provide.

Our work on integrating disparate "infrastructure" services (printing, communications, database, etc) can be applied to *domain* services or objects that may be shared by several applications. For example, a set of financial applications could require the same wire-services to access currency exchange rates or live stock quotes. In this case, the *service group* concept can be used to implement pools of "read-only" objects of different types to accommodate varying application loads. The applications become like views on this environment. This helps ensure a cohesion and stability of the core of these applications.

We experimented with this approach in our own computer science department. Students, professors, administrators and courses are active objects connected to *service groups* shared by various applications. For example, we can have a professor objects that is connected, at the same time, to the professor group and to the administrator group, to reflect the fact that it may be used as both a professor and an administrator. This work was made within the "Group" project [25] that was financed by the regional council.

6.2 Limits

Our definition of a generic service type relies only on signatures but does not handle behavioral specifications [5]; conformance relationships can only guarantee type safety, but do not guarantee the behavioral conformance of the servers to the expected service. For the time being, behavioral conformance is ensured

by the vigilance of the service manager who should only define conformance relationships for those services that she/he knows do implement the actual service!

6.3 Future Work

Currently, we are working on the behavioral conformance problem. We are exploring the possibility of specifying the behavior of a service through functional and nonfunctional assertions. This will be used for both actual server types and for the definition of generic services. Some sort of theorem proving procedure may then be used to validate the behavioral conformance of a server type to a defined service, enabling us to automate all of the functions of a service manager. We are currently developing a pragmatic formal specification language that includes both linguistic constructs (such as the ones used by trader services) as well as formal ones.

References

1. P. A. Bernstein. An architecture for distributed system services. Technical Report CRL 93/6, Digital Equipment Corporation, Cambridge Research Lab, March 1993.
2. A. P. Black and M. P. Immel. Encapsulating Plurality. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 57–79, Kaiserslautern, Germany, July 1993. Springer-Verlag.
3. E. Blake and S. Cook. On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. In *ECOOP'87*, pages 41–50, 1987.
4. W. Brookes and J. Indulska. A type management system for open distributed processing. Technical Report 285, Department of Computer Science, Queensland U., Brisbane QLD (Australia), February 1994.
5. D. Buchs and N. Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Transactions on Software Engineering*, 26(7):635–652, 2000.
6. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
7. D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
9. R. Guerraoui, B. Garbinato, and K. Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4):32–39, 1997.
10. R. Guerraoui and S. André. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.
11. R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In N. Meyrowitz, editor, *Conference on Object-Oriented Programming, Systems, Languages, and Applications/European Conference on Object Oriented Programming (OOPSLA/ECOOP'90)*, ACM SIGPLAN Notices, pages 169–180, October 1990.
12. IONA and Isis. An Introduction to Orbix+Isis. Technical report, IONA Technologies Ltd, November 1995.

13. ITU/ISO. Reference model of open distributed processing - part 2 : Foundations, 1995. ISO/IEC 10746-2, ITU-T Rec. X.902.
14. G. Koscielny and S. Sadou. Type de service générique pour la réutilisation de composants. In Jacques Malenfant and Roger Rousseau, editors, *langages et modèles à objets (LMO'99)*, pages 115–130. Hermès Science Publications, 1999.
15. B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Sytems*, 2(3):143–160, 1996.
16. D. Lea. Objects in groups. Technical report, SUNY Oswego, 1993.
17. S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, February 1995.
18. J. McAffer. Meta-Level Programming with CodA. In W. Olthoff, editor, *European Conference on Object Oriented Programming (ECOOP'95)*, Lecture Notes in Computer Science, pages 190–214. Springer-Verlag, aug 1995.
19. Microsoft. Dcom.
http://www.microsoft.com/cominfo/.
20. M. Muenke, W. Lamersdorf, B. O. Christiansen, and K. Mueller-Jones. Type management: A key to software reuse in open distributed systems. In *EDOC'97*, Gold Coast, AUSTRALIA, 1997.
21. ODP. Open Distributed Processing.
http://info.gte.com/ftp/doc/activities/x3h7/by_model/ODP.html.
22. R. Monson-Haefel. Enterprise JavaBeans, 2nd Edition. March 2000, O'Reilly & Associates, INC.
23. OMG. *CORBAservices : Common Object Services Specification*, chapter Trading Object Service Specification. Object Management Group, Inc. Publications, March 1997. 97-12-23.
24. E. Oskiewicz and N. Edwards. A model for interface groups. Technical Report APM.1002.01, ANSA, Architecture Projects Management Limited, Cambridge, UK, May 1994.
25. S. Sadou, G. Koscielny, P. Frison, and J-M. Inglebert. Groupes pour la coopération entre activités. Technical report, Valoria/Orcade, Université de Bretagne Sud, December 1999.
http://www.iu-vannes.fr/ sadou/groop/rapport/rapport.html.
26. K. Shimizu, M. Maekawa, and J. Hamano. Hierarchical Object Groups in distributed Operating Systems. In IEEE Computer Society and Technical Committee on Distributed Processing, editors, *The 14th International Conference on Distributed Computing Systems*, pages 18–24, San Jose, California, June 1988. IEEE, Computer Society Press.
27. N. Singh and M. A. Gisi. Coordinating distributed objects with declarative interfaces. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models (Coordination'96)*, number 1061 in Lecture Notes in Computer Science, pages 368–385. Springer, 1996.
28. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.

An Efficient Component Model for the Construction of Adaptive Middleware

Michael Clarke[1], Gordon S. Blair[2], Geoff Coulson[1], and Nikos Parlavantzas[1]

[1]Distributed Multimedia Research Group, Computing Department, Lancaster University, Lancaster LA1 4YR, UK.
{mwc,geoff,parlavan}@comp.lancs.ac.uk

[2]Dept of Computer Science, University of Tromsø, N-9037 Tromsø, Norway.
(On leave from Lancaster University)
gordon@cs.uit.no

Abstract. Middleware has emerged as an important architectural component in modern distributed systems. Most recently, industry has witnessed the emergence of component-based middleware platforms, such as Enterprise JavaBeans and the CORBA Component Model, aimed at supporting third party development, configuration and subsequent deployment of software. The goal of our research is to extend this work in order to exploit the benefits of component-based approaches within the middleware platform as well as on top of the platform, the result being more configurable and reconfigurable middleware technologies. This is achieved through a marriage of components with reflection, the latter providing the necessary levels of openness to access the underlying component infrastructure. More specifically, the paper describes in detail the OpenCOM component model, a lightweight and efficient component model based on COM. The paper also describes how OpenCOM can be used to construct a full middleware platform, and also investigates the performance of both OpenCOM and this resultant platform. The main overall contribution of the paper is to demonstrate that flexible middleware technologies can be developed without an adverse effect on the performance of resultant systems.

1 Introduction

Middleware has emerged as an important architectural component in modern distributed systems. The role of middleware is to offer a high-level, platform-independent programming model to users, and to mask out problems of distribution. Examples of key middleware platforms include CORBA, DCOM and the Java-based series of technologies (RMI, JINI, etc). These platforms generally provide an *object-oriented* programming model for the development of distributed applications and services. More recently, however, the industry has witnessed the emergence of *component-based approaches* such as JavaBeans, Enterprise JavaBeans, .NET and the CORBA Component Model (contained in CORBA v3). Such platforms aim to provide underlying support for the third party development, composition and subsequent deployment of components, and also typically ease the task of the management of non-functional properties of applications (e.g. security).

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 160-178, 2001.

With the approaches described above, component-based models are offered on top of the middleware platform. We however believe that there are considerable advantages to also exploiting component-based techniques *within* the middleware platform. In other words, a middleware platform would then be one particular configuration of components, thus encouraging both configurability and reconfigurability of the platform. For example, this approach would enable the selection of a minimal middleware configuration for an embedded device, or indeed a richer configuration with additional quality of service management facilities to offer guaranteed multimedia services. More specifically, we advocate the use of component-based techniques together with *reflection* [7] for developing next generation middleware platforms. Middleware platforms traditionally have a black-box architecture; in our approach, we exploit reflection to open up this black box and to encourage introspection and indeed adaptation of the underlying structure and behaviour of the platform [2]. The resultant platform exploits a (minimal) component model to construct the middleware platform, with the middleware platform then supporting an enhanced component model for the subsequent development of distributed applications.

Previous papers have reported on the motivation and design of OpenORB, our component-based reflective middleware architecture [1] [2]. Prototypes of this architecture have also been developed using the Python language [4]. This paper reports on OpenCOM; an efficient, lightweight and reflective component model that we have used to efficiently re-engineer OpenORB.

The specific goals of this paper are:

- to provide a detailed introduction to OpenCOM in terms of both design and implementation,
- to illustrate how OpenCOM can be used to construct a configurable and reconfigurable middleware platform,
- to investigate the performance of the underlying OpenCOM component model, and also (briefly) the resultant middleware platform.

The rest of the paper is structured as follows. Section 2 reports on the design of OpenCOM, highlighting the programming model offered by OpenCOM, and also the associated meta-interfaces. Section 3 then reports on the associated implementation of this component model. Following this, section 4 outlines the re-engineering of our OpenORB architecture using OpenCOM, while section 5 presents a performance evaluation of both the underlying component model and the resultant middleware platform. Section 6 contains some discussion of related work, and section 7 contains some concluding remarks.

2 The Design of OpenCOM

2.1 Background

OpenCOM is a lightweight and efficient *in-process* component model[1], built atop a subset of Microsoft's COM. We chose COM as the basis of our component model for

1 Meaning that all components in an OpenCOM based system exist in a single address space.

the following reasons: *i*) COM is standardised [10], well understood and widely-used, *ii*) it is inherently language independent, and *iii*) it is significantly more efficient than other component models (such as JavaBeans).

In implementing OpenCOM we ignore higher-level features of COM, such as distribution, persistence, security and transactions, and rely only on certain low-level 'core' aspects. This is because our approach, as mentioned above, is to implement higher-level features such as these in a middleware environment that is itself constructed from components. The core on which we implement OpenCOM consists of the following: *i*) the binary-level interoperability standard (i.e. the *vtable* data structure), *ii*) Microsoft's Interface Definition Language (IDL), *iii*) COM's globally unique identifiers (GUIDs), and *iv*) the *IUnknown* interface (for interface discovery and reference counting). A brief overview of COM, which explains these features, is given in Appendix A.

OpenCOM builds on this core subset of COM as follows:

- it makes explicit the *dependencies* of each component on its environment, i.e., on other components (this is an essential requirement for run-time reconfiguration as it is not otherwise possible to determine the implications of removing or replacing a component [9]);
- it adds mechanism-level functionality for reconfiguration, such as mutual exclusion locks to serialise modifications of inter-component connections;
- it adds support for pre- and post- *method call interception*, enabling us to inject monitoring code (e.g. to drive reconfiguration policies), and offering a lightweight means of adding new behaviours that do not require a reconfiguration of existing components (e.g. security checks on method calls).

Essentially, OpenCOM reinterprets, in an efficient and standards based environment, the reflective introspection and adaptation capabilities we have identified as useful in our earlier work [1].

2.2 Functionality

The fundamental concepts in OpenCOM are *interfaces*, *receptacles*[1] and *connections*. Whereas an interface expresses a unit of service *provision*, a receptacle expresses a unit of service *requirement* and is used to make explicit the dependency of one interface on another (and hence one component on another). For example, if a component requires a service *S*, it would declare a receptacle of type *S* which would be *connected* at run-time to an external interface instance of type *S* (which would be provided by some other component). Thus, as well as declaring interfaces in the usual way, components that depend on services offered by other components must additionally declare a set of receptacles. In our current design, each component can only support a single receptacle of any given type. However, we also support so-called *multi-pointer receptacles* which can be connected to more than one interface instance (see section 3.1 for more detail).

OpenCOM deploys a standard run-time substrate that is available in every OpenCOM address space (it is implemented as a singleton component called

1 The term 'receptacle' is also employed by the CORBA Components Model [14]. The concept itself appears in various other models under various names.

"OpenCOM" and exports an interface called *IOpenCOM*). The primary role of the run-time is to manage a repository of available component types and thus support the creation and deletion of components; this builds on underlying COM facilities. In addition, the IOpenCOM interface serves as a central point for the submission of all requests to connect/ disconnect receptacles and interfaces in its address space. Furthermore, to facilitate reconfiguration, the run-time records every creation/ deletion of each component/ connection in a per-address space meta-structure called the *system graph*. This enables it to support queries (again, on the IOpenCOM interface) which, given a connection identifier (see *IMetaArchitecture* below), yield details of the receptacle and interface(s) participating in the given connection, together with details of their hosting components.

Each OpenCOM enabled component must implement the following pair of *component management* interfaces. These are called by the runtime and assist it in, respectively, creating/ deleting connections and in creating/ deleting components:

- *IReceptacles* offers operations to alter the interface(s) currently associated with (i.e., connected to) each of the host components' receptacles. These operations are only ever called by the run-time's connection management operations.
- *ILifeCycle* offers operations to be called by the run-time when an instance of the host component is created or destroyed. This interface essentially fulfils the role of constructors and destructors in an object-oriented language (we cannot rely on the availability of such facilities in our language independent environment).

Furthermore, each OpenCOM enabled component must inherit the implementation (through *containment* [17]) of three standard sub-components (called *MetaInterception, MetaArchitecture* and *MetaInterface*). These implement the reflective facilities identified in our previous work [1] and (respectively) export the following meta-interfaces from the host component:

- *IMetaInterception* enables the programmer to associate (dissociate) *interceptor* components with (from) some particular interface. Interceptors implement interfaces that contain *interceptor methods*; these are invoked before or after (or both before and after) every method invocation on the specified interface. Multiple interceptors can be added/ removed at run-time and reordered as desired.
- *IMetaArchitecture* enables the programmer to obtain the identifiers of all current connections between the host components' receptacles and external interfaces. These identifiers can then be submitted to the above-mentioned IOpenCOM interface which returns information on the receptacle/ interface/ components involved in the connection.
- *IMetaInterface* supports inspection of the types of all interfaces and receptacles declared by the host component.

Figure 1 visualises the component model. It shows the OpenCOM run-time component (below) and an OpenCOM enabled component (above). The components' management and meta- interfaces are shown on its left hand side. The three meta-interfaces are linked to the embedded sub-components that implement OpenCOM's reflective capability. Of these, *MetaArchitecture* and *MetaInterface* are further linked to corresponding private interfaces in the run-time. Also associated with the illustrated

component are a component specific interface (labeled "custom interface") and two receptacles. Components can export any number of component specific interfaces and receptacles. The OpenCOM runtime component is shown encapsulating the system graph and type libraries, and exporting the *IOpenCOM* interface.

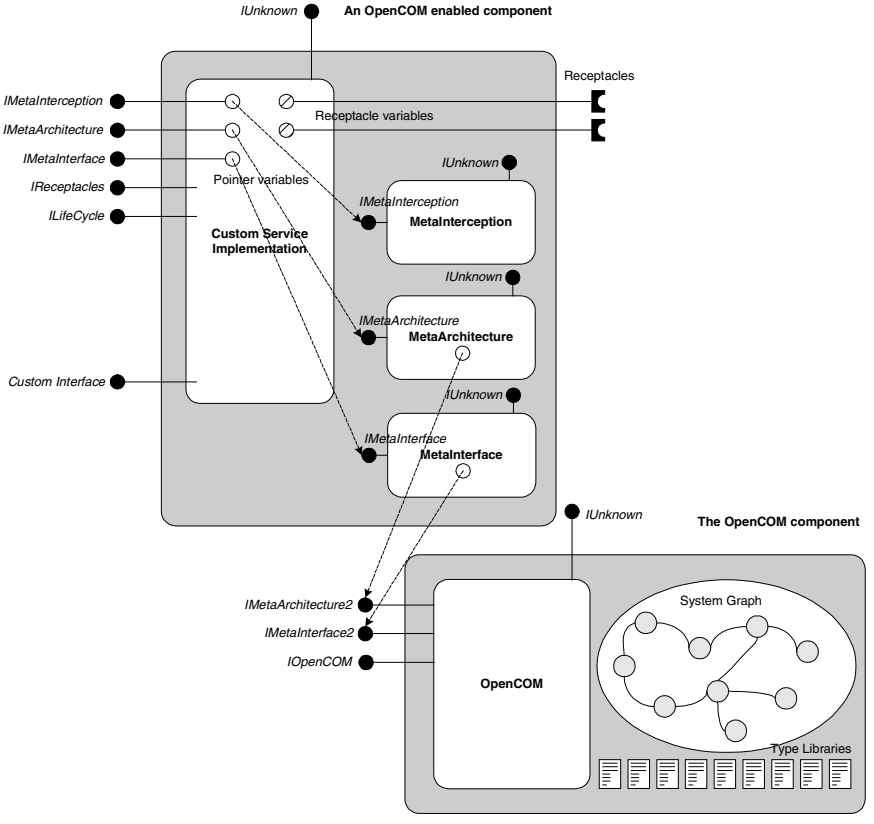


Fig. 1. The Architecture of OpenCOM.

2.3 Reconfiguration Support

The ability to dynamically reconfigure a system is most useful when these reconfigurations can occur arbitrarily and from any point within the system (not just from within the participating members of the reconfiguration). This is known as third party dynamic reconfiguration. However, any system that proposes to allow this type of reconfiguration must consider the implications for its own stability and integrity in the face of such operations. For instance, deleting a component while it is still being used will cause the results to be at best unpredictable and at worst lead to a crash.

COM supports the dynamic configuration of components and can also guarantee to delete components when they are no longer in use (assuming that strict reference

counting guidelines have been followed). In addition, components can also be dynamically reconfigured (but with no support for ensuring the resultant system's integrity) in the sense that they may have their raw interface pointers resolved against new interfaces. However, this can only be achieved from a first party point of view, i.e. only the component hosting the interface pointer can attach to a new interface. This is because interface pointers are simple variables that are not known to the runtime and therefore cannot be arbitrarily reconfigured by a third party.

In contrast, OpenCOM supports arbitrary, third party dynamic reconfiguration of both components and their connections. This is achieved through a combination of the first class status we place on receptacles (thus allowing component interdependencies to remain explicit and be managed at runtime) and the system graph (to allow access to the current runtime status of these interdependencies). OpenCOM makes dynamic reconfiguration safe by associating a lock with each receptacle. This lock is asserted whenever the first of an arbitrary number of simultaneous invocations takes place on the receptacle and is reset when the last invocation completes. During this time, any reconfiguration operations[1] involving the associated connection are blocked. However, before such a reconfiguration operation actually blocks, it is able to ensure that any new invocations are aborted (guaranteeing that it will acquire the lock when the current batch of invocations complete). Once the lock is acquired, all future invocations continue to be aborted until the lock is reset by higher level software (presumably after the receptacle has been successfully connected elsewhere).

3 The Implementation of OpenCOM

In this section we present the implementation of OpenCOM focusing mainly on the concepts introduced in the previous design section. Although our current implementation is in C++ and the material below occasionally refers to C++ specific concepts, the design is sufficiently generic to be implemented in any language compatible with COM.

3.1 Receptacle Implementation

Developers declare *receptacles* as a templated class (templated by its interface type) within the body of the implementation of their OpenCOM enabled components. A receptacle contains (among other things as discussed in section 3.1.2 below) an interface pointer and the supported interface type (expressed as a COM IID). When a receptacle is invoked, the interface pointer is used to invoke methods on the currently associated interface. The stored IID allows the component developer to differentiate between the various receptacles that their component implements and is used in the implementation of the *IReceptacles* interface (see section 2.2). The developer must ensure that the correct receptacle is used to store an interface pointer passed in by the

1 By reconfiguration operation we mean the disconnection of a connection either directly or as a consequence of a component deletion (which causes all of the components' connections to be disconnected). This is followed by a reconnection to a new interface implementation to complete the reconfiguration.

run-time at connection time and, conversely, that the correct receptacle has its interface pointer set to NULL at disconnection time.

In general, we have found three styles of receptacle useful in our implementation:

- the *single pointer* receptacle contains a single pointer to an interface. It is the most common form and represents a simple requirement to utilise a given type of interface,
- the *multi-pointer-with-context* receptacle contains multiple pointers to implementations of the same type of interface. The pointers are discriminated by passing in contextual information when invoking a method on the receptacle. This style is used heavily when there is a need to select one of a number of plug-ins in a Component Framework (CF), see section 4.1,
- the *multi-pointer* receptacle contains multiple pointers to implementations of the same interface type but does not discriminate between them. It is useful for event notification where a callback is invoked on all the interfaces connected to the receptacle.

3.1.1 Locking and Non-locking Receptacles. OpenCOM offers mechanism-level support for the maintenance of system integrity in the presence of dynamic reconfiguration through the provision of per-receptacle locks. However, OpenCOM can be built with or without these locks and this does not affect the way in which receptacles are invoked or manipulated from their users point of view.

Without locking, invocations on receptacles do not incur locking overhead, but reconfiguration operations are potentially unsafe because they may disturb currently executing invocations. In this case, it is assumed that higher level software is constructed in such a way as to make reconfiguration safe at its own level (see section 4.1). In contrast, when locking is used, higher level software can rely on OpenCOM to make reconfiguration safe but must incur an invocation overhead (see section 5).

Invoking a receptacle is achieved by calling its overridden de-reference operator (i.e. `->()` in C++) along with the desired method, c.f. smart pointer classes. In the non-locking case, this simply returns the stored interface pointer and the compiler then generates code to invoke the supplied method on the pointed-to interface. In contrast, the sequence of events that occur after an invocation of a method on a locking receptacle are more complex and are examined in detail below.

3.1.2 Invocation of Locking Receptacles. To understand the implementation of locking receptacles, one must first be aware of the layout of a COM component in memory. Essentially, each component instance contains a sequence of pointers to vtables (each known as an *lpvtbl* – long pointer to vtable) for each interface it implements (see 'the component' in figure 2). Given a pointer to an interface, i.e. a pointer to an lptvbl, and a method to invoke on that interface, the compiler generates code to follow the pointers to arrive at the vtable and add an offset corresponding to the offset of the method in the interface's IDL specification. The slot at the calculated offset into the vtable points to the method's implementation, which is then called.

Our locking scheme requires the insertion of *reference counting* code to record the number of in-progress invocations on a receptacle (the run-time can only obtain a receptacle's lock if this count is zero) and *lock status checking* code that must be

executed on each receptacle invocation. The latter code is implemented as follows: Each receptacle contains a 'fake' lpvtbl field pointing to a fake vtable also embedded within the receptacle. The overridden de-reference operator returns a pointer to the receptacle's fake lpvtbl thus ensuring that subsequent invocations pass through the locking code (see figure 2 – in the 'before' interception state). Each slot in the fake vtable points to hand-crafted assembly code that calculates the offset of the compiler's call into the fake vtable, checks to see if the receptacle has been locked by the run-time (if so, the invocation is aborted with an error code[1]), increments the reference count, calls the intended method (by forming an address from the calculated offset and the stored interface pointers lpvtbl), decrements the reference count and returns the result to the invoker.

Note that it is not viable to simply set a receptacle's interface pointer to NULL and catch the ensuing exceptions that this would cause. This is partly because many COM compliant languages do not support exceptions. In addition, reference counting of in-progress invocations would still required be make component instances deletion-safe. Finally, our invocation abort code is far more efficient than generating and handling an exception.

3.2 Meta-space Implementation

This section details the realisation of the standard components that implement each of OpenCOMs' meta-spaces and which must be inherited by every OpenCOM enabled component.

3.2.1 MetaArchitecture. The meta-architecture meta-space component leverages support from the OpenCOM runtime (in terms of accessing a private interface) in order to access the system graph. The graph stores numerous pieces of information about each component when they are created and updates this information as and when the components are involved in reconfigurations. Most importantly, the graph maintains two lists for each component representing the identities of the components connected to their interfaces and connected from their receptacles respectively. This information allows the meta-architecture component to isolate all the connections that the current component is involved in.

3.2.2 MetaInterface. COM supports interfaces as first class entities and provides a convenient way to query a component for them, namely the Type Library facility. These are binary files generated at the same time that the component's IDL files are compiled and contain many type details about a component's implementation that

---

1 As COM uses the *_stdcall* calling convention, aborting a method call presents the difficulty of having to clean the stack as part of the abort. This is achieved by maintaining a table inside each receptacle that indicates the number of parameter bytes for each method in the interface of the receptacle's type. This information is gleaned from the interface's type library (see section 3.2.2) and is filled in when the receptacle is first connected to an interface. We define macros for receptacle invocation that embody different behaviour to cope with aborted calls. The most widely used is a macro that simply 'spins' on an invocation that is aborted until it succeeds, i.e. when the receptacle is (re)connected to an interface. Using macros avoids intrusion on the application code.

would otherwise be lost when its source files are compiled. The meta-interface meta-space component uses the COM system *ITypeLibrary* interface to query a component's type library file and return the IID's of the interfaces it implements.

Ideally, we would like to extend Microsoft's IDL to allow receptacles (i.e. required interfaces) to have the same status as interfaces, i.e. to be emitted as part of a type library and made accessible through the *ITypeLibrary* interface. Currently, however, we tie the publication of a components' receptacles into its implementation (i.e. part of the declaration of its receptacles) and have our runtime extract them from the component's host Dynamic Link Library (DLL) using a pattern.

3.2.3 MetaInterception. The implementation of our per-interface interception architecture (embodied by the meta-interface meta-space component) is based on a marshal-by-value delegation architecture proposed by Brown [3], but extended with dynamic instantiation capabilities. In our architecture, we can dynamically attach and detach lists of pre- and post-processing methods over any interface. All clients of that interface transparently execute these methods before and/ or after any call to a method on that interface.

The method interception mechanism is very similar to the one used by locking receptacles. In fact, receptacles can be viewed as specialised interceptors, i.e. interceptors that have specific and fixed pre- and post-method processing routines (i.e., for reference counting, lock checking and call abortion). However, a fundamental difference lies in the way that the interception code is entered. A receptacle relates only to a single connection, whereas an interceptor needs to be present in every connection that the intercepted interface is participating in. For this reason it is not possible to simply integrate an interceptor with every receptacle instance because interception over the target interface would occur only on that connection to the interface. To resolve this issue, instantiation of an interceptor over an interface causes the *real* lpvtbl in the component instance hosting the interface to be overwritten with a pointer to a fake vtable inside the interceptor. All invocations on the interface are now directed to the interception code. When deleting an interceptor, the component instance's lpvtbl to the intercepted interface is restored. Note that this mechanism is completely separate to that used by the receptacles; when a receptacle's interception code invokes a real interface method, the invocation is transparently intercepted by any attached interceptor.

Figure 2 shows receptacle based invocation and interface interception working together according to the descriptions above. In this diagram, the receptacle (left) is of interface type IY and contains a pointer to the IY interface of the component (middle). This interface is intercepted by the interceptor (right) in order to add pre and post processing routines over all the implementations of the methods in interface IY.

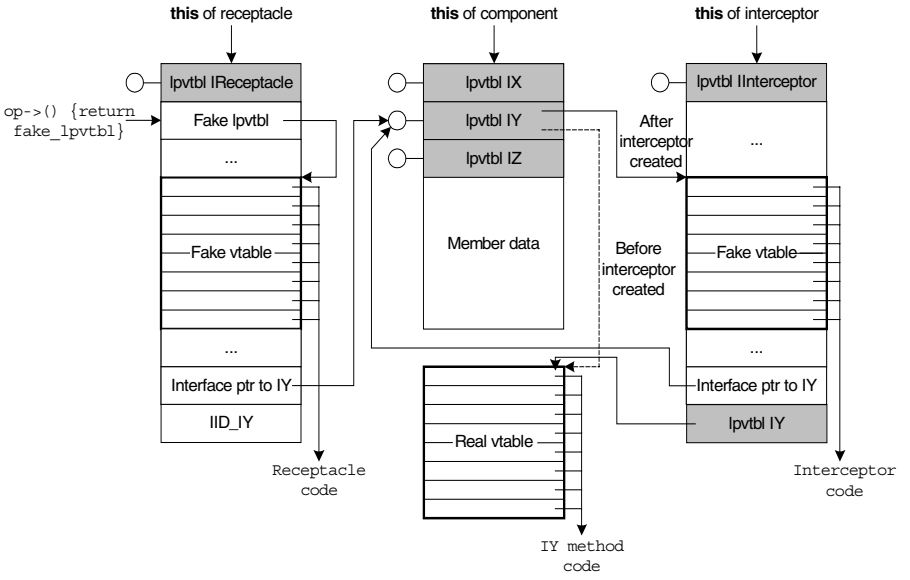


Fig. 2. Receptacles and Interceptors in OpenCOM.

4 OpenORB v2: An OpenCOM Case Study

Our ultimate purpose in designing and implementing the OpenCOM component model is to experiment with the construction of adaptive middleware platforms. This section details our experiences in implementing such a platform which we have named OpenORB v2 in reference to our previous implementation of reflective middleware [4].

4.1 Design

Although OpenCOM allows the construction of lightweight, efficient and reconfigurable components it does not directly support their *sub-composition*, c.f. nested components that can be treated as an individual unit through a unified API. This was a deliberate design decision; we do not wish to enforce any particular system-wide nesting model upon the platforms built from OpenCOM. We envisage that different domains within these platforms would have different nesting requirements and it is therefore the responsibility of the platform builder to specify appropriate support for nesting within the components that populate each of these domains.

In particular in the design of OpenORB v2, we have instantiated the notion of *Component Frameworks* (CFs) [18]) to support the nesting of components. CFs refer to "collections of rules and interfaces (contracts) that govern the interaction of a set of components plugged into them". OpenORB v2's CFs each define an abstract interface

and manage different implementations of this interface embodied by and plugged in as separate components[1] (see figure 3). CFs are targeted at a specific domain and embody rules and interfaces that make sense in that domain. The idea is that users of CFs interact with them for services through well defined APIs that encompass the services of the CF's constituent components. Additionally, these APIs include operations for the constrained (re)configuration of the CF. This implies that in OpenORB v2, it is only the CFs themselves that use OpenCOM's runtime support for reconfiguration (IOpenCOM); external entities use the CF's own API. For instance, the communications domain of a middleware platform may mandate that it will only accept reconfiguration operations on sub-components that support a specific interface, e.g. IProtocol, so that it can constrain its own reconfiguration to units of communication protocols (rather than allowing the replacement of the whole domain).

Note that the design of our CFs does not mandate whether they must be supported by a locking or non-locking OpenCOM substrate. Although using locking receptacles makes reconfiguration trivially safe it does incur overhead (see section 5). Alternatively, a CF may be able to avoid the need to use locking receptacles through other techniques, e.g. the checkpointing of safe reconfiguration points. This becomes plausible if the CF restricts the reconfiguration options for its sub-components.

4.2 Structure

OpenORB v2 is structured as a top-level CF that is composed of three layers of further CFs (see figure 3). The top level CF enforces the three layer structure by ensuring that each component/CF only has access to interfaces offered by components/CFs in the same or lower layers. The second level CFs address more focused sub-domains of middleware functionality (e.g., binding establishment and thread management) and enforce appropriate sub-domain specific policies.

The *resources layer* currently contains *buffer, transport,* and *thread management* CFs which respectively manage buffer allocation policies, transport protocols and thread schedulers. Next, the *communication layer* contains *protocol* and *multimedia streaming* CFs. The former accepts plug-in *protocol* components and the latter accepts *filter* components. Finally, the *binding layer* contains the *binding CF* that accepts *binding type implementations*. This is a crucial part of the platform's architecture because it determines the programming model offered to its users.

4.3 Implementation

OpenORB v2 consists of approximately 50,000 lines of C++ (including 10,000 lines for the OpenCOM runtime and support components) divided into 30 components and six CFs[2]. The bulk of the OpenORB v2 code is derived from GOPI, a CORBA

---

1 The CFs employ a *multi-pointer-with-context* receptacle to select between multiple managed components at run-time.

2 Note that not all of the components belong to a second level CF. Some exist purely as independent services and are therefore not exposed for semantically managed reconfiguration (though they could still be reconfigured through direct access to the IOpenCOM runtime interface if desired).

compliant, multimedia capable, middleware platform that we have developed previously [5]. We chose to reuse an existing middleware platform's code base in order to reduce the development effort needed to produce an OpenCOM enabled ORB. It has allowed us to rapidly experiment with aspects of dynamic reconfiguration within the ORB rather than be side-tracked by the development of its services.

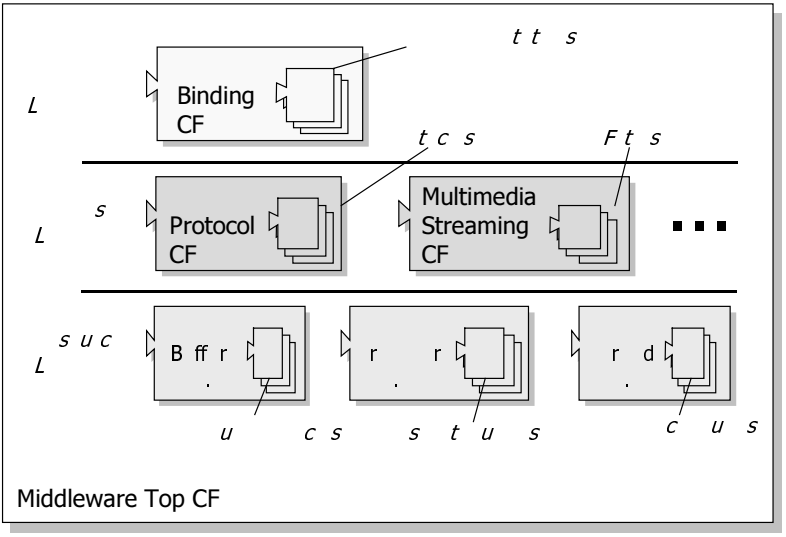


Fig. 3. Top level architecture of OpenORB v2.

GOPI was originally written in C for Unix platforms and consists of a single library statically linked to its applications. In reusing GOPI's code base within an OpenCOM environment we had to undertake a number of tasks, including i) the porting of GOPI to Win32 (as COM and its tools are only faithfully implemented on this platform), ii) the conversion of GOPI to C++ (as COM components are most conveniently implemented in C++), and iii) the conversion from C++ to OpenCOM (i.e. the breaking down of the static GOPI library into dynamically loadable components).

Particularly problematic experiences during this process were:

- the re-implementation of a number of Unix style services under Win32 including; the *signal* abstraction used heavily in GOPI timing code (re-implemented using events and some undocumented Win32 Structured Exception Handling code) and *pipes* (re-implemented using shared memory through memory mapped files),
- the identification of discrete services and their publicly available methods from the C code in order to guide their C++ re-implementation in terms of classes (the basis of COM components) and pure virtual classes (the basis of COM interfaces),
- the use of C++ class and pure virtual class definitions in the reverse engineering of IDL component and interface specifications respectively when migrating the C++ code to the OpenCOM environment,
- the identification of the interdependencies between OpenORB v2 components to facilitate the declaration of their receptacles, and

- the isolation of dependency interactions (i.e. invocations on dependent interfaces) within each OpenORB v2 component, which were then replaced by receptacle based invocations.

Our experience with this sizeable implementation effort has alleviated concerns we had about the explicit identification of component inter-dependencies; we feared that this may lead to a combinatorial explosion in the higher layers such that every component would begin to directly depend upon every other. This would make it difficult to code such components and make the system graph extremely complicated. However, we found that the maximum number of direct dependencies was seven (on the IIOP component) while the average was just three.

5 Performance Evaluation

In this section, we investigate the performance of OpenCOM and the overhead of its deployment within OpenORB v2. To provide meaning for the figures, we also compare against relevant baseline and equivalent technologies. All tests in the subsequent sections were performed on a Dell Precision 410MT workstation equipped with 256Mb RAM and an Intel Pentium III processor rated at 550Mhz. The operating system used was Microsoft's Windows2000 and the compiler was Microsoft's cl.exe version 12.00.8804 with flags /MD /W3 /GX /FD /O2.

5.1 Performance of OpenCOM

In evaluating the performance of OpenCOM we are primarily interested in the additional overhead of its augmentations over COM *on the functional control path*, i.e. the overhead that OpenCOM introduces into a platforms' services. Specifically, we do not try to measure the overhead of *non-functional* OpenCOM characteristics (i.e. reconfiguration and architectural and interface reflection) because such operations are relatively rare and are executed off the functional control path, e.g. by third parties monitoring the functional aspects of the system..

The primary mechanisms of OpenCOM that affect the performance of a systems functional control path are receptacle based invocations and intercepted invocations[1] (i.e. intercepting reflection). Figure 4 presents the raw performance of these mechanisms in terms of maximum calls/sec throughput on a method with a NULL body. We compare against C based invocations (the basis for method calls in GOPI) and COM invocations (the baseline). In addition, we provide figures for the various combinations of locking / non-locking receptacles and interception in OpenCOM.

1 The interceptors used in these tests had a single pre and post method attached, each with a NULL body.

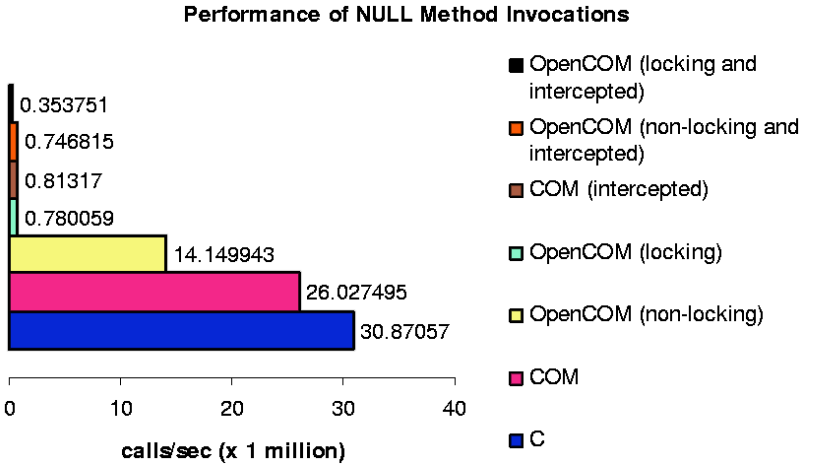


Fig. 4. Performance comparison of NULL method invocations.

The figures demonstrate a marked difference in the processing needed to execute simple method calls compared to the complex interactions embodied by OpenCOM based invocations. The C based invocation simply loads an immediate register and executes a machine level `CALL` through that register. A COM based invocation (i.e. a C++ virtual method call) must traverse lpvtbl and vtable pointers (through memory accesses) before performing the method `CALL`. A non-locking receptacle based invocation must execute the code for the overridden de-reference operator (to access the receptacle's interface pointer) before performing a virtual method call on that pointer. Section's 3.1.2 and 3.2.3 discuss the considerable processing involved in locking receptacle based and intercepted invocations respectively. As expected, there is slightly more overhead involved in locking than interception (see the intercepted COM and locking OpenCOM figures) despite both using similar techniques. This is because locking requires synchronisation to protect its lock variable. We use a Win32 `CRITICAL_SECTION` object to minimise this overhead as it spins at user level for a pre-determined time before blocking in the kernel.

Although the difference in raw invocation throughput between COM and OpenCOM is considerable (especially when using locking and intercepted invocations) it does not represent the actual effect of OpenCOM on a real system. This is because it is expected that the time taken to invoke a method is far smaller than the time taken to actually execute the method's body. Figure 5 demonstrates the effect of replacing the empty method body used in figure 4 with a relatively busy method body (implementing a 1000 iteration empty loop). It clearly shows that as the complexity of the method itself grows, the overhead of its invocation becomes less significant and the various invocation techniques begin to converge in terms of call throughput.

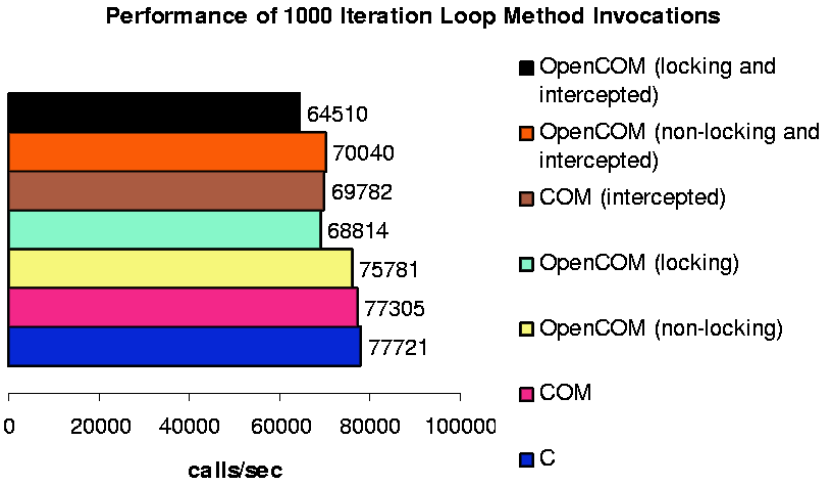


Fig. 5. Performance comparison of complex method invocations.

5.2 Performance of OpenORB v2

Given that receptacle and interceptor based invocations should not overly affect the performance of a realistic system, i.e. one that performs significant amounts of processing within its methods, then we would expect that OpenORB v2 will perform on a par with an equivalently coded system without OpenCOM. To test this theory, we compared the performance of OpenORB v2 with that of two other ORBs: GOPI v1.2 and Orbacus 3.3.4. As stated, GOPI provided much of the source code for OpenORB v2 (i.e. is an equivalent system) but is written in C and implemented in a single library. Orbacus is well known as one of the fastest and most mature CORBA-compliant (i.e. equivalent to OpenORB v2 in this sense) commercial ORBs available.

Our tests compared raw RPC method invocations per second *over the loopback interface* on the reference machine for each ORB. An IDL interface was employed that supported a single operation that took as its argument an array of octets of varying size and returned a different array of the same size. The implementation of this method at the server side was null. OpenORB v2 was tested with both a locking and non-locking OpenCOM substrate and each of it's RPCs involved 67 receptacle based invocations on the control path (32 on the client side and 35 on the server side). The results are shown below in figure 6. It can be seen that for packets of less that 1024 octets, non-locking OpenORB v2 performs about the same as Orbacus, with GOPI running around 10% faster. Locking OpenORB v2 runs around 15% slower than GOPI, i.e. only 5% slower than non-locking OpenORB v2. Both GOPI and OpenORB v2 fair slightly better than Orbacus as packet size goes beyond 1024 octets; we believe this is due to the buddy memory allocation scheme [8] that they use (which performs better for larger buffer allocations). As might be expected, there is a diminishing difference between all three systems as packet size increases further; this is presumably because the overhead of data copying begins to outweigh the cost of call processing.

Despite the additional work involved in carrying out receptacle based invocations, it can be seen that the performance of OpenORB v2 is entirely comparable to the non-componentised ORBs in both non-locking and locking configurations.

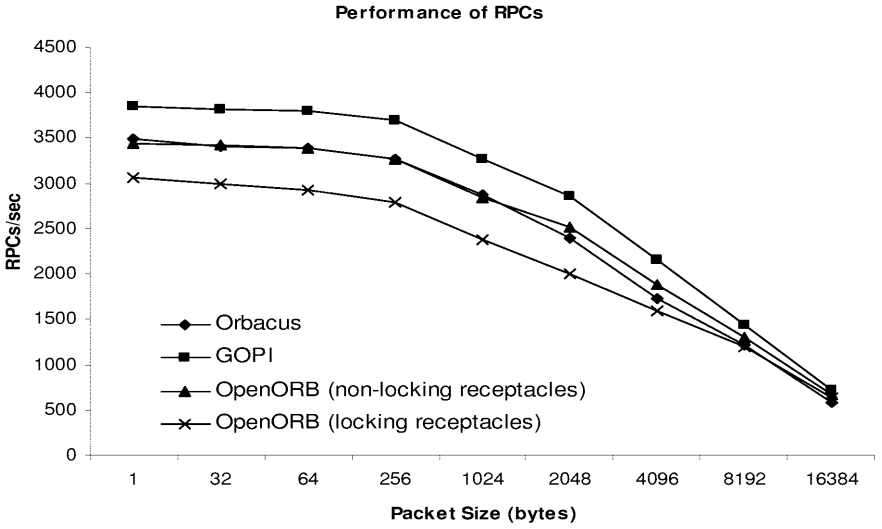


Fig. 6. Performance of OpenORB v2 versus GOPI and Orbacus.

6 Related Work

COM+ [11], Enterprise JavaBeans [19] and CORBA Components [14] are heavy-weight component models for building transactional distributed applications. They all employ a similar architecture providing a separation between the functional aspects of the application, which are captured by the components, and the non-functional, technical concerns, which are captured by the *container*. In contrast, OpenCOM is a lightweight, minimal component model which can be used uniformly throughout the system. Container-based component models can be built on top of OpenCOM if required. .Net [12], the new component model from Microsoft, is a major improvement over COM/COM+ in terms of introspection and dynamic type generation facilities. However, it still follows the same heavy-weight, container-based philosophy, whereby infrastructure services such as remoting (remote method invocation) are inseparable parts of the .Net runtime.

XPCOM [13] is a lightweight component model that, similarly to OpenCOM, is built on top of the core subset of COM. However, it does not provide any special support for dynamic reconfiguration. Knit [15] is a component model for building systems software. However, the model is specifically designed for statically composing systems; the components and their interconnections do not change after the system is configured and initialised. The component interfaces are not object-based and the model mainly targets low-level, C code. MMLite [6] is an operating system built using COM components. It provides limited support for dynamic reconfiguration

through the "mutation" mechanism, which enables the replacement of a component implementation at run-time.

DynamicTAO [9] and LegORB [16] are flexible ORBs that employ a dependency management architecture. This relies on a set of configurators that maintain dependencies among components and provide a set of hooks at which components can be attached or detached dynamically. OpenCOM supports a similar capability but it is an integrated part of the component model.

7 Conclusions

This paper has considered the design and implementation of OpenCOM, a lightweight and efficient reflective component model designed specifically for the development of middleware platforms. In other words, we exploit a component model for the construction of the middleware platform itself, which in turn provides an enhanced component model (for example, with intrinsic support for distribution) to application developers. The resultant middleware is more open and flexible, in terms of both configurability and reconfigurability.

Key features of OpenCOM include:

- the use of various styles of receptacles (single pointer, multi-pointer-with-context, multi-pointer) to make explicit the dependencies of components on their environment,
- the use of reflection to enable introspection of interfaces and receptacles and the associated component graph, as well as the dynamic insertion or deletion of interceptors, and
- backwards compatibility with the COM standard.

We have also demonstrated how OpenCOM can be used to construct a middleware platform, based on the related OpenORB architecture. In addition, it has been shown that the performance of the resultant system is on a par with established monolithic middleware platforms while simultaneously offering the benefits of componentisation and reflection introduced through the use of OpenCOM.

Acknowledgements. The research described in this paper is partly funded by the EPSRC together with BT Labs (through grant GR/M04242). The research is also partly financed by France Telecom R&D (CNET grant 96-1B-239). Finally, we would like to acknowledge the contributions of our partners on the CORBAng project (next generation CORBA) at UniK, and the Universities of Oslo and Tromsø (all in Norway).

References

[1] Blair G.S., Coulson G., Robin P. and Papathomas M., "An Architecture for Next Generation Middleware", Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Davies N.A.J., Raymond K. & Seitz J. (Eds.), The Lake District, UK, pp. 191-206, 15-18 September 1998.

[2] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K., "The Design and Implementation of OpenORB v2", To appear in IEEE DS Online, Special Issue on Reflective Middleware, 2001.
[3] Brown, K., "Building a Lightweight COM Interception Framework Part 1: The Universal Delegator", Microsoft Systems Journal, January 1999.
[4] Costa, F., Duran, H., Parlavantzas, N., Saikoski, K., Blair, G.S., and Coulson, G., "The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications". In Walter Cazzola, Robert J. Stroud and Francesco Tisato, editors, Reflection and Software Engineering, Lecture Notes in Computer Science 1826. Springer-Verlag, 2000
[5] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.
[6] J. Helander and A. Forin. "MMLite: A Highly Componentized System Architecture". In Proc. of the Eighth ACM SIGOPS European Workshop, pp 96-103, Sintra, Portugal, September 1998.
[7] Kiczales, G., des Rivières, J., and Bobrow, D.G., "The Art of the Metaobject Protocol", MIT Press, 1991.
[8] Knuth, D.E., "The Art of Computer Programming, Volume 1: Fundamental Algorithms", Second Edition, Reading, Massachusetts, USA, Addison Wesley, 1973.
[9] Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., and Campbell, R.H., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
[10] Microsoft, "The Component Object Model Specification", http://www.microsoft.com/com/resources/comdocs.asp. Last updated: 15/04/1999.
[11] Microsoft, COM Home Page, http://www.microsoft.com/com/default.asp. Last updated: 01/06/2000.
[12] Microsoft, .Net Home Page, http://www.microsoft.com/net. Last updated: 01/02/2001.
[13] Mozilla Organization, XPCOM project, 2001, http://www.mozilla.org/projects/xpcom
[14] Object Management Group, "CORBA Components" Final Submission, OMG Document orbos/99-02-05.
[15] A. Reid, M. Flatt, L. Stoller, J. Lepreau, E. Eide "Knit: Component Composition for Systems Software". In proceedings of 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), Usenix Association, pp. 347-360, October 2000.
[16] Roman, M., Mickunas, D., Kon, F., and Campbell, R.H., IFIP/ACM Middleware'2000 Workshop on Reflective Middleware. IBM Palisades Executive Conference Center, NY, April 2000.
[17] Rogerson, D., "Inside COM", Microsoft Press, Redmond, WA, 1997.
[18] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.
[19] Sun Microsystems, "Enterprise JavaBeans Specification Version 1.1", http://java.sun.com/products/ejb/index.html.

Appendix A: Essentials of Microsoft's Component Object Model

This appendix offers a brief overview of Microsoft's Component Object Model (COM) [Microsoft,99] that relates *to its use in OpenCOM*, i.e. using it's *in-process server* model (where components reside in the same address space). It is not intended to provide an exhaustive overview of the technology. In particular, we do not discuss aspects of COM's distribution mechanisms, i.e. its *local server* model (where

components reside in different address spaces but on the same machine) and its latterly introduced *remote server* model extension (where components reside on different machines) known as DCOM.

COM is underpinned by three fundamental concepts: i) uniquely identified and immutable interface specifications, ii) uniquely identified components that can implement multiple interfaces, and iii) a dynamic interface discovery mechanism. COM supports uniqueness through the use of 128 bit globally unique identifiers known as GUIDs, these are generated through the use of platform specific tools. The interface discovery mechanism is implemented through the notion of a special interface called *IUnknown* that must be implemented by every COM component. The purpose of IUnknown is actually twofold: *i*) it allows the dynamic querying of a component (*QueryInterface()* operation) to find out if it supports a given interface (in which case, a pointer to that interface is returned), and *ii*) it implements *reference counting* in terms of the number of clients using a components' interfaces. Reference counting is used to garbage collect components when they no longer have any clients.

Component and interface definitions are specified in Microsoft's language independent Interface Definition Language (MIDL) and then a tool (*midl*) is used to automatically generate language specific templates of these specifications for programmers to complete. *Midl* also generates files known as *type libraries* that efficiently embody all manner of type information related to components and their interfaces. Though initially intended to support dynamic method dispatch through late binding, these files include meta-information describing components and their interfaces that would otherwise not be available to a compiled language at runtime.

Importantly, the COM standard also defines the way in which components interoperate at the binary level. Primarily, this means in terms of the *vtable*; a C++ native data structure that mandates the way in which access to a components' interfaces is achieved. The vtable is effectively a per-component table of function pointers and any language that can support function pointers may natively interoperate with components written in C++. In addition, languages that do not support function pointers can still implement COM components if their support environments can be modified to export their functionality through function pointers. For instance, Java can implement COM components if the Java Virtual Machine (JVM) is modified to make its hosted Java classes available through a vtable. In addition to the use of the vtable to support binary compatibility, COM also mandates that all components must be compiled using the _stdcall calling convention (which essentially defines that each component method should clean the stack of its parameters before returning). This has important implications for our receptacle locking and interface interception architectures (see sections 3.1.2 and 3.2.3 respectively).

Finally, COM employs a system-wide repository known as the registry for locating component object files, type libraries, interface definitions etc. based on their GUIDs.

Rule-Based Transactional Object Migration over a Reflective Middleware

Damián Arregui, François Pacull, and Jutta Willamowski

Xerox Research Centre Europe
6, chemin de Maupertuis, 38240 Meylan, France
{Damian.Arregui, Francois.Pacull, Jutta.Willamowski}@xrce.xerox.com

Abstract. Object migration is an often overlooked topic in distributed object-oriented platforms. Most common solutions provide data serialization and code mobility across several hosts. But existing mechanisms fall short in ensuring consistency when migrating objects, or agents, involved in coordinated interactions with each other, possibly governed by a multi-phase protocol. We propose an object migration scheme addressing this issue, implemented on top of the Coordination Language Facility (CLF). It exploits the particular combination of features in CLF: the resource-based programming paradigm and the communication protocol integrating a negotiation and a transaction phase. We illustrate through examples how our migration mechanism goes beyond classical solutions. It can be fine-tuned to consider different requirements and settings, and thus be adapted to a variety of situations.

1 Introduction

Distributed systems use migration to perform load balancing, reduce network traffic and support mobile users. Both the operating systems and the mobile agents communities have extensively discussed the issues around process and agent migration. Current proposed solutions do not however cover all the requirements of today's enterprise distributed applications in domains such as electronic commerce, workflow, and process control. Indeed, such applications are often characterized by complex and dynamic relationships between distributed components. Suspending and later resuming this type of system (totally or partially) while preserving consistency becomes a real challenge. Nevertheless, maintenance operations on the underlying hardware and software infrastructures often require such operations to be performed. In order to ensure the continuous availability of the affected applications some of their components have to be transparently migrated from one node of the network to another.

In this article, we describe how the Coordination Language Facility (CLF) provides advanced support for object migration. It includes the externalization of migration control, and the reflexive use of the CLF middleware capabilities in order to offer transactional and negotiated migration. This enables flexibility and consistency.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 179–196, 2001.

CLF is a middleware platform aimed at coordinating distributed active software components over a Wide Area Network (typically the Internet). Mekano is a set of reusable coarse grain components and component development tools, compliant with the CLF middleware, that have been used in the implementation of various distributed applications deployed across multiple intranets. Complementary information can be found in [2].

Section 2 presents the basic functionalities of CLF used for object migration, section 3 describes the migration mechanism itself, section 4 gives examples of use, section 5 discusses related work, and section 6 concludes the paper.

2 Fundamental Features

We have implemented our approach for object migration on top of the CLF/Mekano platform. The migration facilities we describe could be implemented with other distributed object-oriented platforms but this would impose developing significant additional code in order to mimic some of the features readily provided by CLF. This section presents these features along with the minimum set of information needed to make this paper self contained.

2.1 Object Model

The CLF approach relies on the resource-based programming paradigm [5] that we have extended in order to cope with the requirements of a distributed system. We model objects as *resource managers*, the interactions between them as *transactional resource manipulations* (e.g. *removal* and *insertion* of resources in its simplest form) and the *resources* as tuples of strings. Each string element of

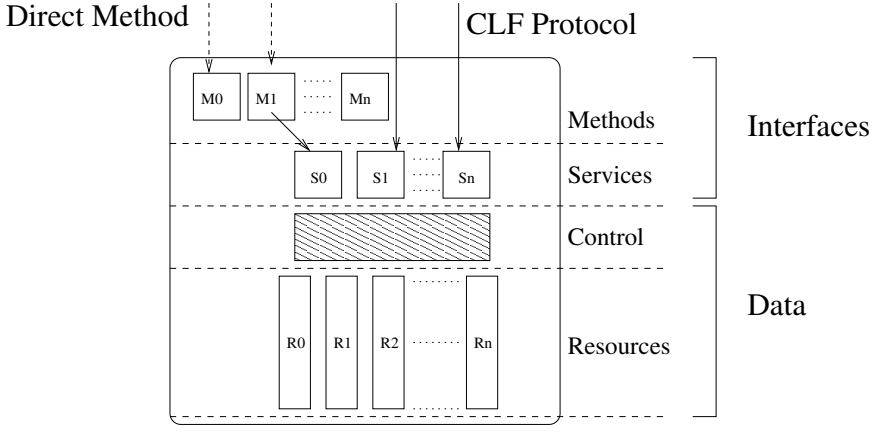


Fig. 1. CLF Object Model.

a resource can either contain text, string-encoded objects (e.g. XML-encoded) or marshalled objects (e.g. through Java [7] serialization).

The CLF object model goes beyond the classical dichotomy between *data* and *behavior*. Indeed, among the data itself we distinguish two separate kinds: the *control* data and the *resources* managed by the object (see figure 1). The control data covers information related to resource management while the resources represent the actual information contained in the object. For instance, the control data contains the structures (e.g. locks) to manage concurrent access to the object resources.

CLF enforces the traditional object encapsulation policy: resources are not accessible directly but only through an interface. But, unlike traditional objects, CLF objects offer two kinds of interfaces: *direct methods* and *services*. Direct methods correspond to traditional remote method invocations on a single CLF object while services allow the coordination of access to resources held by multiple CLF objects. Both are described in more detail in the two following sections.

2.2 Direct Methods

Direct methods typically provide user interface functionality for CLF objects, in particular for thin clients such as Web browsers. A direct method has the following abstract signature:

Perform: *input-method-parameters* -> *output-method-parameters*

Direct method invocations are similar to Remote Procedure Calls [14] available in conventional middleware, e.g. CORBA, and renamed "Remote Method Invocations" [8] in the Java world. Being HTTP-based, they can be invoked through a simple URL call, providing various encoding options for the input and output parameters. Initially they were designed to quickly add simple user interfaces to a CLF application using direct methods whose parameters and result encodings are directly supported by standard Web browsers (so called "form-data" or "url-encoding" for input parameters and HTML for the result). The now effective new generation of XML browsers and the standardization of various flavors of XML-RPC[15] makes this approach even more valuable.

Direct methods enable a *synchronous single-phase* interaction with a *single* CLF object. Services on the contrary allow to *transactionally access and consistently modify* the resources held by a *set* of CLF objects.

2.3 Services and Interaction Protocol

A service of a CLF object corresponds to a partial view of the resources the object manipulates. A three phase protocol deployed on top of the object services specifies how to access these services and how to manipulate the underlying resources. It allows in particular to coordinate the interaction within a set of CLF objects. This protocol consists of three phases: *negotiation*, *performance*, and *notification*. Each phase is in turn materialized through a set of corresponding interaction verbs that are invoked as described below.

First phase: negotiation. The negotiation phase asks a CLF service about offers for actions on resources matching a given filter. On such a request, formulated through the **Inquire** verb, the service returns a potentially infinite stream of offers. Therefore the service returns an `InquiryId` allowing the requester to access the corresponding offers one by one through the **Next** verb. For each offer, an `actionId` allows the identification, and to reference it later in the protocol. When no currently available resource matches the inquiry the **Next** operation remains pending until a new offer becomes available. The latter might happen either after internal changes within the object or through the insertion of new resources (see the notification phase). The service can also explicitly close the stream of offers by raising the NO-MORE-VALUE exception. This happens if the service can guarantee that no new offer matching the inquiry will ever become available. The requester can also **Kill** the inquiry if he is no longer interested in corresponding offers. Finally, the requester may, at any time, check the validity of an offer through the **Check** verb. The verbs involved in the negotiation phase have thus the following abstract signature:

Inquire: *input-service-parameters* -> *inquireId*
Next: *inquireId* -> <*output-service-parameters*, *actionId*> | NO-MORE-VALUE
Kill: *inquireId* -> VOID
Check: *actionId* -> YES | NO

Second phase: performance. The performance phase unrolls a classical two-phase commit protocol ensuring the atomic execution of a set of actions found during the negotiation phase. To achieve atomicity, the requester first attempts to **Reserve** the resources corresponding to these `actionId`s. If successful, it enacts all the actions through the **Commit** verb. Otherwise, if any reservation fails, it **Cancel**s all previously executed successful reservations. The verbs of the performance phase have the following abstract signature:

Reserve: *transactionId*, *actionId* -> ACCEPT | SOFT-REJECT | HARD-REJECT
Commit: *actionId* -> VOID
Cancel: *actionId* -> VOID

Third phase: notification. The notification phase allows for asynchronous creation of new resources. The verb **Insert** notifies their creation to the corresponding services:

Insert: *service-parameters* -> VOID

2.4 Scripting Language

The CLF scripting language exploits the CLF object model and services through the above described protocol. It views coordination as a complex block of interrelated manipulations of resources held by a set of objects (called the *participants* of the coordination).

CLF scripts describe, through rules, the expected global behavior of such blocks in terms of resulting resource manipulations, but abstracts away from the detailed sequencing of invocations of the CLF interaction verbs required to achieve such a behavior. This abstraction feature considerably simplifies the design and verification of coordination scripts. It makes them highly platform independent and hence, portable.

In a CLF application, dedicated CLF objects called *coordinators* enact the coordination scripts. As any CLF object, coordinators manage resources, accessible through CLF services: these resources are CLF coordination scripts and the rules which compose them. When a script is inserted in a coordinator, it is immediately enacted. Being CLF objects, coordinators can participate (i.e. occur as participants) in higher level coordinations, thus offering a reflexive model of coordination. Moreover, it is possible to create and insert scripts on the fly.

The CLF coordination scripting facility does not specify, per-se, any computing feature. If computation is needed in a coordination (arithmetic computation, string manipulation etc.), it must be handled by a participant. However, in the CLF distribution, a basic stateless computing facility is delivered with the coordinator prototype in order to provide simple computation, verification of assertions and timeout evaluation.

As shown later, we use the features and the power of this scripting language to implement our object migration mechanism.

2.5 Sample Script

The sample script detailed here is intended to show most of the CLF scripting language features used later in this paper. More applied script examples may be found in [3].

```
interfaces:
  ...
  YP(Obj): -> Obj is LOOKUP YellowPages.Objects
  apply(Obj,Serv,Y,Z): Obj,Serv,Y -> Z is DISPATCH

rules:
  S(X) @ P(X,Y) @ 'YP(Obj) @ apply(Obj, 'service',Y,Z) <>- R(X,Z)
```

The tokens `S`, `P`, `Q`, `R`, `YP`, `apply` refer to CLF services declared in the interface of some CLF objects (found and described in the name server, as shown later in section 2.7). For each token, the parameters appear between parentheses. The output parameters are underlined.

The logical name of the service can be statically defined (e.g. `YP` is linked to the service `Objects` of the `YellowPages` object) or dynamically according to the result of the instantiation of some parameters (e.g. the first two parameters of the token `apply` refer to the object and service name, the latter being known here only at run-time).

If such a rule is inserted in a coordinator, it is executed as follows:

1. Resources satisfying properties `S`, `P`, `Q`, `R`, `YP` are found:
   - The token `S(X)` finds some resource satisfying property `S`. The parameter `X` is instantiated accordingly using a value returned from the service that corresponds to `S`;
   - The second token finds a resource satisfying the property `P(X,Y)` for consistent values of `X`,`Y`.
   - The token `YP(Obj)` finds some resource satisfying property `YP`. The backquote (`) before the `YP` prevents the resource satisfying the `YP(Obj)` from being extracted in the transaction phase (if any).
   - The token `apply` is not statically linked to a logical name but will use the first two parameters as the object and the service name that will be used for the lookup. It is what we call the *dispatch* mechanism. The first one, `Obj`, is returned by the previous token `YP` while the second is set to the constant *`'service'`*. It finds consistent values of `Y`,`Z`.
2. For each tuple of consistent values `X`, `Y`, `Z`, `Obj`, the rule is triggered and transactionally extracts the resources satisfying `S`, `P` and `apply`.
3. A resource satisfying `R(X,Z)` is finally inserted.

2.6 Runtime

The CLF runtime provides the basic system facilities required for hosting and managing CLF objects on a site (See figure 2). Among these facilities we use the following for our migration mechanism:

- the managers (Search Manager, Concurrency Manager, Insert Manager) supporting the CLF protocol interaction for the objects, i.e. the asynchronous aspects of Inquiry/Next (SM), concurrency control and deadlock avoidance for Reserve/Cancel/Confirm (CM), event detection for Insert (IM), and garbage collection for Kill/Check (SM).
- the communication blocks encapsulating the communication protocol on top of which the CLF protocol verbs and the direct methods are implemented. These blocks allow for decoupling of the requests from the communication scheme (message passing, RPC) and the communication protocol (HTTP, SSL, XML-RPC, RMI).
- the system facilities allowing remote interaction with a CLF object. These facilities can allow for instance to start and stop an object within an application. They can also trigger the instantiation of a CLF object from a description or on the contrary to reify a CLF object into a description. The latter two cases are normally used for deploying and controlling an application. They are of particular interest in the context of object migration as described in section 3.

The CLF runtime facilities are, in general, directly invoked by various types of clients for deploying, monitoring and controlling CLF applications. However, we have encapsulated some of them in the services of a dedicated CLF object (see section 3.2) to be able to access and coordinate them through rule scripts.

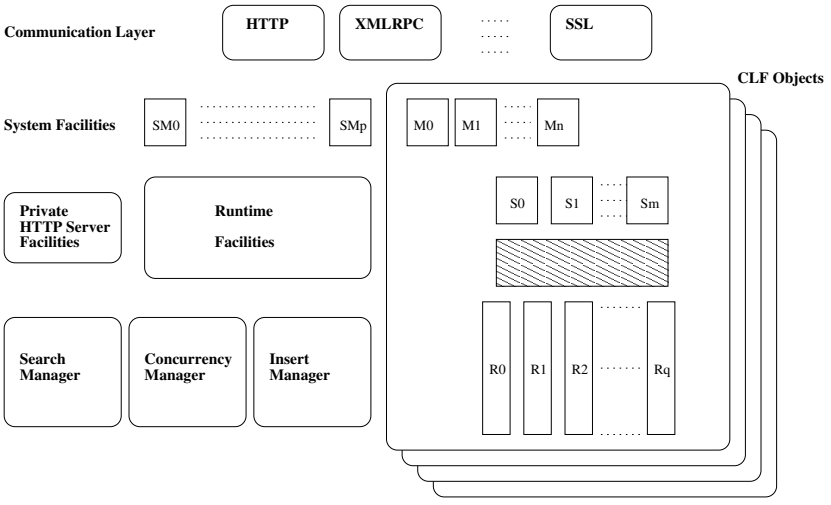


Fig. 2. CLF Runtime.

2.7 Name Server

The *name server* is a CLF object whose resources are *access patterns* binding *logical names* of objects to their *physical location*. CLF objects register themselves at the name server. The name server allows the coordinators to lookup the physical location of CLF objects from their logical names. This maps the logical description of the coordination contained in the rule scripts to the actual location of the CLF object service of an application distributed across several hosts.

Ensuring that the information contained in the name server is always accurate is of course essential to implement object migration. The possibility to access the resources manipulated by the name server through a CLF rule provides a simple way to modify them in a consistent manner during ongoing migration.

3 Object Migration Scheme

This section describes how the features presented in the previous section interoperate to provide the global CLF migration mechanism. This mechanism essentially relies on the reflexivity of the CLF platform. The basic idea is to first transform an active CLF component into a static resource, then transport it over the distributed system, just like any other resource, then instantiate it on the destination site.

This section first describes how a CLF object enters a freezable state where it can be safely and consistently transformed into a static resource. It then presents how we encapsulated the CLF runtime into a CLF object, the *object manager*,

providing services relevant to the migration scheme. We show in detail how these services are customized, i.e. how the verbs of the CLF protocol directly trigger the different methods readily provided by the runtime in order to realize the various migration steps. Finally, we illustrate through a very simple scenario how this allows us to manage object migration.

3.1 Reification and Freezable State

Migration requires the target object to enter a *freezable state*, where the data representing the current execution state of the object can be captured. Once this state is reached, we can easily transform the dynamic object into a static description, the *reified object*. From the reified object we can later re-instantiate the corresponding CLF object, possibly at a different location.

Freezable State Definition. Thanks to the clear distinction on one hand between control data (state of the inquiries and the transactions) and on the other hand the resources managed, we can easily define the freezable state of a CLF object as a state where the control data set is empty. In fact, as the resources are tuples of strings, they can be directly stored and recovered. So, for the freezable state we only have to consider the control data set: it contains information about the state of multi-phase interactions involving the object. In CLF, an empty control data set corresponds with the situation where no critical operation modifying the object resources is pending. In this case, the reified object (the set of resources together with the object description) entirely describes the object and can thus be used for object migration. This reified object is exactly the one used when starting objects at deployment time. Thus restarting objects after migration does not necessitate any specific additional code.

Freezing an object. As explained, to reach a freezable state, we have to flush the control data set. We now define in the following a process to guarantee that no new control data appear and that the control data set decreases and finally becomes empty. We therefore have to consider all interactions that modify the control data set of an object. The direct methods represent synchronous single-phase interactions which can only access the resources through the services. We can thus ignore them in this context. The CLF protocol however is a multi-phase interaction protocol that allows for the modification of the object resources. The control data set represents the state of this interaction. To reach a freezable state we therefore have to consider the different phases and verbs of the CLF protocol:

- The verbs belonging to the negotiation phase are *idempotent*: they do not modify the resources. Thus, even if the negotiation phase implements a multi-phase protocol: an Inquiry followed by a potentially infinite number of Nexts. It can be aborted without harm. Indeed, the coordinator will detect that the negotiation phase was aborted, in the same manner as if the object became

unavailable. In this case, the coordinator will restart the Inquiry when the object becomes available again, possibly at another location.
- The verbs belonging to the performance phase are *non-idempotent*: they can reserve and remove resources. They implement a two-phase commit protocol. The execution state of the protocol, in particular the reservation of a resource involved in a transaction, is stored in the object control data ; it might be complex to store and recover. Furthermore, by policy, CLF transactions are guaranteed to be short-lived. A reserved resource is either rapidly consumed through a Commit or released through a Cancel. Thus, the appropriate solution to reach a freezable state is to temporarily reject any new transaction and to wait until all ongoing transactions are completed.
- The notification phase only consists of the Insert verb and implements a synchronous one-phase interaction, that does not impact the control data set. Furthermore, it can be delayed without harm, as the coordinator will repeat pending insertions until they are acknowledged by the object. An object can therefore be frozen independently from pending notifications.

To sum up, the steps to freeze an object are:

1. break the connection for Inquiry/Next verbs simulating somehow a stopped object ;
2. return a *soft-reject* for any `Reserve`. A soft-reject tells the coordinator to retry the reservation later again if still required[1] ;
3. accept Commit/Cancel as usual ;
4. accept Insert invocations, but only until the control data set is empty. After that, apply the same treatment as for Inquiry/Next. Just like for the Inquiry/Next, the coordinator will retry the Insert invocations once the object becomes available again.
5. Once the object is freezable (i.e. the control data set is empty), we close the connection for any subsequent invocation, and freeze its state.

The CLF interaction model (services + direct methods) ensures that a freezable state will eventually be reached for every object.

3.2 Encapsulating the CLF Runtime

As described in the previous section the CLF runtime provides methods to start a CLF object and then to control it. Among these methods, the following ones are relevant for the migration process:

- `EnableObjectIsolation(objectname)`
  Triggers the processing of incoming invocations as described in the previous section for reaching a freezable state. At the same time the object is unregistered from the name server, preventing other useless subsequent lookups.

[1] In fact, the transaction is retried only if none of the other participants has returned a hard-reject implying the abortion of the transaction as a whole.

- `DisableObjectIsolation(objectname)`
  Resumes the normal management of incoming invocations and registers the object again on the name server.
- `isObjectFreezable(objectname) -> status`
  Returns a status, either *'ok'*, or *'notOk'*, meaning if the control data set is already empty or not. Once *'ok'*, this is a stable state as long as the isolation is not disabled.
- `ReifyObject(objectname) -> objectDescription,resourceSet`
  Returns the static description of the (frozen) object: object description (object type name and configuration parameters) and resources currently managed.
- `InstantiateObject(objectname,objectDescription)`
  Instantiates a new object according to the object description and registers it under the name `objectName` on the name server.

To benefit from the transactional semantics for object migration, and to fully take advantage of the reflexivity provided by CLF, we encapsulated these methods through a CLF object, the *object manager*. This object offers two CLF services: *reify* and *instantiate*. As we show in the following, these services are implemented in such a way that they call the above described methods, correctly mapping the migration process to the phases and verbs of the CLF protocol. Object migration in CLF is implemented as a coordination of the appropriate services of two object managers.

3.3 Service Reify

Reifying an object consists in transforming an *active object* into a *static piece of data*, typically a resource, that will later allow the re-activation of the corresponding object, possibly at another location. Reification is handled through the CLF object manager containing the object to reify.

The `Reify` service of a CLF object manager manages resources that are tuples of arity three. The first field corresponds with the logical name of the object to reify, the second one with object description (type name and configuration parameters), and the last one with its current set of resources. The values of the second and third fields are XML documents containing the description of the object: the set of modules corresponding to the objects static code, and the set of resources held by the object.

With respect to the CLF protocol described in section 2.3 the `Reify` service implements the following behavior:

- `Inquire`: On this verb the freezing process of the requested object is triggered. The < *objectname* > parameter, provided as input, denotes the object to reify. The second and third parameters will contain as output the reified form of the corresponding object. On such an Inquiry a new thread is launched, triggering the following reification process:

1. invoke the method `EnableObjectIsolation(`< *objectname* >`)` ;
2. loop until the method `IsObjectFreezable(`< *objectname* >`)` returns $'ok'$;
3. invoke the method `ReifyObject(`< *objectname* >`)` to obtain the reified form < *objectDescription* > and < *resourceSet* > of the object ;
4. insert through an internal API the resource `(`< *objectname* >`,` < *objectDescription* >`,`< *resourceSet* >`)` into the service enabling it to respond to a Next.

- `Next`: This verb blocks until the resource corresponding to the reified object becomes available as described above. Then, it returns an `actionId` to access and reserve this resource.
- `Kill`: This verb cancels the object freezing process associated with the `InquiryId` input parameter. The method `disableObjectIsolation` allows to stop this process and to restore the normal processing of external interactions.
- `Check`, `Reserve`: These verbs have their usual meaning with respect to the CLF protocol (see section 2.3).
- `Cancel`: This verb has the same effect as Kill, i.e. interrupting the freezing of the object.
- `Commit`: This verb destroys the frozen object definitively (within this object manager): it removes all remaining data allocated to the object.
- `Insert`: Invoking this verb raises an exception ; it has no sense with respect to the Reify service.

3.4 Service Instantiate

Instantiating an object consists of creating an active object from its reified form. This is handled through the Instantiate service of the target object manager.

The `Instantiate` service has three parameters identical to the Reify service but they are all input parameters.

The particularity of this service is that it can be used in two ways. Used on the right hand side of a rule, in the notification phase (verb Insert), it simply instantiates the object corresponding to the inserted resource. Used on the left hand side of a rule, in the negotiation and performance phases, it verifies possible preconditions for object reification (verb Reserve). As a consequence the whole object migration process can be cancelled and renegotiated as we will see in section 3.5.

With respect to the CLF protocol described in section 2.3 the Instantiate service implements the following behavior:

- `Inquire`: The given object name and reified description are associated with an `inquireId`. An associated resource corresponding to the given object name and the reified description is inserted into the service through an internal API.
- `Next`, `Cancel`: These verbs have their usual meaning in the CLF protocol (see section 2.3).

- `Kill`: This verb removes the corresponding resource created through the implementation of the Inquire verb from the service.
- `Reserve:` This verb checks in advance if the object can actually be instantiated from its reified form. This verification can have several facets. For instance, we can verify that all the required libraries or system resources are available at the target site. If any of these conditions are not verified then the associated object migration can be aborted and possibly renegotiated.
- `Commit`: This verb instantiates the object from its reified form. It removes the corresponding resource created through the implementation of the Inquire verb from the service. Once instantiated, the object registers itself under the given name on the name server.
- `Insert`: This verbs triggers the instantiation of an object from the given reified object description, without verifying any preconditions. Once instantiated, the object registers itself under the given name on the name server.

3.5 Migration Scripts

Object migration consists of three steps: object reification, transportation, and instantiation. As discussed in the previous section the CLF object managers provide services for object reification and instantiation. Coordination scripts initiate and handle object migration through these services. They achieve object transportation simply by passing the reified object description from the source object manager Reify service to the target object manager Instantiate service.

Migration scripts can be generated on the fly, either on user request or responding to a particular monitored run-time condition of the distributed system. Once inserted into the coordinator they are automatically enacted.

In the following, we show two elementary migration rules and will provide more practical examples in the next section. The first rule describes the unconditional migration of an object `obj1` from a source object manager `objMgr1` to a destination object manager `objMgr2`:

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) <>-
objMgr2.instantiate(''obj1'',objectDescription,resourceSet)
```

In this case the instantiation of `obj1` on `objMgr2` is taken for granted. Indeed, as the token appears on the right hand side of the rule, the instantiation is handled through the Insert verb of the Instantiate service. Thus no preconditions for instantiating the object on the target object manager are verified.

If preconditions have to be verified before migration, the `instantiate` token has to appear on the left hand side of the rule:

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) @
objMgr2.instantiate(''obj1'',objectDescription,resourceSet) <>-
```

The instantiation is now handled through the Inquire, Reserve, and Commit verbs of the Instantiate service, thus including the verification of basic preconditions for instantiating the object on the target object manager.

As we will see in the next section, several objects can be atomically migrated within a single transaction. Also, further constraints to be verified before commiting the migration can be integrated in the rule.

4 Examples of Use

As shown in the previous section, CLF scripts explicitly describe object reification, transportation, and instantiation and this, externally to the concerned objects. We now illustrate this further through different migration scripts.

4.1 Cloning Objects

Simple Cloning. The first example rule shows how to clone an object and to start the clone on another machine, within another object manager:

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) <>-
objMgr1.instantiate(''obj1'',objectDescription,resourceSet) @
objMgr2.instantiate(''obj2'',objectDescription,resourceSet)
```

Here the original object `obj1` is reified and re-instantiated on the original site `objMgr1` while the clone `obj2` is instantiated on a second site `objMgr2`.

Cloning and Load Balancing. Here, we go one step beyond and split the resources of the original object into two before cloning.

```
objMgr1.reify(''obj1'',objectDescription,resourceSet) @
split(resourceSet,part1,part2) <>-
objMgr1.instantiate(''obj1'',objectdescription,part1) @
objMgr2.instantiate(''obj2'',objectDescription,part2)
```

The token `split` appearing on the left hand side of the rule is implemented via a simple stateless computation service directly enacted by the coordinator (see section 2.4). On the right hand side two `instantiate` tokens instantiate the two clones with separate sets of resources on different object managers.

4.2 Contextual Migration

Our scripting approach also enables more complex migration schemes, allowing to trigger and manage migration transactionally. Contextual migration is an example. Indeed, transactions allow to verify a global context or condition among distributed components: within a transaction, the involved components can agree on the global distributed condition, thus triggering migration accordingly. Obviously, the considered context can be linked to the objects themselves, the application, the user, the distributed system or any combination of them.

As an illustration, consider the common situation where the system administrator of a machine starts a shutdown process with a countdown. This typically

warns other human users of the machine that the machine will be stopped. In reaction, they can stop their activities, save their work, and possibly log in to another machine or just wait until the machine is up again. However, for an autonomous software component, this situation is critical. Without the external help of a human, the shutdown will kill the component. This is an issue for a lot of applications that need to run continuously, e.g. knowledge management, workflow, electronic-commerce, or on-line trading for example.

The following rule allows us to manage the necessary migration process:

```
systemEvent(''shutdown'',objectMgrSrc) @
objectname(objectMgrSrc,''Objectname'',obj) @
reify(objectMgrSrc,''Reify'',obj,objectDescription,resourceSet) @
possibleDestination(objectDescription,objectMgrDst) @
instantiate(objectMgrDst,''Instantiate'',objectDescription,resourceSet) @
generateMsg(obj,''shutdown'',objectMgrSrc,objectMgrDst,msgTitle,msgBody)
<>-
email(''clf@xrce.xerox.com'',''appMgr@xrce.xerox.com'',msgTitle,msgBody)
```

Captured system events of type *shutdown* trigger this rule. As soon as a resource corresponding with such an event becomes available along with the involved `objectMgrSrc` (first token), we fetch all objects it hosts (2nd token). Here the use of the dispatch mechanism that fixes the object and the service name linked to the token `objectname` only at run-time (the token `systemEvent` instantiates the object manager variable `objectMgrSrc` and the service name is a constant *"objectname"*). For each object hosted by the concerned object manager, we initiate reification (3rd token, also a dispatch) and then fetch a possible destination (4th token). For each possible destination the token `instantiate` (dispatch) checks that instantiation is effectively possible. Finally, the last token builds a notification e-mail message through a simple computation service.

As soon as a global solution for all tokens on the left hand side of the rule is found, the respective operation of the services is performed, effectively migrating the corresponding object. The `email` token encapsulating a simple e-mail client, on the right hand side of the rule notifies the application administrator that the object has migrated. Note that the reified object is a single resource, meaning that as soon as it has been consumed, no other instantiation of the rule can be applied. This ensures that each object migrates only once.

4.3 Transactional Migration of a Set of Objects

Transactional migration of a set of objects at the same time is another interesting case easily covered by our approach. Consider, for instance, the case of a powerful server, hosting several dozens of objects, that has to be stopped. Here, we would like all objects to migrate to other hosts of the system. In this context several issues arise. First, as all the objects need to migrate at the same time, we have to ensure that the migration process does not lead to overloaded hosts, e.g. that the first possible host in the list of available ones receives all the migrating objects. Thus, a solution relying on an unconditional migration is not suitable.

Secondly, for performance issues we have to keep certain groups of objects collocated. Their migration has to be managed via a single rule ensuring that they either all migrate to the same site or none of them migrates (see section 3.5). In such cases, as the scripts are very specific to a given problem, they have to be dynamically generated. Dedicated rules detecting shutdown events can trigger their generation on the fly and insert the resulting rules into a coordinator enacting them.

5 Related Work

The migration problem in itself is not new and some existing systems provide mechanisms enabling process or object migration. In this section we compare our approach to three classes of systems providing similar facilities: operating systems, object-oriented systems, and mobile agent platforms.

5.1 Operating Systems

In the operating systems [9] community, process migration is a well-known research topic: it is used to support load balancing, increase reliability and avoid communication overhead by exploiting locality. Operating systems implement migration at a lower level than the one addressed in this paper. But the issues to solve and the decisions on design are similar.

A first problem is to capture the execution state of a process to suspend its execution and to later resume it on a different node. This task is usually complex. In our approach, the CLF resource-based programming paradigm and the associated object model allow the definition of a freezable execution state, as described in section 3.1.

A second issue is the control of the migration process. Depending on the intended use of the migration capabilities, different operating systems adopt different solutions on who controls this mechanism: the kernel, a manager process, the migrating process itself, the system administrator or any other user. In some systems [4], the source and the destination nodes can negotiate the migration, on the basis of the available resources. In our approach, as the CLF rules control the migration process (see section 4) we allow any entity of the system to create and enact such rules via the coordinator. Furthermore, with the CLF protocol, negotiation naturally comes into the picture.

A third major topic is to ensure the availability of the process environment and resources (in the operating system sense) after migrating to a different machine. We do not address this issue. Providing a generic solution to migrate the legacy systems, databases or Web services we encapsulate with CLF objects is beyond the scope of this paper.

Finally, the biggest issue concerns Inter-Process Communication, i.e. how to cope with communication channels which are open when migration is requested, messages which arrive during migration and the new, different process location after migration. The various adopted solutions include buffering messages and

handling the process location through name servers. In our approach, we first empty the control data set, i.e. the communication state, of the migrating object (see section 3.1). During migration, the coordinator buffers further interaction requests. Afterwards it tries to perform them again, automatically retrieving the new, updated location of the migrated object from the name server.

5.2 Object-Oriented Systems

The object-oriented paradigm seems particularly well-suited to migration [1]: it provides data encapsulation. The encapsulated data in principle directly represents the object state. It also standardizes the address scheme through global namespaces. Both features directly support object migration. Still, coherently stopping and migrating an active object engaged in interaction with several other objects remains difficult. With respect to this problem, our model goes one step beyond, distinguishing between control data and resources as described in section 2.

Some existing systems provide language-level support for migration [13]. This approach requires the addition of new keywords to the language, and hence to build new compilers. Besides this, it often requires additional code such as marshalling/unmarshalling routines and to explicitly define which objects can migrate. Our migration mechanism does not impose such constraints: within CLF a component developer does not need to do anything special to benefit from object migration.

5.3 Mobile Agents

In the mobile agents community migration is obviously a central issue. In fact, the aim is to optimize locality through agent mobility: rather than communicating with other agents remotely, a mobile agent decides to travel close to the agent or service with whom it wants to communicate and then communicates locally. Mobile agents are in general considered to be self-contained and autonomous. A typical example is the commercial agent traveling on behalf of a client from one electronic market place to another, negotiating each time with the locally available providers. This approach of mobility seems rather restricted, in contrast with the consistent migration of objects involved in multiple complex interactions with others, as provided by our approach.

The OMG "Mobile Agent Facility" specification [11] defines mobility aspects for agent systems. Roughly an agent can travel from one location to another having its class and state serialized and sent to the destination location. There the information is deserialized and the agent resumes execution according to its state. This specification contains no particular considerations about consistent migration. Voyager [10] and Aglets [6] for instance implement a corresponding mechanism. However, Voyager only guarantees consistent handling of "synchronized" methods for mobile agents. The programmer has to ensure that no other methods are active when an agent receives a travel request. With Voyager and Aglets the programmer has furthermore to ensure that all entities referenced

by a possibly mobile agent are serializable. Thus, just like in object-oriented systems, in mobile agents platforms the programmer has to provide specific additional implementation aspects needed for mobility. Within our approach, we reuse intrinsic mechanisms of the platform and thus, we can say that migration comes almost for free.

6 Conclusion

In CLF, as opposed to most existing systems, migration control is explicit. CLF rules, external to the migrating object(s), initiate and control the migration process. These rules can be very simple or rather complex depending on the migration conditions to express. This allows for high flexibility and control of the migration process at different levels:

- *initiating the migration*: the migrating object itself, the user, the application, or the underlying system.
- *triggering the migration*: the verification of local or distributed application and system conditions, the capture of external events, or predefined milestones.
- *negotiated migration*: migration can be simply imposed, or negotiated, e.g. between the source and the destination object managers. Also the destination of the migration process can be negotiated with a site broker.
- *transactional migration*: migration can concern only a single object or a set of objects as a whole.

In CLF the migration logic of an application is specified independently of the application logic itself. This allows systems-oriented developers to take care of the migration logic whereas application-oriented developers may design the application logic.

Our migration scheme goes beyond classical ones, in the sense that we do not migrate code but the description of an object that is used to re-create it at the destination. Thus, we consider several migration axes:

- *space*: an object travels across space in the underlying distributed system. This is classical migration.
- *versions*: an object travels across versions in the sense that it can be reified in one version and instantiated later on in an updated version. Moreover, the migration script could modify the object description on the fly, during migration.
- *environment*: an object can travel from one implementation language to another, e.g. initially running in a Python [12] virtual machine and then with the same resources transferred to a Java virtual machine. This is of particular interest when some components have to be migrated to portable devices imposing a particular system/language combination.

In order to complete our migration facilities, we plan to build tools (and in particular visual tools) to better harness the power of the CLF scripting language

for controlling object migration. The goal is to enable both application designers and system administrators to use higher level concepts to configure the migration mechanism.

Acknowledgements. The authors would like to thank Eric Forestier for his contribution in implementing an earlier migration mechanism on top of the CLF infrastructure, and Irene Maxwell for her helpful comments on the final version of this paper.

References

1. P. Amaral, C. Jacquemot, P. Jensen, R. Lea, and A. Mirowski". Transparent object migration in COOL2. In Yolande Berbers and Peter Dickman, editors, *Position Papers of the ECOOP '92 Workshop W2*, pages 72–77, 1992.
2. J-M. Andreoli, D. Arregui, F. Pacull, M. Riviere, J-Y. Vion-Dury, and J. Willamowski. CLF/Mekano: a framework for building virtual-enterprise applications. In *Proc. of EDOC'99*, Mannheim, Germany, 1999.
3. J-M. Andreoli, D. Pagani, F. Pacull, and R. Pareschi. Multiparty negotiation for dynamic distributed object services. *Journal of Science of Computer Programming*, 31(2–3):179–203, 1998.
4. Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9):47–58, 1989.
5. D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
6. IBM. Aglets. http://www.trl.ibm.com/aglets/.
7. Sun Microsystemns. Java. http://java.sun.com/.
8. Sun Microsystems. Java Remote Method Invocation specification. Technical report, Sun Microsystems, 1997.
9. M. Nuttall. Survey of systems providing process or object migration. Technical Report 94/10, Imperial College, London, UK, 1994.
10. ObjectSpace. Voyager. http://www.objectspace.com/products/voyager/.
11. OMG. Mobile Agent Facility specification. http://www.omg.org/cgi-bin/doc?formal/2000-01-02, January 2000.
12. Python. http://www.python.org/.
13. M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, pages 191–204, Nottingham (GB), July 1989. Cambridge University Society.
14. Sun Microsystems. RPC: Remote Procedure Call protocol specification. Technical Report RFC-1057, Sun Microsystems, Inc., June 1988.
15. XML-RPC. http://www.xmlrpc.org/.

The CORBA Activity Service Framework for Supporting Extended Transactions

Iain Houston[1], Mark C. Little[2,3], Ian Robinson[1], Santosh K. Shrivastava[3], and Stuart M. Wheater[2,3]

[1]IBM Hursley Laboratories, Hursley, UK
[2]HP-Arjuna Laboratories, Newcastle-Upon-Tyne,UK
[3]Department of Computing Science, Newcastle University, Newcastle-Upon-Tyne, UK

Abstract. Although it has long been realised that ACID transactions by themselves are not adequate for structuring long-lived applications and much research work has been done on developing specific extended transaction models, no middleware support for building extended transactions is currently available and the situation remains that a programmer often has to develop application specific mechanisms. The CORBA Activity Service Framework described in this paper is a way out of this situation. The design of the service is based on the insight that the various extended transaction models can be supported by providing a general purpose event signalling mechanism that can be programmed to enable activities - application specific units of computations – to coordinate each other in a manner prescribed by the model under consideration. The different extended transaction models can be mapped onto specific implementations of this framework permitting such transactions to span a network of systems connected indirectly by some distribution infrastructure. The framework described in this paper is an overview the OMG's *Additional Structuring Mechanisms for the OTS* standard now reaching completion. Through a number of examples the paper shows that the Framework has the flexibility to support a wide variety of extended transaction models. Although the framework is presented here in CORBA specific terms, the main ideas are sufficiently general, so that it should be possible to use them in conjunction with other middleware.

1 Introduction

Distributed objects plus ACID transactions provide a foundation for building high integrity business applications. The ACID properties of transactions (ACID: Atomicity, Consistency, Isolation, Durability) ensure that even in complex business applications the consistency of the application's state is preserved, despite concurrent accesses and failures. In addition, object-oriented design allows the design and implementation of applications that would otherwise be impractical. However, it has long been realised that ACID transactions by themselves are not adequate for structuring long-lived applications [1,2]. One well-known enhancement (supported by the CORBA Object Transaction Service, OTS [3]) is to permit *nesting* of transactions; furthermore, nested transactions could be concurrent. The outermost transaction of

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 197-215, 2001.

such a hierarchy is typically referred to as the top-level transaction. The durability property is only possessed by the top-level transaction, whereas the commits of nested transactions (subtransactions) are provisional upon the commit/abort of an enclosing transaction. This allows for failure confinement strategies, i.e., the failure of a subtransaction does not necessarily cause the failure of its enclosing transaction. Resources acquired within a subtransaction are inherited (retained) by parent transactions upon the commit of the subtransaction, and (assuming no failures) only released when the top-level transaction completes, i.e., they are retained for the duration of the top-level transaction.

The above enhancement is sufficient if an application function can be represented as a single top-level transaction. Frequently this is not the case. Top-level transactions are most suitably viewed as "short-lived" entities, performing stable state changes to the system [1]; they are less well suited for structuring "long-lived" application functions (e.g., running for hours, days, …). Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources for a long time; further, if such a transaction aborts, much valuable work already performed could be undone. In short, if an application is composed as a collection of transactions, then during run time, the entire activity representing the application in execution is frequently required to relax some of the ACID properties of the individual transactions. The entire activity can then be viewed as a non-ACID 'extended transaction'. The spheres of control model [4] describes the underlying concepts of recovery and commitment for extended transactions. Much research work has been done on developing specific extended transaction models [e.g., 5 - 8]. Nevertheless, most of the proposed techniques have not found any widespread usage; indeed, most commercial transaction processing systems do not even support nesting of transactions. One reason cited is lack of flexibility [9], in that the wide range of extended transaction models is indicative that a single model is not sufficient for all applications, so it would be inappropriate to 'hardwire' a specific extension mechanism. In any case, most transaction processing monitors are monolithic in structure, so difficult to extend. Thus the situation remains that a programmer often has to develop application specific mechanisms to build extended transactions.

There is a way out of this situation by exploiting a middleware based approach; in the case of CORBA for example, a set of open services are already available for building distributed applications. Within this context, it is appropriate to examine what additional functionality is required for flexible ways of composing an application using transactions, with the support for enabling the application to possess some or all ACID properties. The CORBA Activity Service Framework described in this paper provides such functionality through a set of structuring mechanisms to complement the OTS.

The design of the service is based on the insight that the various extended transaction models can be supported by providing a general purpose event signalling mechanism that can be programmed to enable activities - application specific units of computations – to coordinate each other in a manner prescribed by the extended transaction model under consideration. This has led to the development of an *Activity Service Framework* which we believe is sufficient to allow middleware to manage complex

business transactions that extend the concept of transaction from the well-understood, short-duration atomic transaction. The different extended transaction models can be mapped onto specific implementations of this framework permitting such transactions to span a network of systems connected indirectly by some distribution infrastructure. The framework described in this paper is an overview the OMG's *Additional Structuring Mechanisms for the OTS* standard [10] now reaching completion. The authors of this paper have been active in all phases this standardisation activity that included defining the scope of the RFP issued in early '99 [11] to making the initial submission and guiding it through to its present form [10]. Although the framework is presented here in CORBA specific terms, the main ideas are sufficiently general, so that it should be possible to use them in conjunction with other middleware.

2 Requirements and Approach

2.1 Requirements

We begin with some examples that illustrate that the need for non-ACID behaviour.
(i) bulletin board: posting and retrieving information from bulletin boards can be performed using transactions. While it is desirable for bulletin board operations to be structured as transactions, if these transactions are nested within other application transactions, then bulletin information can remain inaccessible for long times. Releasing of bulletin board resources early would therefore be desirable. Of course, if the application transaction aborts, it may be necessary to invoke compensating activities; this is consistent with the manner in which bulletin boards are used.
(ii) name server access: Consider the situation where persistent objects have been replicated for availability. The naming service needs to maintain up-to-date information about object replicas to enable clients to be bound to available replicas. For the sake of consistency it is desirable to structure lookup and update operations on the naming service as transactions. Application transactions, upon finding out that certain object replicas are unavailable can invoke operations to update the naming service database accordingly, while carrying on with the main computation [12]. There is no reason to undo these naming service updates should the application transaction subsequently aborts.
(iii) billing and accounting resource usage: if a service is accessed by a transaction and the user of the service is to be charged, then the charging information should not be recovered if the transaction aborts.

These applications share a common feature that as viewed by external users, in the event of successful execution (i.e., no machine failures or application-level exceptional responses which force transactions to rollback), the work performed possesses all ACID features of traditional transactional applications. If failures occur, however, non-ACID behavior is possible, typically resulting in non-serializability. For some applications, e.g., the name service example above, this does not result in application-level inconsistency, and no form of compensation for the failure is required. However, for other applications, e.g., the bulletin board, some form of compensation may be

required to restore the system to a consistent state from which it can then continue to operate.

(iv) Long-running business activity: Long-running activities can be structured as many independent, short-duration top-level transactions, to form a "logical" long-running transaction. This structuring allows an activity to acquire and use resources for only the required duration of this long-running transactional activity. This is illustrated in fig. 1, where an application activity (shown by the dotted ellipse) has been split into many different, coordinated, short-duration top-level transactions. Assume that the application activity is concerned with booking a taxi (*t1*), reserving a table at a restaurant (*t2*), reserving a seat at the theatre (*t3*), and then booking a room at a hotel (*t4*), and so on. If all of these operations were performed as a single transaction then resources acquired during *t1* would not be released until the top-level transaction has terminated. If subsequent activities *t2*, *t3* etc. do not require those resources, then they will be needlessly unavailable to other clients.

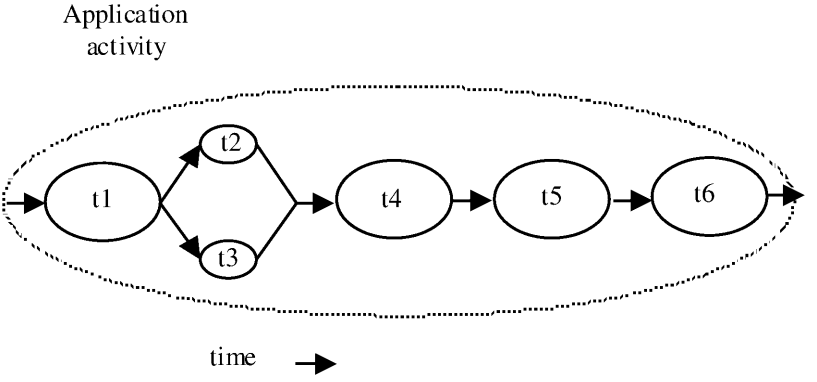


Fig. 1. An example of a logical long-running "transaction", without failure.

However, if failures and concurrent access occur during the lifetime of these individual transactional activities then the behaviour of the entire "logical long-running transaction" may not possess ACID properties. Therefore, some form of (application specific) compensation may be required to attempt to return the state of the system to (application specific) consistency. For example, let us assume that *t4* aborts (fig. 2). Further assume that the application can continue to make forward progress, but in order to do so must now undo some state changes made prior to the start of *t4* (by *t1*, *t2* or *t3*). Therefore, new activities are started; *tc1* which is a compensation activity that will attempt to undo state changes performed, by say *t2*, and *t3* which will continue the application once *tc1* has completed. *tc5'* and *tc6'* are new activities that continue after compensation, e.g., since it was not possible to reserve the theatre, restaurant and hotel, it is decided to book tickets at the cinema. Obviously other forms of transaction composition are possible.

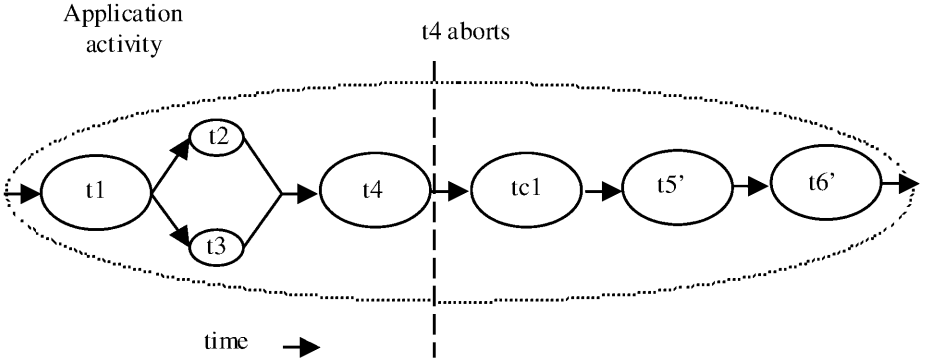


Fig. 2. An example of a logical long-running "transaction", with failure.

There are several ways in which some or all of the application requirements outlined above could be met. However, it is unrealistic to believe that the "one-size fits all" paradigm will suffice, i.e., a single approach to extended transactions is unlikely to be sufficient for all (or even the majority of) applications. Whereas in case of the last example, a transactional workflow system with scripting facilities for expressing the composition of the activity with compensation (a workflow) may be the most suitable approach, a less elaborate solution might be desirable for the first three examples.

2.2 Approach

As hinted earlier, the approach taken in the CORBA Activity Service is to provide a low-level infrastructure capable of supporting the coordination and control of abstract, application specific entities to enable construction of various forms of extended transaction models as desired by workflow engines, component management middleware and other systems. As we shall see, these entities (*activities*) may be transactional, they may use weaker forms of serializability, or they may not be transactional at all. The important point is that a computation is viewed as composed of one or more activities and the activity service is only concerned with their control and co-ordination, leaving the semantics of such activities to the application programmer. As is the case with other middleware standards, the Activity Service does not specify the implementation details of how the activities should be coordinated, only providing interfaces for coordination to occur.

An activity containing component activities may impose a requirement on the Activity Service implementation for managing these component activities. It must be determined whether these component activities worked as specified or failed or terminated exceptionally and how to map their completion to the enclosing activity's outcome. This is true whether the activities are strictly parallel, strictly sequential or some combination of the two. In general, an activity (or some entity acting on its behalf) that needs to co-ordinate the outcomes of component activities has to know what state each component activity is in:

- which are active
- which have completed and what their outcomes were
- which activities failed to complete

This knowledge needs to be related to its own eventual outcome. A responsible entity may be required to handle the sub-activity outcomes, and this can be modelled as an (distinguished) activity so that control flows can be made explicit. The activity determines the collective outcome in the light of the various outcomes its component activities present it with.

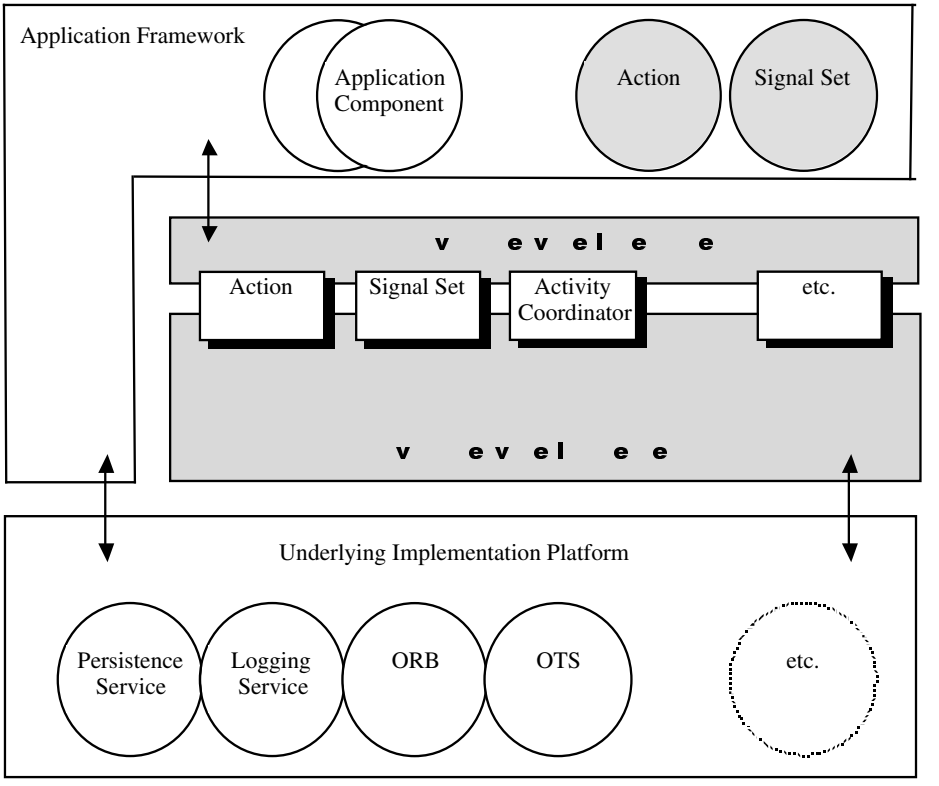


Fig. 3. The role of the Activity Service.

The activity service meets the above requirements in a very simple manner. Basically, associated with each activity is an *activity coordinator* that can coordinate the execution of constituent activities. In general, the coordination required can vary depending upon the phase of the execution of the activity (e.g., starting, terminating), so associated with a coordinator are one or more *signal sets*, each such set implementing a specific coordination protocol. For example, a signal set could implement a two phase commit protocol. Constituent activities are required to register themselves with a given signal set of the coordinating activity; this is done by an activity registering an *action* with the signal set. At an appropriate time, the coordinating activity triggers the execution protocol implemented by one of its signal set by invoking a standard opera-

tion; this leads to the set *signalling* each registered activity by invoking an operation on the registered action. The signalled activity can now perform some specific computation and return results (e.g., flush the data on to stable store and return 'done'), and this way the protocol advances. These aspects of activity coordination are discussed at length in the subsequent sections.

A very high level view of the role of the Activity Service is shown in fig. 3. It is not expected that the operations in the Activity Services interfaces will be used directly by end-user application programmers. When we talk about application programmers here we mean those who write for example, application framework for workflow managers or component management systems or who are extending the functionality of the Containers of Enterprise Java Beans (EJBs).

3 The Activity Service Framework

3.1 Activities

An *activity* is a unit of (distributed) work that may, or may not be transactional. During its lifetime an activity may have transactional and non-transactional periods. An activity may be composed of other activities. Each activity is represented by an *activity object*. An activity is *created*, made to *run*, and then *completed*. The result of a completed activity is its *outcome*, which can be used to determine subsequent flow of control to other activities. Activities can run over long periods of time and can thus be *suspended* and then *resumed* later.

Demarcation signals of any kind are communicated to registered entities (*actions*) through *signals*. For example, the termination of one activity may initiate the start/restart of other activities in a workflow-like environment. Signals can be used to infer a flow of control during the execution of an application. One of the keys to the extensibility of this framework is the *signal set* whose implemented behaviour is peculiar to the kind of extended transaction. The signal set is the entity that generates signals that are sent to actions and processes the results returned to determine which signal to send next. Similarly, the behaviour of an action will be peculiar to the extended transaction model of which it is a part. So as new types of extended transaction models emerge, so will new signal set instances and associated actions. This allows a single implementation of this framework to serve a large variety of extended transaction models, each with its own idea of extended transactions, each with its own action and signal set implementations. The *Activity Service implementation* will not need to know the behaviour which is encapsulated in the actions and signal sets it is given, merely interacting with their interfaces in an entirely uniform and transparent way.

3.2 Activity Coordination and Control

An activity may run for an arbitrary length of time, and may use atomic transactions at arbitrary points during its lifetime. For example, consider fig. 4, which shows a series

of connected activities co-operating during the lifetime of an application. The solid ellipses represent transaction boundaries, whereas the dotted ellipses are activity boundaries. Activity *A1* uses two top-level transactions during its execution, whereas *A2* uses none. Additionally, transactional activity *A3* has another transactional activity, *A3'* nested within it.

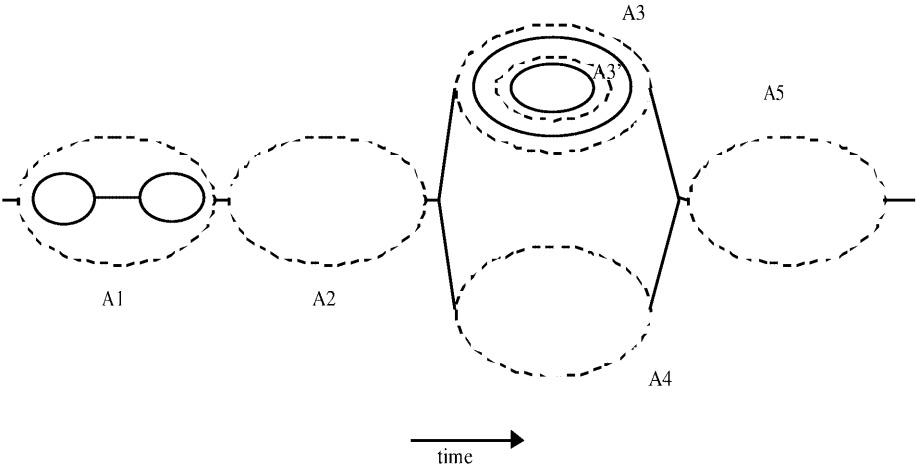


Fig. 4. Activity and transaction relationship.

3.2.1 Actions and Signals

An activity may decide to transmit activity specific data (*Signals*) to any number of other activities at specific times during its lifetime, e.g., when it terminates. The receiving activities may either have been running and are waiting for a specific Signal, or may be started by the receipt of the Signals. The information encoded within a Signal will depend upon the implementation of the extended transaction model and therefore the definition of the Signal is designed to accommodate this.

```
struct Signal
{
    string signal_name;
    string signal_set_name;
    any    application_specific_data;
};
```

To allow activities to be independent of the other activities, and also to allow the insertion of arbitrary coordination and control points, Signals are sent to *Actions*. An Action can then use the Signal in an application specific manner and return an indication of it having done so.

```
interface Action
{
    Outcome process_signal(in Signal sig)
                                    raises(ActionError);
};
```

3.2.2 SignalSets

To drive the Signal and Action interactions an *activity coordinator* is associated with each activity. Activities that require to be informed when another activity sends a specific Signal can register an appropriate Action with that activity's coordinator. When the activity sends a Signal (e.g., at termination time), the coordinator's role is to forward this signal to all registered Actions and to deal with the outcomes generated by the Actions.

```
interface SignalSet
{
   readonly attribute string signal_set_name;

   Signal get_signal (inout boolean lastSignal);
   Outcome get_outcome () raises(SignalSetActive);

   boolean set_response (in Outcome response,
                         out boolean nextSignal)
                           raises (SignalSetInactive);

   void set_completion_status (in CompletionStatus cs);
   CompletionStatus get_completion_status ();
};
```

The implementation of the coordinator will depend upon the type of extended transaction model being used. For example, if a Sagas type model [6] is in use then a compensation Signal may be required to be sent to Actions if a failure has happened, whereas a coordinator for a CA action model [13] may be required to send a Signal informing participants to perform exception resolution. Therefore, to enable the coordinator to be configurable for different transaction models, the coordinator delegates all Signal control to the SignalSet. Signals are associated with SignalSets and it is the SignalSet that generates the Signals the coordinator passes to each Action. The set of Signals a given SignalSet can generate may change from one use to another, for example based upon the current status of the Activity or the responses from Actions. The intelligence about which Signal to send to an Action is hidden within a SignalSet and may be as complex or as simple as is required. Importantly, a SignalSet is dynamically associated with an activity, and each activity can have a different SignalSet controlling it.

The activity coordinator therefore interacts with the SignalSet to obtain the Signal to send to registered Actions, and passes the results back to the SignalSet, which can collate them into a single result (fig. 5). When a Signal is sent to an Action, the SignalSet is informed of the result generated by that Action to receiving and acting upon that Signal; the SignalSet may then use that information when determining the nature of the next Signal to send. When a given Signal has been sent to all registered Actions the SignalSet will be asked by the coordinator for the next Signal to send.

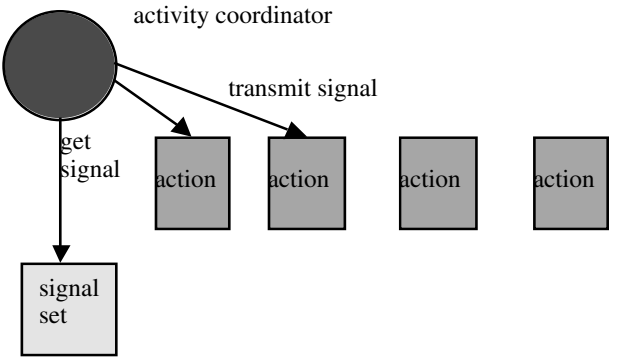


Fig. 5. Activity coordinator signalling actions.

Since it may not be possible to determine beforehand the set of Signals that will be generated by a SignalSet, Actions register interest in SignalSets, rather than specific Signals. Whenever a SignalSet generates any Signal, those Actions which have registered interest in that SignalSet will receive the Signal. An Action may register interest in more than one SignalSet and an activity may use more than one SignalSet during its lifetime (fig. 6).

As shown in fig. 7, a given SignalSet is assumed to implement a state machine, whereby it starts off in the *Waiting state* until it is required by the Activity Coordinator to send its first Signal, when it then either enters the *Get Signal state* or the *End state* if it has no Signals to send. Once in the *End state* the SignalSet cannot provide any further Signals and will not be reused. Once in the *Get Signal* state the SignalSet will be asked for a new Signal until it enters the *End state*. A new Signal is only requested from the SignalSet when all registered Actions have been sent the current Signal.

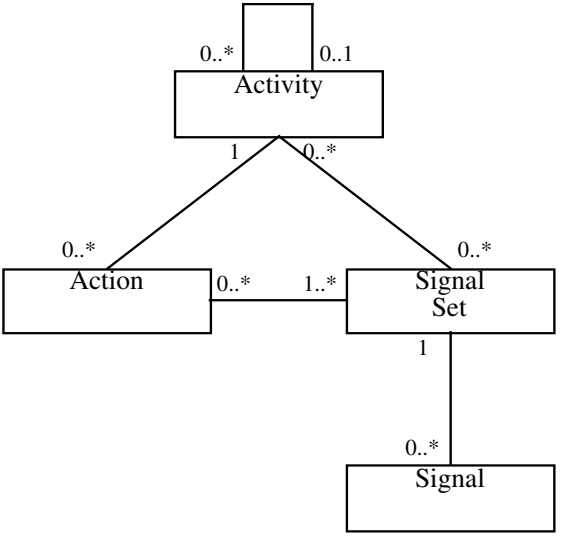


Fig. 6. Relationship of SignalSets, Signals, Actions and Activities.

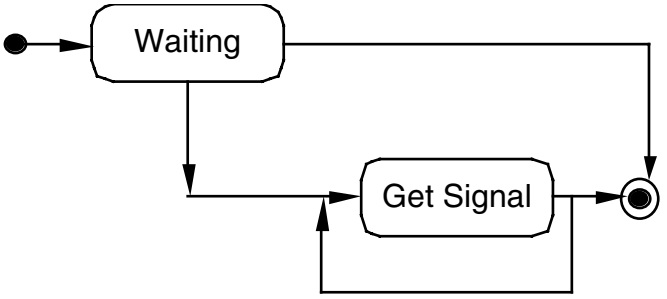


Fig. 7. SignalSet state transition diagram.

With the exception of some predefined Signals and SignalSets, the majority of Signals and SignalSets will be defined and provided by the higher-level applications that make use of this Activity Service framework. To use the generic framework provided within this specification it is necessary for these higher-level applications to impose application specific meanings upon Signals and SignalSets, i.e., to impose a structure on their abstract form. Illustrative examples are given in section 4.

3.2.3 Treatment of Failure and Recovery

The failure of an individual activity may produce application specific inconsistencies depending upon the type of activity.

- if the activity was involved within a transaction, then any state changes it may have been making when the failure occurred will eventually be recovered automatically by the transaction service.
- if the activity was not involved within a transaction, then application specific compensation may be required.
- an application that consisted of the (possibly parallel) execution of many activities (transactional or not) may still require some form of compensation to "recover" committed state changes made by prior activities. For example, the application shown in fig. 2.

Rather than distinguish between compensating and non-compensating activities, we consider that the compensation of the state changes made by an activity is simply the role of another activity. A compensating activity is simply performing further work on behalf of the application. Just as application programmers are expected to write "normal" activities, they will therefore also be required to write "compensating" activities, if such are needed. In general, it is only application programmers who possess sufficient information about the role of data within the application and how it has been manipulated over time to be able to compensate for the failure of activities. For example, suitable Actions may be created that compensate for work performed by an Activity, and triggered only if a specific SignalSet is used (see the example given in section 4.2).

Recovering applications after failures, such as machine crashes or network partitions, is an inherently complex problem: the states of objects in use prior to the failure may be corrupt, and the references to objects held by remote clients may be invalid. At a minimum, restoring an application after a failure may require making object states

consistent. The advantage of using transactions to control operations on persistent objects is that the transactions ensure the consistency of the objects, regardless of whether or not failures occur.

Rather than mandate a particular means by which objects should make themselves persistent, many transaction systems simply state the requirements they place on such objects if they are to be made recoverable, and leave it up to the object implementers to determine the best strategy for their object's persistence. The transaction system itself will have to make sufficient information persistent such that, in the event of a failure and subsequent recovery, it can tell these objects whether to commit any state changes or roll them back. However, it is typically not responsible for the application object's persistence.

In a similar way, we only state what the requirements are on such a service in the event of a failure, and leave it to individual implementers to determine their own recovery mechanisms. Unlike in a traditional transactional system, where crash recovery mechanisms are responsible for guaranteeing consistency of object data, the types of extended transaction applications we envision using this service will typically also require the ability to recover the activity structure that was present at the time of the failure. This will then enable the activity application to then progress onwards. However, it is not possible for the Activity Service to perform such complete recovery on its own; it will require the co-operation of the Transaction Service, the Activity Service and the application. Since it is the application logic that imposes meaning on Actions, Signals, and SignalSets in order to drive the activities to completion during normal (non-failure) execution, it is predominately this logic that is required to drive recovery and ensure activity components become consistent.

The recovery requirements imposed on the Activity Service and the applications that use it can be itemised as follows:

- *application logic*: the logic required to drive the activities during normal runtime will be required during recovery in order to drive any in-flight activities to application specific consistency. Since it is the application level that imposes meaning on Actions, Signals, and SignalSets, it is predominately the application that is responsible for driving recovery.
- *rebinding of the activity structure*: any references to objects within the activity structure which existed prior to the failure must be made valid after recovery.
- *application object consistency*: the states of all application objects must be returned to some form of application specific consistency after a failure.
- *recover actions and signal sets*: any Actions and SignalSets used to drive the activity application must be recovered.

Finally, a few words on the delivery of Signals. Minimally, the delivery semantics for Signals is required to be *at least once*, although implementations are free to provide better deliver guarantees. This means that an Action may receive the same Signal from an Activity multiple times, and must ensure that such invocations are idempotent, i.e., that multiple invocations of the same Signal to an Action are the same as a single invocation. Stronger delivery semantics - *exactly once* – can be provided by the activity service itself making use of the underlying transaction service.

4 Examples

In this section we describe how the Activity Service can be used to support a variety of coordination protocols, ranging from two-phase commit to workflow coordination.

4.1 Two-Phase Commit

We begin with a simple example illustrating how the Activity Service can be used to implement the classic transaction commit protocol; fig. 8 shows the exchanges involved when the transaction commits. The coordinating activity initiates commit by invoking get_signal operation of its 2PCSignalSet. The Set returns a 'prepare' signal that is sent to the first registered Action, whose response – done, rather than abort in this case - is communicated to the Set (operation set_response); the Set returns the prepare signal again that is then sent to the next registered Action and so forth.

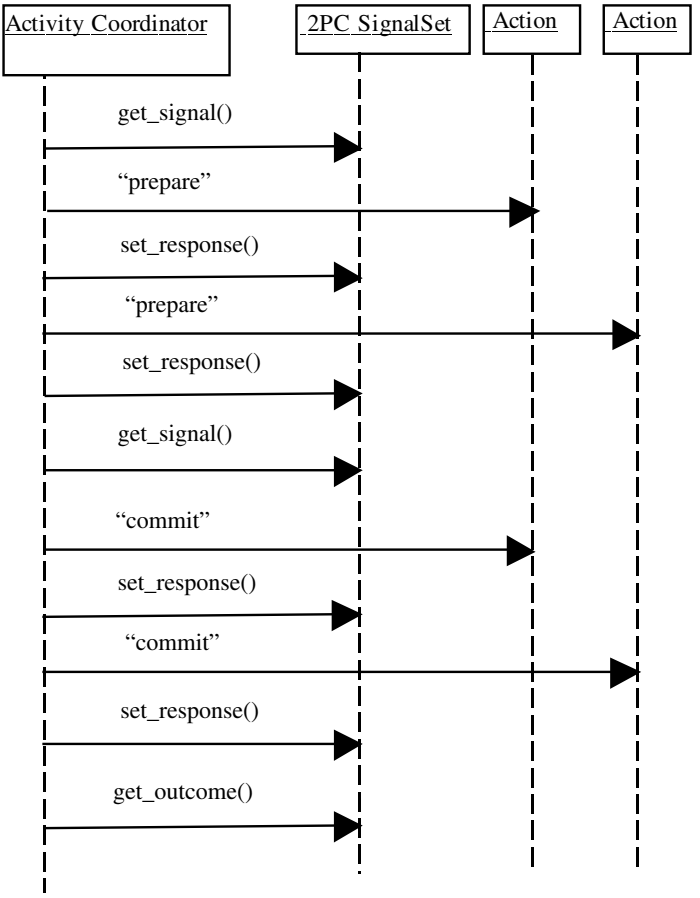


Fig. 8. Two-phase commit protocol with Signals, SignalSets and Actions.

4.2 Nested Top-Level Transactions with Compensations

We next illustrate how coordination of transactional activities with compensation for failures can be provided using the framework described. Consider the sequence of transactions shown in fig. 9, and assume as before that solid ellipses represent transaction boundaries and dotted ellipses represents an enclosing activity.

What we want to provide is the situation where within a top-level transaction (A), the application can start a new top-level transaction (B) that can commit or rollback independently of A. This scheme (also called open nested transactions [8]) can be useful if resources are required for only a short duration of the transaction A (as in the bulletin board example, section 2.1). If A subsequently commits then there is no problem with application consistency. However, if A rolls back, then it is possible that the work performed by B may be required to be undone (represented by transaction !B).

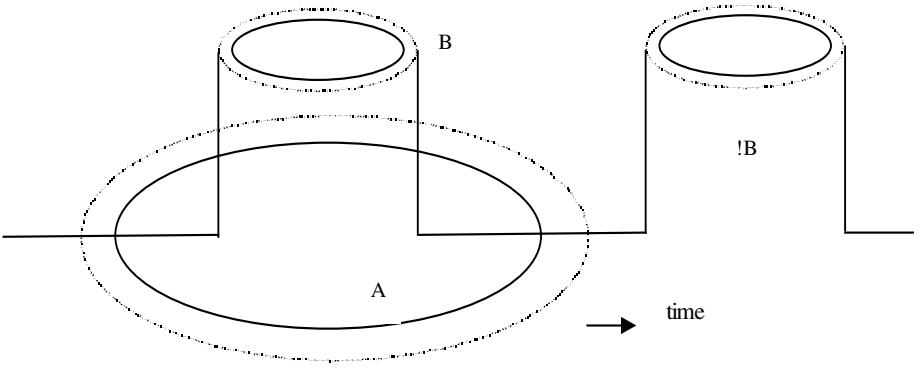


Fig. 9. Nested top-level transactions.

We make the following assumptions: (i) that each enclosing activity has a single SignalSet that is used when the activity completes (say, the CompletionSignalSet), and this SignalSet has Success, Failure and Propagate Signals, depending upon whether it completes successfully (and has no dependencies on other activities), completes abnormally (aborts), or completes successfully but has other activity dependencies, respectively; (ii) there is an Action that is responsible for starting !B if it receives the Failure Signal from an enclosing activity (say, the CompensationAction); the "state transitions" for the Action are:

- If it receives the Success Signal then it can remove itself from the system.
- If it receives the Propagate Signal, then encoded within this Signal will be the identity of an Activity it should register itself with. It must also remember that it has been propagated.
- If it receives the Failure Signal and it has never been propagated then it can remove itself from the system. If the Action has been propagated then it should start !B running, before removing itself.

Then the above structure can be obtained in the following manner:

- When transaction A's activity is begun, it registers with its coordinator the CompletionSignalSet as the one to use when the activity terminates. At this point no Actions are registered with that activity and hence with the SignalSet.
- When B is begun (and hence it's enclosing activity is also started), the activity registers the CompensationAction with B's activity, i.e., it's CompletionSignalSet.
- If B commits, the enclosing activity will terminate in the successful state, and the CompletionSignalSet will have the coordinator send the Propagate Signal to the registered CompensationAction. Encoded within this Signal will be the identity of the activity to propagate to, i.e., A. The CompensationAction can then enlist itself with A.
- If B rolls back, the enclosing activity will terminate in the failure state, and the CompensationAction will do nothing when it receives the Failure Signal.
- If A subsequently commits, it's enclosing activity's CompletionSignalSet will generate the Success Signal (since it has no dependencies on other activities), which will be delivered to the CompensationAction. In this case, no compensation is required, so the Action does nothing.
- On the other hand, if A subsequently rolls back, it's enclosing activity's CompletionSignalSet will generate the Failure Signal, and the CompensationAction will start !B to undo B.

4.3 LRUOW: Long Running Unit Of Work

The LRUOW model described in [14] is another extended transaction model to support long-running transactions. It combines some of the semantics of nested transactions and type-specific concurrency control; it relies on being able to execute long-running transactions in two phases: the *rehearsal phase*, where the work is performed without recourse to serializability and which may take an arbitrary amount of time and the *performance* phase, where the work is confirmed (committed) only if suitable locks and consistency criteria can be obtained on the data. In order to do this, it is necessary to have sufficient support from the resources used within the transactions, and to be able to specify operation predicates.

The LRUOW model could be implemented on the activity service infrastructure using a Rehearsal SignalSet and a Performance SignalSet. Each LRUOW resource could register a suitable Action with each SignalSet which would be driven when the activity completes. The higher-level API proposed in [14] would still be applicable, but would be mapped down to using these SignalSets and Actions. Each transaction would also be enclosed within an activity, which would be responsible for propagating resources from the child to the parent if the transaction completes successfully. This has the advantage that no modification to existing transaction systems would be required.

4.4 Workflow Coordination

Transactional workflow systems with scripting facilities for expressing the composition of an activity (a business process) offer a flexible way of building application specific extended transactions. Here we describe how the Activity Service Framework can be utilised for coordinating workflow activities. The SignalSet required to coordinate a business activity contains four signals, "start", "start_ack", "outcome" and "outcome_ack".

- *start*: signal is sent from a "parent" activity to a "child" activity, to indicate that the "child" activity should start. The *application_specific_data* part of the signal contains the information required to parameterise the starting of the activity.
- *start_ack*: signal is sent from a "child" activity to a "parent" activity, as the return part of a "start" signal, to acknowledge that the "child" activity has started.
- *outcome*: signal is sent from a "child" activity to a "parent" activity, to indicate that the "child" activity has completed. The *application_specific_data* part of the signal contains the information about the outcome of the activity, e.g., whether or not it completed successfully.
- *outcome_ack*: signal is sent from a "parent" activity to a "child" activity, as the return part of an "outcome" signal, to acknowledge that the "parent" activity has completed.

The interaction depicted in fig. 10 is activity *a* coordinating the parallel execution of *b* and *c* followed by *d*.

Referring to fig. 1 we can tie an activity to a single top-level transaction, such that when an activity begins (e.g., t1) it immediately starts a new transaction. A coordinating activity (implied by the dotted ellipse in the figure) would send appropriate "start" Signals, and wait for the "outcome" Signals to occur.

To do this, each potential activity registers an Action with a specific SignalSet at the coordinating activity (the parent); each activity that needs to be started for a specific event would register an Action with a specific SignalSet, e.g., t2 and t3 would register with the same SignalSet since they need to be started together, whereas t4 would be registered with a separate SignalSet.

Whenever a child activity is started the parent activity registers an Action with it that is used to deliver the "outcome" Signal to the parent. Let's assume that each child activity has a Completed SignalSet to facilitate this. When a child activity terminates, it uses the Completed SignalSet to send a Signal to the parent's registered Action. The content of this Signal will contain sufficient information for the parent to determine the outcome of the activity, and use this to control the flow of activities appropriately.

For example, in fig. 2, the parent activity would receive a successful termination outcome from t1, which would cause it to send "start" Signals to t2 and t3 via their registered Actions. When they both complete successfully (i.e., sent "outcome" Signals), it can then start t4. However, if t4 sends a failure outcome, or simply fails to send any outcome (e.g., it crashes), the parent activity can use this information to start tc1 in order to do the compensation.

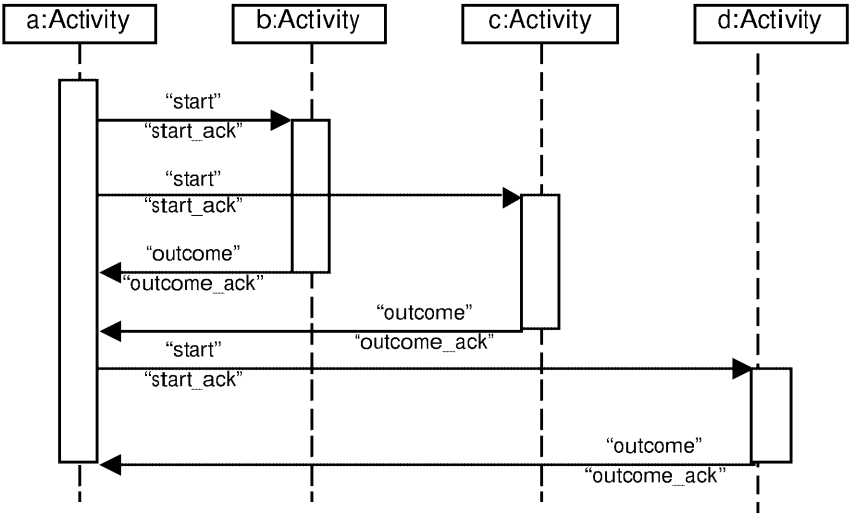


Fig. 10. Workflow coordination.

The task (i.e., activity) coordination scheme used in the OPENflow transactional workflow management system [15] is very similar to the above scheme. Here, associated with each task is a transactional *task controller* object. The purpose of a task controller is to receive notifications of outputs of other task controllers and use this information to determine when its associated task can be started. The task controller is also responsible for propagating notifications of outputs of its task to other interested task controllers.

5 Concluding Remarks

Although it has long been realised that ACID transactions by themselves are not adequate for structuring long-lived applications and much research work has been done on developing specific extended transaction models, no middleware support for building extended transactions is currently available and the situation remains that a programmer often has to develop application specific mechanisms. The CORBA Activity Service Framework described in this paper is a way out of this situation; it provides a general purpose event signalling mechanism that can be programmed to enable activities to coordinate each other in a manner prescribed by the model under consideration.

Through a number of examples we have shown that the Framework has the flexibility to support a wide variety of extended transaction models. The framework is deliberately designed to give a great deal of flexibility to higher-level services. For example, coordination points (places where the activity coordinator can communicate with its enrolled participants) may occur as many times as required and at arbitrary points during the activity's lifetime. Although all of the extended transaction models we present in this paper only require coordination at the end of the activity, others

(such as split transactions [16]) do not. In addition, we do not know what requirements will exist for future extended transaction models.

At the time of writing (July 2001), the specification on which this paper is based has just been adopted by the OMG. Work is also underway (through the Java Community Process) to incorporate the Activity Service approach to extended transactions in the next version of J2EE [17]. No implementations of the framework exist yet, although we know of several that are under development. As such, it is not possible to give a comparative evaluation of implementing a specific extended transaction model using this framework and implementing the model without it. This is a topic for further investigation.

Acknowledgements. Discussions with the partners involved in the development of the Standard is gratefully acknowledged; these include: Eric Newcomer (IONA Technologies), Malik Saheb (INRIA), Michel Ruffin (Alcatel), Shahzad Aslam-Mir (VERTEL/Expersoft) and Nhan T. Nguyen (Bank of America). The work at the Newcastle University was supported in part by a grant from IBM, Hursley.

References

[1] J. N. Gray, "The transaction concept: virtues and limitations", Proceedings of the 7th VLDB Conference, September 1981, pp. 144-154.

[2] D. J. Taylor, "How big can an atomic action be?", Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, January 1986, pp. 121-124.

[3] OMG, *CORBAservices: Common Object Services Specification*, Updated July 1997, OMG document formal/97-07-04.

[4] C. T. Davies, "Data processing spheres of control", IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

[5] A. K. Elmagarmid (ed), "Transaction models for advanced database applications", Morgan Kaufmann, 1992.

[6] H. Garcia-Molina and K. Salem, "Sagas", Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1987.

[7] S. K. Shrivastava and S. M. Wheater, "Implementing fault-tolerant distributed applications using objects and multi-coloured actions", Proc. of 10th Intl. Conf. on Distributed Computing Systems, ICDCS-10, Paris, June 1990, pp. 203-210.

[8] G.Weikum, H.J.Schek, "Concepts and Applications of Multilevel Transactions and Open Nested Transactions", in Database Transaction Models for Advanced Applications, ed. A.K. Elmagarmid, Morgan Kaufmann, 1992.

[9] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, "Advanced transaction models in workflow contexts", Proc. of 12th Intl. Conf. on Data Engineering, New Orleans, March 1996.

[10] OMG, *Additional Structuring Mechanisms for the OTS Specification*, September 2000, document orbos/2000-06-19.

[11] OMG, *Additional Structuring Mechanisms for the OTS,* RFP, May 1999, OMG document orbos/99-05-17.

[12] M. C. Little, D. McCue and S. K. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system", Proc. of 13th Intl. Conf. on Distributed Computing Systems, ICDCS-13, Pittsburgh, May 1993, pp. 491-498.

[13] J. Xu, A. Romanovsky and B. Randell, "Concurrent exception handling and resolution in distributed object systems", IEEE Trans. on Parallel and Distributed Systems, Vol. 11, No. 10, 2000.

[14] B. Bennett, B. Hahm, A. Leff, T. Mikalsen, K. Rasmus, J. Rayfield and I. Rouvellou, "A distributed object oriented framework to offer transactional support for long running business processes", Middleware 2000, (J. Sventek and G. Coulson eds.), LNCS 1795, pp. 331-348, 2000.

[15] S.M. Wheater, S.K. Shrivastava and F. Ranno, "A CORBA Compliant Transactional Workflow System for Internet Applications", Middleware 98, (N. Davies, K. Raymond, J. Seitz, eds.), Springer-Verlag, London, 1998, ISBN 1-85233-088-0, pp. 3-18.

[16] C. Pu, G. Kaiser and N. Hutchinson, "Split-transactions for open-ended activities", Proc. Of 14th Intl. Conf. On Very Large Data Bases, VLDB, Los Angeles, CA, September 1988.

[17] www.javasoft.com/aboutJava/communityprocess/jsr/jsr_095_activity.html

Failure Mode Analysis of CORBA Service Implementations

Eric Marsden and Jean-Charles Fabre

LAAS-CNRS, Toulouse, France
{emarsden,fabre}@laas.fr

Abstract. Before using middleware in critical systems, integrators need information on the robustness of the software. This includes having a clear idea of the failure modes of middleware candidates, including their core and side elements. In this paper we describe ongoing work on the failure mode analysis of CORBA-based middleware. Our initial work targets the CORBA Name Service, and the characterization is addressed using fault injection techniques. We present the results of injecting corrupted messages at the targeted middleware. Our experiments have been performed on four different implementations, and some comparisons are provided. First lessons learnt from these experiments, from a critical system integrator's viewpoint, are also reported.

1 Introduction

Middleware is being applied to an increasingly wide range of application domains, including critical applications such as Air Traffic Control systems, electronic interlocking systems in railways, and space platforms. In these application domains, dependability issues are of prime importance. The failure of these systems may put human lives at risk, or cause significant economic losses. However, there has been little published work on the dependability characterization of middleware. Information on the robustness of middleware implementations is useful from two points of view:

- Assisting system integrators in selecting the middleware candidate that is best suited to their requirements;
- Aiding middleware implementers by providing data on vulnerable points of their software, which may suggest alternative design or implementation techniques.

The analysis of system behaviour in the presence of faults is a crucial and complex issue. Faults can impact all system layers, and their effect may propagate from one layer to the other. The failure modes of a middleware can thus be very much dependent on the failure modes of the underlying operating system. In addition, the behaviour of the system depends on the types of faults to which it is exposed. These encompass design and physical faults. The interest of the results that can be obtained rely on the fault assumptions which are made

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 216–231, 2001.

for the experiments, and on the assumptions made concerning the behaviour of companion components in the system.

The long-term objective of this work is to address multiple facets of the characterization of a middleware implementation. Our first work presented here focuses on the characterization of the CORBA Naming Service, which constitutes a single point of failure in most CORBA-based systems.

The paper is organized as follows. In Section 2, we introduce terminology from the dependability community, and present the experimental dependability characterization technique that we have applied to the CORBA Naming Service. In Section 3, we describe the overall methodology, including the fault assumptions, the classification of the possible failure modes, and our experimental setup. We then summarize the experimental results obtained in Section 4, and discuss related work in Section 5. Section 6 presents some first lessons which can be learned from our work, from the viewpoint of a system integrator, and we draw the conclusions in Section 7.

2 Dependability and Failure Modes

The dependability of a system is defined as the ability to place a justified reliance in the service it delivers [1]. When a system no longer delivers the service that is expected of it, it is said to have failed. Failure occurs at the interface of the system with its environment, and results from the perception of an erroneous internal state. The hypothesized origin of this error is called a fault. Thus, there is a causal chain from fault to error to failure.

Users of a component need to reach a certain level of confidence in its ability correctly to deliver service. One process towards reaching this confidence is the analysis of the component's failure modes. An understanding of a component's behaviour in the presence of faults can be used by system integrators to decide on the mechanisms to be used to tolerate the remaining faults.

There are two main approaches to obtaining information on a component's failure modes: *(i)* analysis of error logs from a large number of operational systems, as in [2], or *(ii)* by simulating faults and observing the resulting behaviour. The first approach relies on error information obtained either from logs maintained by system administrators or from automatic surveillance mechanisms provided by the system. The former are often subjective, incomplete and not always available. The latter generally provide more structured information and thus are more appropriate for later analysis. However, faulty conditions are rare and thus obtaining a sufficient set of data is problematic, since the observation of a large population of identical systems during a long period is often impossible.

The second approach for failure mode characterization is mainly based on fault injection techniques. Fault injection experiments enable the robustness of the target component to be analyzed from different viewpoints, depending on the way the target component is corrupted. This approach can trigger the system's error detection mechanisms more frequently and also observe how it behaves when the error detection coverage is not 100%, as is usually the case for complex systems.

2.1 Fault Injection

Fault injection is a well-known dependability characterization technique [3], which studies a system's reaction to abnormal conditions. It is a testing approach that is complementary to analytical approaches, and which allows the examination of system states which would not be reached by conventional functional testing. The aim of fault injection is to simulate the effect of real faults impacting a target system, namely the error due to the activation of a fault.

A number of fault injection techniques have been developed. Physical fault injection (e.g., heavy-ion and electromagnetic perturbations, pin-level fault injection) was the initial type of technique used (stuck-at-1, stuck-at-0, bit-flips) [4]. Due to the complexity and the speed of modern integrated circuits, Software-Implemented Fault Injection (SWIFI) techniques are often preferred. In this technique, the corruption is performed by a software tool, and can target different components or different layers in a system (operating system kernel, system services, middleware components). This approach is very generic and flexible, since a large variety of fault models can be used. Several studies have shown that a single bit-flip leads to similar errors to those produced by physical fault injection techniques (e.g., [5,6]), and also that they simulate errors produced by software faults [7] fairly faithfully.

The target for the fault injection can either be the interface of a software component, or its internal address space. Targeting the interface assesses the component's *robustness*, its ability to function correctly in the presence of invalid inputs and stressful environmental conditions. It is a useful way of evaluating the probability of error propagation from one system component to another. Targeting the address space assesses the impact on the component's behaviour of internal corruptions, resulting either from physical faults or a software faults.

An ideal component would be able immediately to detect the effects of a fault. Its internal error detection mechanisms (such as behavioural checks and executable assertions) would detect the incorrect inputs or the perturbation of the component's state or behaviour, and signal the problem to interacting components. In practice, this is not always the case, particularly with off-the-shelf components where dependability may not be an important criterion. The outputs of a fault injection campaign are thus interesting to *(i)* discover weak or missing error detection mechanisms and to *(ii)* identify the unexpected behaviour that must be taken into account at upper layers of the system. Experimental failure mode analysis is thus an input for the design of error confinement wrappers and fault tolerance mechanisms. This last statement shows the important place of fault injection experiments in the design and the implementation of fault tolerant systems.

Several factors must be considered before launching a fault injection campaign:

- the fault model: which classes of errors to insert, where to insert them, and when? The injection may be triggered by the occurrence of an event of interest, or occur after a predetermined time period. The fault may be transient in nature (e.g, a single bit-flip), or permanent (e.g., a bit remains stuck at zero).

- the observations: how to detect and classify the failure modes? This can be delicate in a distributed system, where failures can be partial.
- the workload: what operational profile or simulated system activity should be applied during the experiment?

There is a considerable body of work applying fault injection for the characterization of operating systems (such as the Ballista work on robustness testing of POSIX-compliant operating systems [8]), and real-time kernels (such as [9], which assesses the behaviour of real-time microkernels in the presence of both external and internal faults). Fault injection has also been applied to network protocol stacks, for example [10] on the characterization of TCP stacks in Unix operating systems. However, there has been little published work targeting middleware. In the next section we describe the methodology we have followed for our fault injection experiments targeting the CORBA Name Service.

3 Methodology

In this section, we present the experimental target that we have chosen for our fault injection experiments, and describe the fault model that we have used. We then explain the classification of failure modes that we have chosen, and describe our experimental setup.

3.1 Experimental Target

Our dependability characterization work has so far concentrated on middleware services, specifically the CORBA Name Service [11]. This service provides a hierarchical directory for object references, allowing server applications to register a service under a symbolic name, and clients to obtain references by resolving a name.

We chose this target since its standardized interface makes it easy to compare different implementations of the service. Furthermore, the Name Service may constitute a single point of failure in a CORBA-based system: while it is possible to deploy applications without using a naming or trading service, by allocating object references statically, most systems require the dynamicity provided by this service.

We believe that the failure modes exhibited by a vendor's implementation of the name service will also be present, to a significant extent, in other applications built using the vendor's CORBA ORB. Indeed, a vendor's name service implementation is typically composed of a certain amount of application code implementing the service-specific functionality, which is linked with the vendor's shared library implementing its ORB. A significant proportion of the robustness failings we have observed are relatively low level, and thus more likely to come from the ORB library than from the application code; we would therefore expect that they will also be present in other applications using the ORB.

3.2 Fault Model

Middleware-based systems are more complicated from an architectural point of view than centralized systems, and are consequently exposed to a wider range of classes of faults:

- physical faults affecting RAM or the processor's registers (so-called Single Event Upsets or soft errors [12]), for example bitflips due to cosmic radiation.
- software faults (design or programming errors) at the application, middleware and operating system levels. For instance, an application may pass a NULL pointer to the middleware, or the middleware may omit checking of error codes returned by the operating system. Previous work described in Section 5 has investigated the impact of this class of faults.
- "environmental" faults, such as the interruption of network connections and disk-full conditions. Exhaustion of resources, such as RAM (due to memory management problems, common in CORBA applications) and file descriptors, arises from "process aging". These conditions will generally be signalled to the middleware by the operating system; a robust implementation should propagate the condition gracefully to the application level, typically by raising an exception.
- communication errors, such as message loss, duplication, reordering or corruption. While this class of fault is widely assumed not to affect middleware that builds on a reliable network transport protocol, as is the case of CORBA's IIOP, recent research discussed below suggests that these classes of error deserve attention.

Our work investigates the impact of the last category of faults: corrupt method invocations arriving over the network. Our experiments consist of sending a corrupted request to the target, and observing its behaviour. This fault model simulates three different classes of faults:

- transient physical faults in the communication subsystem, resulting for example from faulty memory banks in routers, or faulty DMA transfers with the network interface card.
  Network corruption, even over reliable transport protocols such as TCP (on which IIOP is based), is more frequent than is commonly assumed. Based on analysis of traffic traces on a LAN and the Internet, [13] reports that approximately one packet in 32000 fails the TCP checksum, and that between one in a few millions and one in 10 billion packets are delivered corrupted to the application level. This is because the 16-bit checksum used in TCP is not able to detect certain errors. While this proportion is very small, it is non-negligible given the high capacity of modern LANs.
- propagation to the target of a fault that occurred on a remote machine interacting with the target. The fault may have affected the remote operating system kernel, its protocol stack implementation, or the remote ORB, leading to the emission of a corrupted request.

– malicious faults, such as denial of service attacks against the target[1]. Given the pivotal rôle of the name service in most CORBA-based systems, an attacker who can crash the service may be able to cause the entire system to fail. We note, however, that most CORBA systems will be deployed on private networks where all parties can be assumed to be trustworthy.

A general characteristic of fault injection is that the errors which are provoked can simulate the effects of various types of faults: physical, design and implementation, which affect different parts of the system. Different faults often lead to similar classes of errors. We wish to reproduce these errors to assess their impact at the middleware level.

The types of errors we have investigated are single bitflips and the zeroing of two successive bytes in a message. These are among the most common patterns of corruption identified in [13], and we presume that they are representative of error propagation from remote machines.

There are several possible means of injecting these faults. We could use dedicated network hardware, but this is cumbersome and expensive. Using software-implemented fault injection, we could inject faults at the protocol transport layer (for example by instrumenting the operating system's network stack, as in [14]). However, this form of corruption has a very high probability of being detected by the remote host's network stack, and therefore of not being delivered to the middleware. Consequently, we choose to inject the fault at the application level, before the data is encapsulated by the transport layer. In this way we simulate the proportion of corrupt packets that TCP incorrectly delivers as being valid.

3.3 Failure Modes

Although the classification of failure modes may depend on the target component, the various possible outcomes of a component's behaviour in the presence of faults are similar. Roughly speaking, either the fault is successfully detected by various error detection mechanisms (behavioural checks, executable assertions, hardware mechanisms, etc.) and signalled by different means (error status, exceptions, interrupts, etc.) to the interacting components, or it is not.

The latter case is the more difficult to classify. The first possible situation is the crash or the hang of the target component. Observing this situation involves external mechanisms that control the liveness of the component under test. When no crash or hang are observed, then more subtle mechanisms must be used to distinguish the correct outcomes of the target. In testing, this is known as the notion of *oracle*. This oracle must be defined beforehand and is part of both the activation profile of the component under test and the fault injection campaign at runtime. Indeed, during a test experiment, the outputs of the component must

[1] However, our work is not a realistic study of malicious faults. The errors we simulate in our experiments are unlikely to be representative of the activities of a malicious user of the system, who is likely to employ more sophisticated sequences of unexpected inputs than single bitflips. Robustness is a necessary, but not sufficient, characteristic of survivable systems.

be obtained to be compared (at the end) to the oracle. This is the only way to detect incorrect behaviour of the target component during the test phase, when built-in error detection mechanisms fail.

We classify the experimental outcomes for injections targeting the name service as follows:

- Kernel crash: the machine hosting the service becomes inaccessible from the network. We test for this condition by attempting to execute a command from a remote machine.
- Service crash: attempts to establish a network connection to the service are refused. Typically this means that the process implementing the service has died.
- Service hang: the service accepts the incoming connection, but does not reply within a given time span. Note that this does not necessarily mean that other clients of the service are blocked, since processing may continue in other threads.
- Application failure (error propagation to the application level): the service starts returning erroneous results to clients. We assume conservatively that error propagation to the application causes an application failure.
- Exception: an invocation of the service results in a CORBA exception being raised. We distinguish between System Exceptions (which come from the ORB) and User Exceptions (which are raised at the application level).

These failure modes are not exclusive: for example a service crash will generally result in clients of the service receiving an exception indicating that a communication error has occurred. In the results we provide below, each experiment is classified according to the most serious failure mode observed by the testbed.

The observation of these failure modes is a crucial issue in a fault injection campaign. It is difficult to achieve 100% coverage of the error detection mechanisms, so some failures may be undetected. In particular, since all fault injection experiments are finite in time, it is possible for an injected fault not to lead to any observable effect during the duration of the experiment. This does not necessarily mean that the fault has no impact, since its effect may be postponed (notion of error latency).

These failure modes are not equivalent from a dependability point of view. Signalling an exception is the "best" experimental outcome, since the service remains available to other users, and the application can decide on the most appropriate recovery action, such as retrying the operation (in the case of a `TRANSIENT` exception) or deciding to use an alternative service (for `COMM_FAILURE`). It is important that the exception provide as much information as possible: `COMM_FAILURE` is more useful than `UNKNOWN`, since in the latter case the application has less information on which to base its recovery strategy.

The most serious failure mode is error propagation to the application level: indeed, any fault tolerance mechanisms implemented at the application level will not be activated, and the error is free to propagate to the system's service interface. The kernel and service crash, and service hang failure modes, while

not positive outcomes, are considered less serious, since they can be detected and masked by system-dependent mechanisms such as watchdog timers.

3.4 Experimental Setup

The infrastructure we use to support our fault injection experiments consists of the following components:

- the workload application, which activates the target service's functionality (the workload runs on a different machine from the service);
- the fault injector, which sends a corrupted request to the target once the workload has been running for a certain time span;
- monitoring components, which observe the behaviour of the target and log their observations to an SQL database;
- offline data analysis tools.

Our workload application repeatedly constructs a naming graph, resolves names against this graph, and then destroys the graph. Since the graph is built up in a deterministic way, the workload is able to check that the results returned by the service are correct (it plays the rôle of oracle with respect to the functional specification of the service). If the workload detects an anomaly in the operation of the target service, such as an incorrect result, this is signalled as an application failure. If it receives an exception from the target, it signals the appropriate exception outcome.

Each experiment corresponds to a single injected fault. A controller process launches the target service and obtains its object reference (in the implementations which we have targeted, the name service is implemented as a Unix dæmon). It then starts the workload application, passing it the service's reference. After 20 seconds, the fault injector sends a corrupted `resolve` request to the target service (for a name which has not been given a binding) and waits for the reply. The expected reply is a `NotFound` exception raised by the naming service. If no reply arrives within 20 seconds, a ServiceHang failure mode is signalled. At the end of the experiment, the monitoring components check for the presence of the different failure modes by trying to launch a command on the target host, checking for returned exceptions, etc.

For each target implementation, a fault injection campaign involves running an experiment for each bit or byte position in the `resolve` request. A campaign lasts around 48 hours per target for the bitflip fault model.

4 Experimental Results

4.1 Target Implementations

We have carried out our experiments on four implementations of the CORBA Name Service:

- *omniORB* 2.8, by AT&T Laboratories, Cambridge. Freely available under the GNU General Public Licence. Implemented in C++.

- *ORBit* 0.5.0, also available under the GNU General Public Licence, and implemented in C.
- *ORBacus* 4.0.4, a commercial product from *Object Oriented Concepts*, implemented in C++.
- the `tnameserv` bundled with version 1.3 of Sun's Java SDK.

All experiments were carried out on workstations running the Solaris 2.7 operating system, connected by a 100Mb/s Ethernet LAN. While we tried to make the experimental conditions as similar as possible across experiments, a number of factors require particular attention:

- persistency: the *omniORB* implementation maintains log files so as to provide persistency across service shutdowns. To ensure a fresh environment for each experiment, we erase these log files before starting the service. The *ORBacus* implementation can be configured to use log files, but we do not enable them in our experiments. The two other tested implementations do not support persistency.
- number of experiments: as mentioned earlier, we perform experiments for each bit or byte position in the corrupted method invocation. CORBA method invocations contain an ORB-dependent parameter called the service context (which can be used to propagate implementation-specific data and implicitly propagate transactions). The size of this parameter differs slightly between ORB implementations, so the exact number of experiments changes slightly from target to target.
- the *ORBit* implementation defaults to using non-interoperable object references. We configured it to use standard IIOP profiles.
- we do not reboot the machines after each experiment (this is justified by the fact that we have not observed any host crashes).

In certain experiments, we observe several failure modes: for example a SystemException coupled with a ServiceCrash. In the figures presented below, the failure modes are classified according to gravity, and for each experiment the most serious mode is selected.

4.2 Analysis of Results

In this section we present the results of our fault injection experiments, for both the double-zero and bitflip fault models. More general analysis from a dependable system integrator's perspective is presented in Section 6.

Figure 1 compares the experimental outcomes for each target implementation, for the double-zero fault model. The outcomes whose names in the legend are in capital letters correspond to CORBA SystemExceptions. The `NotFound` outcome is a CORBA application-level exception raised by the naming service when it cannot resolve a name; this is the expected behaviour of the service for our experiments. The sum of the vertical bars for each target is 100%.

A first remark is that we have not observed any cases of error propagation to the application level, which is a positive point. However, there are a relatively large proportion of service hangs and crashes.

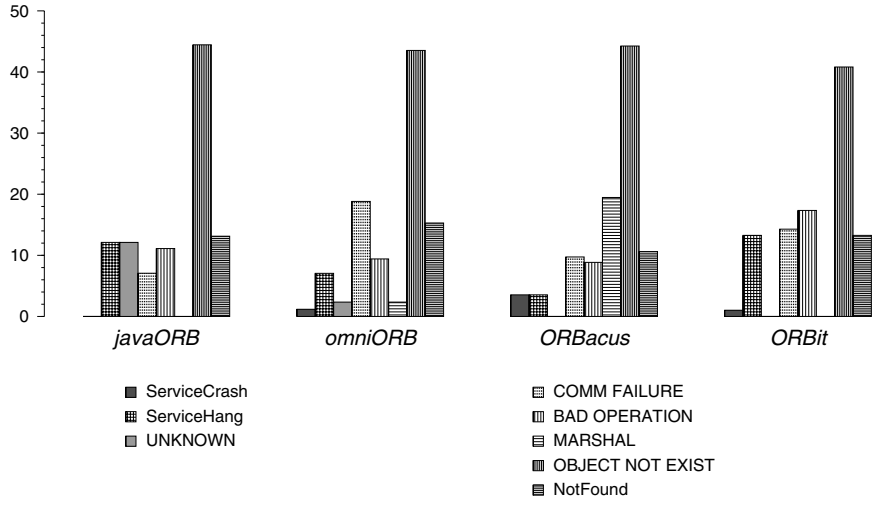


Fig. 1. Experimental outcomes for double-zero fault model

As stated earlier, our service hang failure mode does not imply that other clients of the naming service are blocked; we only consider the time taken to reply to the corrupted invocation. However, given its relative frequency, it is one of the more serious dependability problems we have identified. The upcoming CORBA 2.4 specification allows clients to specify timeouts on their requests, which would be helpful for detecting this type of situation without resorting to application-level watchdog mechanisms. Some of the implementations tested already support these interfaces or provide similar mechanisms.

Examining the details of the breakdown of CORBA exceptions, we observe that the Java implementation raises very few `COMM_FAILURE` exceptions, but a larger proportion of `UNKNOWN` exceptions (this exception is raised by an ORB when it detects an error in the server execution whose cause it cannot determine – for example, in Java, an attempt to dereference a null pointer). `UNKNOWN` is a less useful exception to signal to the application layer, since it conveys no information on the cause of the exception, so from this point of view the Java ORB can be considered less robust. The *ORBacus* service raises a greater proportion of `MARSHAL` exceptions, which indicates that its marshalling code does more error checking than other implementations (a positive point from a robustness point of view); *ORBit* does not raise `MARSHAL` exceptions.

The proportion of `OBJECT_NOT_EXIST` exceptions, which the ORB uses to signal that the object reference against which the method was invoked does not exist, is very similar between implementations. This is to be expected, since an ORB is required to check the validity of this value before dispatching the method invocation. A similar remark can be made for the `BAD_OPERATION` exception.

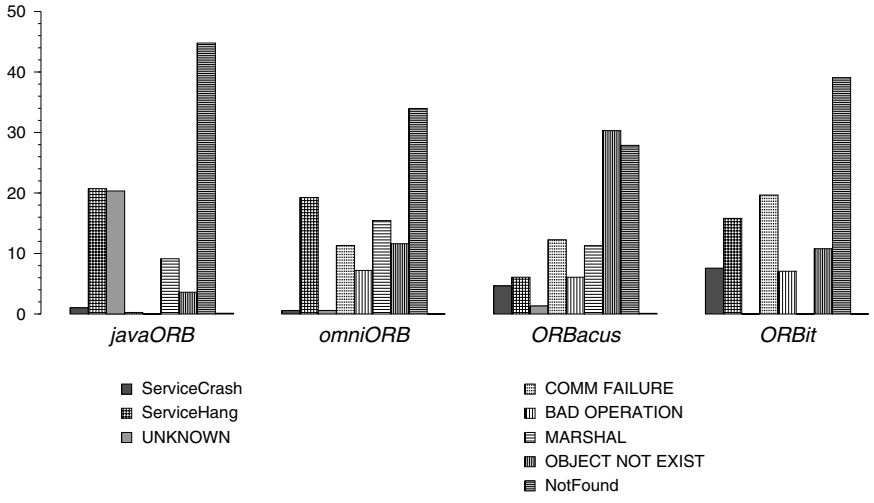


Fig. 2. Experimental outcomes for bitflip fault model

Influence of the error position. Figures 1 and 2 aggregate the results of faults injected at each possible position in the message. It is also interesting to examine the failure modes as a function of the position in the message where the fault was injected. For example, when the fault affects the part of the message which identifies the invoked operation, primarily `BAD_OPERATION` exceptions are signalled, as would be expected. Similarly, faults injected in the first few bytes of the IIOP request (which contain a special signature which identifies the message type) result mainly in `COMM_FAILURE` exceptions.

When the fault affects the header bits encoding the message's length, we mostly observe service hangs. Given that there are 32 bits to encode the message length, and that our messages are relatively short (around 900 bits), a bitflip in this zone is likely to increase the announced message length, so the service waits to read more data than will actually arrive.

Differences between fault models. Figure 2 shows the experimental outcomes for each target implementation for the bitflip fault model. The results differ slightly from those for the double-zero fault model. The first difference between the results from the two fault models is the appearance of a `InvalidName` exception which is not provoked by the double-zero fault model. This exception is raised by the naming service either when the name it is asked to resolve is empty, or –more likely in our case– when the name contains an invalid character.

A second observation is that the bitflip fault model results in a greater proportion of `MARSHAL` and `NotFound` exceptions. In the latter case, the difference is likely to be due to the service masking certain errors. Indeed, certain bits

in an IIOP message are unused. For example, the byte order of a message is represented by a zero or a one marshalled into an octet; seven of these bits are not significant, and so their corruption may not be detected by the ORB. In contrast, a double-zero error is unlikely to escape the notice of the ORB.

Certain other phenomena, such as the small proportion of `COMM_FAILURE` and `BAD_OPERATION` exceptions raised by *JavaORB* for the bitflip fault model, are more difficult to explain.

Internal error checking mechanisms. The *ORBacus* service was compiled in its default configuration, without disactivating internal "can't happen" assertions. When these assertions fail, the program voluntarily exits using the `abort` procedure. This leads to *ORBacus* showing a relatively high proportion of service failures, some of which could be avoided by using a different configuration. The *omniORB* implementation can be configured at runtime to abort when it detects an internal error, but we did not enable this feature.

System call trace analysis. Our testbed allows us to obtain system call traces and execution stack backtraces of the target process. These show that different middleware implementations activate the operating system in different ways. For instance, the *ORBacus* implementation makes a large number of `lwp_mutex` and `lwp_sema` calls, which enable the synchronization of threads, whereas the *omniORB* implementation uses a much narrower range of system calls, primarily for reading and writing to the network and to its log file.

The system call traces also illustrate differences in the level of internal error checking between ORB implementations. For example, when faults are injected into certain bit positions, the *ORBit* implementation causes a segmentation violation while decoding the corrupted message, and is forcibly aborted by the operating system. In contrast, the *ORBacus* implementation sometimes detects the corruption internally, and is able to print a warning message indicating the position in the program where the error was detected, before voluntarily aborting. This lack of internal error checking is a reasonable implementation decision for *ORBit*, since its primary design goals are high performance and a small footprint.

5 Related Work

There has been little research on middleware dependability characterization. Indeed, we know of no quantitative data on the failure rates of middleware-based systems, whereas this topic has been extensively studied for mainframe-type applications.

The most interesting work in middleware failure mode analysis through fault injection is [15], which reports on robustness testing of a number of CORBA implementations with respect to corrupted invocations of a portion of the client-side interface exposed by an ORB. For example, the `object_to_string` operation, which converts an object reference into a textual representation, is invoked with

an invalid object reference, to see whether the ORB crashes or hangs or signals an exception. This work has only targeted client-side operations; activity that involves interaction between a client and a server is not covered.

The Ballista methodology has also been applied to embedded automotive software built on CAN (Control Area Network), injecting faults such as message loss and reception delays [16].

There has been some work [17] comparing the robustness of CORBA and DCOM applications with respect to high-level failures. The faults considered are hangs and crashes of threads, processes and machines. Similarly to our results, the authors found a significant proportion of application hangs, which led them to recommend the use of application-level watchdog mechanisms. [18] used fault injection to evaluate the ability of fault tolerance middleware (Microsoft Cluster Server and NT-SwiFT from Bell Labs, which monitor processes and restart them upon failure) to improve the reliability of web and SQL services.

We are not aware of other characterization work on CORBA using fault injection. Other validation efforts have used a functional testing approach (such as the *CORVAL* project, which aims to test the functional correctness and the interoperability of ORB implementations) or have concentrated on performance evaluation (e.g., [19]).

6 First Lessons Learned

The experimental results presented in Section 4 show a relatively large variability of behaviour of the target candidates in the presence of faults. This demonstrates that, although the service's interface is standardized, a particular candidate's behaviour depends on the design and implementation decisions made by the vendor. In this section, we adopt the viewpoint of a system integrator who must select a candidate implementation for a safety critical system.

As such, we rank first candidates that deliver significant error reporting information, i.e., those which exhibit fewer service hangs and `UNKNOWN` exceptions. These are the most problematic failure modes when deciding on fault tolerance strategies and error recovery mechanisms that can meet the system's dependability requirements.

By grouping all the exceptions except for `UNKNOWN` together, we obtain the percentages for the bitflip fault model shown in Table 1.

Table 1. Ranking of service implementations

Implementation	Exception	`UNKNOWN`	Service Hang	Service Crash
ORBacus	88.0	1.3	6.1	4.6
omniORB	79.5	0.6	19.3	0.6
ORBit	76.6	0.0	15.8	7.6
Java SDK	58.0	20.3	20.7	1.0

From this viewpoint, the *ORBacus* and *omniORB* implementations exhibit the safest behaviour: more significant exceptions are reported, i.e., fewer `UNKNOWN` exceptions, and there is a smaller proportion of service hangs. *ORBacus* has a relatively high rate of service failure, which (as discussed in Section 4.2) is partly due to the configuration we chose. This type of reaction to abnormal situations is not necessarily a negative point from a dependability viewpoint. Many fault tolerance strategies, particularly in a distributed computing context, make a *fail silence* assumption, which requires components to produce either correct results, or none. Silent failures can successfully be handled by replication, either by using identical copies located on different sites, to deal with physical or environmental faults [20], or by using diversified copies to protect against software faults [21].

We also observed in the experiments that the behaviour depends on the fault model. The results obtained with double zeroing and bitflips lead to a different statistical distribution of the failure modes. However, the resulting numbers do not disturb the ranking given in Table 1. Many issues can influence the observed results. Nevertheless both types of experiments leading to the same conclusions reinforce the confidence one can have in the ranking.

Clearly, many other aspects of middleware dependability must be taken into account in the final selection of a candidate. In particular, the effects of other classes of faults need to be investigated. From this viewpoint, the work done by the Ballista project [15], which uses a different fault model and targets a different part of the middleware, is complementary to ours.

7 Conclusions and Future Work

This paper has presented an experimental robustness evaluation method for CORBA-based services, and results of experiments targeting four implementations of the CORBA Name Service. These experiments evaluate the effect of corrupted method invocations at the middleware level.

The implementations we have tested show a non-negligible number of robustness weaknesses, but we have not observed cases of error propagation from the middleware to the application level. Our results suggest that the robustness of CORBA-based systems would be enhanced by the addition of an application-level checksum to GIOP. The achieved failure mode characterization aids in the selection of a candidate middleware implementation for critical systems, and helps system integrators decide on the error detection and recovery mechanisms, fault tolerance strategies and architectural solutions that are needed to meet dependability requirements.

Our technique is non-intrusive, and (thanks to the transparency provided by CORBA) easy to port, both to new implementations of the service, and to alternative operating environments (operating system, hardware platform). The approach could also be applied to assess the robustness of other CORBA services, by modifying the workload and the fault injector.

We plan to continue this work in several directions. Our fault injection tools will be improved to obtain measures of error detection latencies. We plan to

investigate the influence of the operating system on service behaviour in the presence of faults, by carrying out experiments on Windows2000 and Linux. We would also like to examine the influence of system and network load on the service's failure modes, since previous research on fault injection has shown that error propagation is more likely under heavy load. More generally, we plan to conduct further fault injection experiments using other fault models, including RAM bitflips to simulate the effect of transient hardware faults, and code mutation techniques to study the effect of software faults.

Acknowledgements. This work is partially supported by the European Community (Project IST-1999-11585: DSoS – Dependable Systems of Systems). The authors would like to thank Jean Arlat for helpful comments on their experiments.

References

1. J. C. Laprie, "Dependable computing: Concepts, limits, challenges," in *25th IEEE International Symposium on Fault-Tolerant Computing - Special Issue*, pp. 42–54, IEEE Computer Society Press, 1995.
2. M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure data analysis of a LAN of Windows NT based computers," in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99)*, (Washington - Brussels - Tokyo), pp. 178–189, IEEE, Oct. 1999.
3. J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 42, pp. 913–923, Aug. 1993.
4. J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture," in *Proc. 5th IFIP Working Conference on Dependable Computing for Critical Applications: DCCA-6*, pp. 267–287, IEEE Computer Society Press, 1998.
5. M. Rimén, J. Ohlsson, and J. Torin, "On microprocessor error behavior modeling," in *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing*, (Los Alamitos, CA, USA), pp. 76–85, IEEE Computer Society Press, June 1994.
6. E. Fuchs, "Validating the fail-silence of the MARS architecture," in *Proc. 6th IFIP Int. Working Conference on Dependable Computing for Critical Applications: DCCA-6*, pp. 225–247, IEEE Computer Society Press, 1998.
7. H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN2000)*, pp. 417–426, IEEE Computer Society Press, 2000.
8. P. J. Koopman and J. DeVale, "Comparing the robustness of POSIX operating systems," in *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS-29)*, (Los Alamitos, CA, USA), pp. 30–37, IEEE Computer Society Press, 1999.

9. J.-C. Fabre, M. Rodríguez, J. Arlat, F. Salles, and J.-M. Sizun, "Building dependable COTS microkernel-based systems using MAFALDA," in *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing (PRDC-2000)*, IEEE Computer Society Press, 2000.
10. S. Dawson, F. Jahanian, and T. Mitton, "Experiments on six commercial TCP implementations using a software fault injection tool," *Software Practice and Experience*, vol. 27, pp. 1385–1410, Dec. 1997.
11. Object Management Group, Inc, "CORBAServices: Common Object Service Specification: Naming Service Specification," Documentation available at www.omg.org, Object Management Group, Feb. 2001.
12. J. F. Ziegler and G. R. Srinivasan, "Preface: Terrestrial cosmic rays and soft errors," *IBM Journal of Research and Development*, vol. 40, pp. 2–2, Jan. 1996.
13. J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," in *Proceedings of the 2000 ACM SIGCOMM Conference*, pp. 309–319, 2000.
14. S. Dawson and F. Jahanian, "Probing and fault injection of dependable distributed protocols," *The Computer Journal*, vol. 38, no. 4, pp. 286–300, 1995.
15. J. Pan, P. Koopman, D. Siewiorek, Y. Huang, R. Gruber, and M. L. Jiang, "Robustness testing and hardening of CORBA ORB implementations," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN2001)*, IEEE, June 2001.
16. P. Koopman, E. Tran, and G. Hendrey, "Toward middleware fault injection for automotive networks," in *Proc. 28th Int. Symposium on Fault-Tolerant Computing (FTCS-28)*, pp. 127–135, IEEE Computer Society Press, June 1998.
17. P. E. Chung, W. Lee, J. Shih, S. Yajnik, and Y. Huang, "Fault-injection experiments for distributed objects," in *Proceedings of the International Symposium on Distributed Objects and Applications* (IEEE, ed.), 1999.
18. T. Tsai and N. Singh, "Reliability testing of applications on Windows NT," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN2000)*, pp. 427–436, IEEE Computer Society Press, 2000.
19. S. Nimmagadda, C. Liyanaarachchi, A. Gopinath, D. Niehaus, and A. Kaushal, "Performance patterns: Automated scenario based ORB performance evaluation," in *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pp. 15–28, The USENIX Association, 1999.
20. D. Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Berlin, Germany: Springer-Verlag, 1991.
21. K. S. Tso and A. Avizienis, "Community error recovery in N-version software: A design study with experimentation," in *Proc. 17th Int. Symposium on Fault-Tolerant Computing (FTCS-17)*, pp. 127–135, IEEE Computer Society Press, 1987.

ROOM-BRIDGE : Vertically Configurable Network Architecture and Real-Time Middleware for Interoperability between Ubiquitous Consumer Devices in the Home

Soon Ju Kang, Jun Ho Park, and Sung Ho Park

School of Electrical Engineering and Computer Science,
Kyungpook National University, Korea
sjkang@ee.knu.ac.kr,{zec,slblue}@palgong.ac.kr

Abstract. This research focuses on the development of a generic home network architecture and related middleware for supporting interoperability between ubiquitous consumer devices and computing devices in the home. In order to build a practical home network, design criteria are discussed, including real-time constraints and interoperability between heterogeneous protocols. As a result, a vertically configurable home network architecture and real-time middleware, called ROOM-BRIDGE, are presented that meet the design criteria. ROOM-BRIDGE is specially designed and implemented using an IEEE1394 backbone network to provide the proposed network architecture with a guarantee of seamless and reliable communication between heterogeneous sub-networks and ubiquitous devices in the home. The performance of the proposed network architecture and ROOM-BRIDGE was verified by prototype implementation and testing using a home network test bed.

1 Introduction

The home is a typical ubiquitous computing environment as it contains many types of consumer devices and computing devices. Intensive research efforts have been made to build a practical home network [1,2], however, at this point there is an overabundance of protocols and middleware systems in the home network industry, with no clear winner to cover all services in the home. This is basically because a home network requires multiple services, whereas most protocols have only been designed to support a specific service. The currently existing protocols and middleware systems can be categorized into three major services [1,2]. The first is home automation services, including the remote control of room temperature, lighting, or windows, plus the implementation of electric or gas telemetry. Since this type of network must be very reliable and safe, it is normally implemented using solutions in a control area network [4,8], originating from the building and factory automation industries. The second is home theatre services through an audio/video(A/V) network. This type of network connects various kinds of audio and video devices, including TVs, VCRs, camcorders,

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 232-251, 2001.

and so on. IEEE1394 is one of the de facto standard protocols in this A/V field and is already embedded in many audio and video devices [5,6]. The third is Internet and wireless access services [7], which enable remote reality for home devices through an Internet or wireless network. The second and third types of service can also be re-grouped into a data network category.

Because the development of such services has been pursued independently, there is currently an overabundance of protocols and middleware systems for supporting each specific type of service in the home network industry. LonWork [4,8], BACnet, and DeviceNet are examples of control area networks and support home automation services. HAVi(Home Audio/Video Interoperability) [6] and VESA [1] are the dominant middleware systems for A/V networks through IEEE1394 for supporting home theater services. CORBA [12] and JINI [9] are middleware systems for the Internet, while Bluetooth [1,2] is the main standard for wireless solutions in the home. In addition to the above protocols, there are also many more, e.g. UPnP by MicroSoft, CEBUS and CAL, in fact too many to mention in this paper.

Accordingly, due to the plethora of protocols and idiosyncrasies of each one, this creates a serious dilemma when attempting to build a practical home network. One homogeneous protocol is certainly not enough, whereas wiring a house with a sophisticated web of several independent networks to accommodate the idiosyncrasies of each protocol is equally impractical. To make matters worse, each protocol and middleware has little consideration for interoperability with other systems.

Therefore, to solve these problems and build a practical home network using currently existing protocols and the resources produced by each protocol supporting an organization, the current study adopts the concept of a vertically integrated network architecture, as first introduced by E. D. Jensen [3]. As such, the focus is on the development of a new home network architecture and related middleware that can vertically abstract low-level heterogeneous protocols without losing any of their idiosyncrasies. The basic design criteria for the proposed architecture are as follows; first, the structure of a traditional house is considered as a design criterion for the vertical integration of a home network. For example, a house can be divided into several subsections, such as rooms, the kitchen, and so on. Similarly, a home network can also be configured into several subnets and each subnet copes with the networking of ubiquitous consumer devices inside a particular subsection, such as a room or the kitchen. As a result, each sub-network consists of several local networks with homogenous protocols according to the services, e.g. a LonTalk network for home automation, IEEE1394 network for multimedia services, and so on. Second, a broadband backbone network is needed to connect all the subnets. Third, a subnet gateway server, called a ROOM-BRIDGE server, is specially designed to connect each subnet into the backbone network and accommodate un-networked ubiquitous computing devices, such as irDA devices. Fourth, a specific middleware, called ROOM-BRIDGE, and device driver architecture for embedding into each ROOM-BRIDGE server are designed and implemented to guarantee interoperability, a real-time response, and reliability in the proposed home network.

Chapter 2 introduces the basic concept and related research, along with the design criteria for the proposed architecture. Chapter 3 explains an overview of the proposed

home network architecture as a target hardware system architecture and introduces the ROOM-BRIDGE server. Chapter 4 presents a brief explanation of the core components in the proposed middleware, called ROOM-BRIDGE. Chapter 5 provides a detailed description of the resource repository model, while the priority based event channel model and real-time device driver for an IEEE1394 adaptor are presented in chapters 6 and 7, respectively. Chapter 8 discusses the experimental results of the prototype home network based upon the ROOM-BRIDGE architecture. Chapter 6 presents some final conclusions along with areas for further research.

2 Related Work and Background

Although many studies continue to be conducted on developing a practical home network [1,2] [4,5,6,7,8] a de facto standard protocol and middleware that can cover all services in the home has not yet appeared. As already mentioned, the main reason is due to the idiosyncrasies related to each existing protocol. For a brief survey of these idiosyncrasies, two protocols, LonTalk [4] for home automation services and IEEE1394 [5] for home theatre services, were selected as they have already been accepted as the dominant standard in their respective areas.

2.1 LonTalk

The LonTalk protocol is a widely adopted protocol used for communication between intelligent embedded sensors, actuators, and controllers. The LonTalk protocol is based on a control area network, therefore, it is a mechanism whereby intelligent devices can exchange control and status information. LonTalk was originally developed by the Echelon Corp. and implemented as an integrated microcontroller and network communications chip (the Neuron Chip). The LonTalk protocol is able to describe using the 7-layer ISO model, as shown in Table 1, and has been standardized as EIA-709.1. As such, it is freely available for implementation in any microprocessor making it fully interoperable with Neuron Chip implementations [4].

2.2 IEEE1394 and HAVi

IEEE 1394 is an international standard, low-cost digital interface that can integrate entertainment, communication, and computing electronics into consumer multimedia. Originally designed by Apple Computer as a desktop LAN and then developed by the IEEE 1394 working group, IEEE 1394 has the following features:

- Hot pluggable - users can add or remove 1394 devices using the active bus. Scaleable architecture - can mix 100, 200, and 400 Mbps devices on a bus.
- Flexible topology – can support daisy chaining and branching for true peer-to-peer communication.

- Serial Bus Management provides overall configuration control of the serial bus in the form of optimizing arbitration timing, guarantee of adequate electrical power for all devices on the bus, the assignment for which the IEEE 1394 device is the cycle master, assignment of isochronous channel ID, and notification of errors. The bus management is built on IEEE 1212 standard register architecture

There are two types of IEEE 1394 data transfer: asynchronous and isochronous. Asynchronous transport is a traditional computer memory-mapped, load and store interface. Data requests are sent to a specific address and an acknowledgment is returned. In addition to an architecture that scales with silicon technology, IEEE 1394 features a unique isochronous data channel interface. Isochronous data channels provide guaranteed data transport at a pre-determined rate. This is especially important for time-critical multimedia data where just-in-time delivery eliminates the need for costly buffering. As shown in Table 1, the IEEE 1394 protocol layer is only defined until the transport layer, plus all network management features from the physical layer to the transport layer are integrated into a hardware-based module. Therefore, the application programmer is not required to understand any of the complex network management features. All these features of IEEE 1394 are key reasons why it has become the A/V Digital interface of choice [2,7].

HAVi [6] is the dominant standard middleware architecture for audio / video home networks, and is intended for consumer electronic devices and computer devices supporting the IEEE Std 1394-1995 and IEC 61883 [5] interface standard. HAVi provides a set of services that facilitate interoperability and the development of distributed applications in home networks.

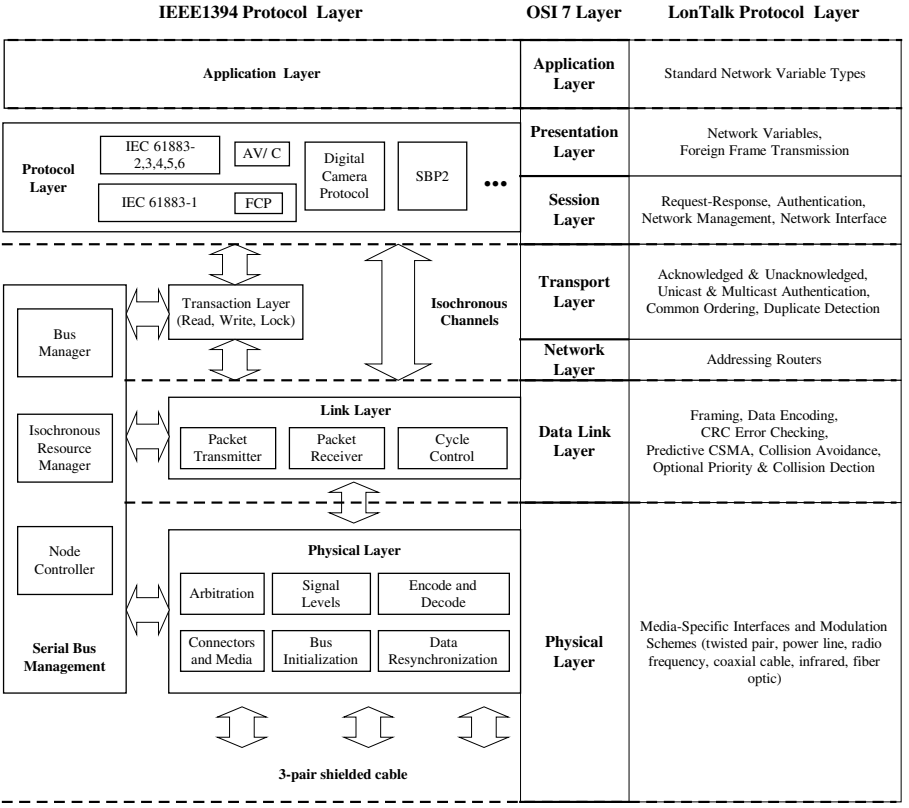
2.3 Comparison of Idiosyncrasies between LonTalk and IEEE1394

Table 1. shows a brief comparison of the characteristics of the two protocols relative to the ISO 7 layer [4,5] As shown in Table 1, the characteristics of each protocol are quite distinct from each other, e.g. LonTalk supports various physical communication media, such as power lines, coaxial cables, twisted pair cables, yet IEEE1394 only supports 3-pair shielded cables. Based on this property, the LonTalk supporting association can also produce a network interface module, called a Neuron Chip, for embedding into home appliances or small-scale devices, such as sensors or lights. Since a Neuron Chip has a certain level of computing power, various small-scale devices with either no or less computing power can be easily connected. In addition, LonTalk supports event-based priority scheduling and rigorous timing in order to perform real-time services.

In contrast, IEEE1394 has a casual data communication channel, referred to as the asynchronous mode, plus an additional powerful broadband communication channel, referred to as the isochronous mode, for guaranteeing multimedia communication with QoS (Quality of Service). Although the IEEE1394 supporting association also produces a communication chip [5] for embedding into devices, this chip only includes a module for communication support with no additional module to support the computing power of the embedded devices. Therefore, when compared to LonTalk, it is much

more difficult to embed IEEE1394 into small-scale devices, such as lights or sensors. In addition, the asynchronous mode of IEEE1394 has no features to guarantee real-time constraints, and the isochronous mode is too different to achieve interoperability with LonTalk or traditional data communication protocols, such as TCP/IP. In the case of middleware, LonTalk has a homogenous protocol stack that covers the 7 layers, as shown in Table 1, and no middleware concept for supporting interoperability with other protocols. In the case of IEEE1394, although the IEEE1394 supporting association already suggests a well-known middleware, called HAVi, it has less consideration for none-IEEE 1394 devices, and no specific technical consideration for devices belonging to the home automation category. Accordingly, it has no definition of protocol level interoperability, such as IEEE1394 via LonTalk, and no features to guarantee real-time constraints through its asynchronous channel.

Table 1. Comparison between LonTalk and IEEE1394



Due to these facts, LonTalk is currently widely utilized in factory automation, building automation, and even home automation, plus a variety of products supporting LonTalk, including air-conditioners, gas and electricity meters, and lighting solutions, have been produced by several companies. Conversely, IEEE 1394 has become the de

facto standard for multimedia consumer devices, such as camcorders, digital VCRs, and digital cameras, thus the majority of commercial products in this area support an IEEE1394 port. However, at this point there is no possibility of interoperability without device-level protocol changes. Accordingly, since neither of these protocols can be ignored when building a practical home network, a new architecture that can simultaneously incorporate these protocols on a home network platform is necessary.

2.4 Design Requirements and Guidelines for Proposed Architecture

Based on the above study of the idiosyncratic properties of the two heterogeneous protocols, the following design requirements are defined for developing a practical home network architecture and related middleware:

- Accommodating various ubiquitous consumer and computing devices
  It is reasonable to expect that a home network should accommodate various kinds of consumer devices, whether or not they have computing power, and as many as possible in order to increase the practicality and maximize the benefit of home networking. Therefore, to achieve this goal, the proposed home network and related middleware should have an extendable and re-configurable architecture to accommodate any kind of consumer device.

- Vertically configurable network hardware architecture
  As mentioned in the previous section, a home network is totally different from a general data network in that it has many and various kinds of devices and each device has different requirements for joining a service transaction in the network, i.e. some devices require a high communication bandwidth with a QoS guarantee, while other devices require reliable and rigorous communication with a real-time guarantee. In spite of these low-level individual properties [21], high-level application service agents can still perform without considering the low-level protocol properties. In addition, agents or a home member also can control several consumer devices without considering their location and properties, such as “turn off all consumer devices in room 5” or “ send a turn-on signal to the TV in room 2 using the remote control in room 3”. To support these features in the proposed architecture, a vertically configurable architecture is considered, which means that the application programmer can perform unit-base device programming or group-base device programming, plus logical unit programming, for example, a room or a group of devices related with a service. For implementation, the proposed home network consists of a backbone network and several subnets. IEEE1394 is used as the backbone protocol and each subnet accommodates several homogenous local networks and un-networked ubiquitous devices. In order to connect these subnets to the backbone network, a new subnet gateway is proposed and implemented.

- Messaging system to meet real-time and QoS constraints
  As shown in Table 1, the LonTalk protocol supports priority-based messaging and rigorous timing. Plus, according to the application agent, a real-time response

based on the interaction between non-real-time devices and real-time devices is also needed. To enhance this feature in the proposed middleware, an asynchronous messaging mechanism is implemented that is a modified version of the event channel concept from Real-Time CORBA [12]. Since multimedia communications for home theater services inevitably need a mechanism for guaranteeing QoS [13], the features in IEEE1394 and HAVi are used in the proposed middleware.

- Decentralized resource repository
  The current status and computing resource information for all connected devices should be monitored and managed in a resource repository for the application agents. However, if all the agents look up the necessary information from only one centralized resource repository, as in the JINI model [9], then the traffic for looking up the resources is centralized and the network frequently becomes unreliable when too many active agents are running on the network [15]. Therefore, to prevent this situation, a replication model of a resource repository is suggested, where each subnet has it's own resource repository yet all repositories in the network always have the same content.
- Minimizing the overhead despite multiple protocols and ubiquitous device support
  To develop a practical middleware, the reuse of features in existing low-level protocols is more important than implementing from a scratch-base. Although the software architecture of the proposed middleware is not dependent on a specific low-level protocol, it was decided to use the IEEE1394 protocol as the base protocol for the backbone network, thereby enabling the use of all the features of IEEE1394, such as an addressing scheme, dynamic reconfiguration, and multimedia communication through isochronous channels. Consequently, only one device driver and a protocol adaptor module are added to the middleware layer for supporting the new protocol in the proposed middleware.
- Room-based self-management architecture to reduce complexity and increase reliability
  In a traditional house, a room is one of the important management units from several aspects, such as management complexity, security, and reliability. Therefore, a room-based self-management architecture is definitely required to accommodate traditional management customs in a house.

3 Vertically Configurable Network Architecture: Target Hardware System Model

Base on the design guidelines discussed in section 2.4, a vertically configurable network architecture (Figure 1 shows the physical topology of the proposed architecture) is presented. As shown in Figure 1, the main criterion for dividing a home network into subnets is not a homogenous subnet with a protocol, but rather the physical structure of a traditional house. For example, a house consists of several subdivisions, such

as rooms, the kitchen, and so on. This physical structure then becomes the main criterion for dividing the entire home network into several logical subnets. Each subnet only has one subnet gateway, called a ROOM-BRIDGE server and supports several homogenous local networks, such as a LonTalk network, IEEE1394 network, and wireless network.

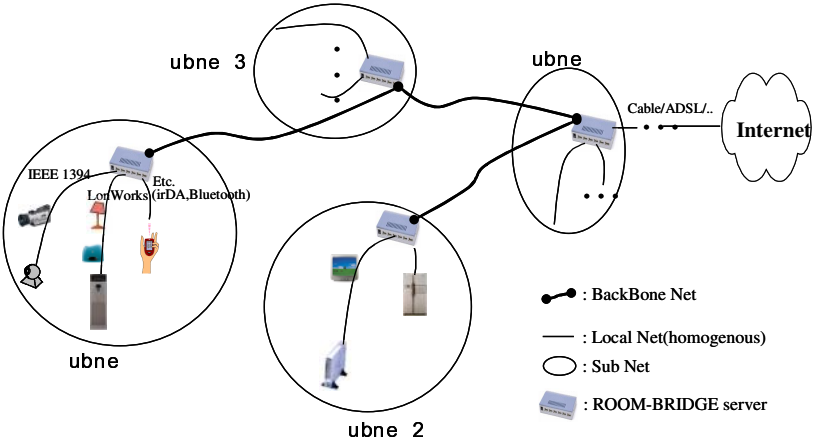


Fig. 1. Physical topology of proposed home network architecture

All subnets are connected to an IEEE1394-based backbone network via a ROOM-BRIDGE server. As such, the topology of the proposed network is dependent on the characteristics of the backbone network, i.e. flexible daisy chaining with IEEE1394. In particular, all the local homogenous networks attached to a subnet gateway need not be aware of other protocols or backbone middlewares and only need to focus on their own subnet gateway as the network management server for their local network. Accordingly, it is the ROOM-BRIDGE server that provides the interoperability between local networks and the backbone network. To fulfill this task, a middleware, called ROOM-BRIDGE, was designed and implemented for embedding in all ROOM-BRIDGE servers(the proposed middleware is explained in next section). To perform the functions of a ROOM-BRIDGE effectively, a specially designed embedded computer system was used in the current study as a ROOM-BRIDGE dedicated hardware platform, however, any computing device, such as a game machine (e.g. PlayStation 2) or PC (Personal Computer) that has enough computing power to execute the functions of the proposed middleware could be used as an alternative ROOM-BRIDGE server. Since the detailed architecture of the ROOM-BRIDGE dedicated hardware platform is outwith the scope of the current paper, it is not mentioned further.

4 Overview of ROOM-BRIDGE

The proposed middleware, called ROOM-BRIDGE, was designed and implemented to support the above design guidelines and proposed network configuration architecture,

as shown in Figure 1. The following introduces the middleware components and operational principle for realizing the proposed home network. The components in the middleware layer, as shown in Figure 2, are the core components of ROOM-BRIDGE that need to be installed in all the subnet gateway servers, whether they are a ROOM-BRIDGE-dedicated hardware platform or full-powered computing devices, e.g. a game machine or PC. As such, the core components of the proposed middleware directly interact with the local networks through pairing a specific network manager with the appropriate device driver. For example, if a subnet wants to support a LonTalk network as a local network, the ROOM-BRIDGE server will pair up the LonTalk network manager with the device driver of a LonTalk adaptor board. In addition, ROOM-BRIDGE can also support several kinds of APIs(Application Program Interfaces) for programming application agents using the proposed middleware.

The middleware layer components can be divided into four categories: Device Status and Resource Registry, Device Proxies, Network Managers, and Priority-based Event Channel. Since the proposed middleware currently does not consider an isochronous communication channel and just uses the basic features in IEEE1394, isochronous applications, such as multimedia communications, only use the proposed middleware layer for sending control commands, then the IEEE 1394 network manager facilitates multimedia data communications by-passing the proposed middleware directly to an isochronous channel, as shown in Figure 2.

A brief summary of each group of components follows and more detailed descriptions are included in later sections as appropriate.

- Network Managers
  Each network manager (IEEE1394 network manager, LonTalk network manager, other network managers in Figure 2.) uses the subnet gateway as a network management server for each homogenous local network and also performs a filtered conversion of local network data to backbone network data. In particular, since the IEEE1394 network manager includes additional functions allowing other software elements to perform asynchronous and isochronous communication over 1394, command and network management data are processed by the proposed middleware components, whereas isochronous data, such as a multimedia data stream with QoS controls, are by-passed by the IEEE 1394 network manager to an isochronous channel.

- Device Manager
  Whenever a new device is connected into the home network, the Device Manager acknowledges it, initializes a device proxy for the physical device, and then registers it into the device pool.

- Device Proxy
  The Device Proxies in the device pool can be categorized into two groups: One group is real device proxies mapped one-to-one with physical devices, whereas the other is virtual device proxies mapped one-to-many or with non-physical devices, e.g. application program tasks or user program tasks, such as graphical user interface tasks, intelligent agents for instruments or the autonomous control of physical devices.

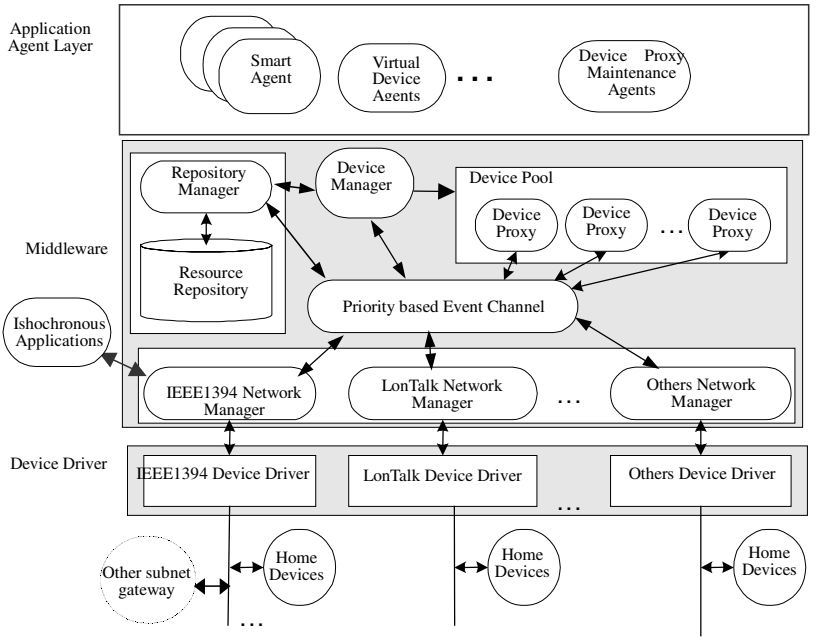


Fig. 2. Overview of ROOM-BRIDGE as middleware embedded in subnet gateways

Each device proxy mapped with a physical device in a one-to-one manner must have remotely invocable functions for monitoring and controlling the physical device without considering low level communication protocols(this protocol conversion is part of the network manager). A device proxy also has abstracted I/O variables for each active device in the home network. In addition, physical device proxies can act as a control and monitoring server to distributed clients by supporting callback functions and remote methods.

- Resource Repository and Repository Manager
  The repository manager plays the role of maintaining the status information on all the devices in the home network and computing resources for looking up remote methods. Since at least one repository must exist in each ROOM-BRIDGE server as a subnet gateway, the home network has the same number of repository managers and subnets. The device status information and related operational methods for monitoring and controlling each device are registered in the repository data base and managed by the repository manager in a subnet gateway. The resource repository database is a temporal and memory resident database, plus it always reflects real-time status information on all devices in the home network. To achieve this, the tuple - space model [10] and space programming concept in Java [11]are used. Further details are included in the following section.

- Priority-based Event Channel
  All events in the proposed home network are processed in an asynchronous manner. Therefore, an event channel with a priority queue is the basic asynchorouns message sending mechanism in the proposed concept, plus it also includes an en-

hanced mechanism for the real-time response of higher priority events. All events in a home network are categorized into three groups; the first group comprises events for maintaining the integrity of the resource repositories, the second includes events for reporting a status change from physical device proxies to resource repository managers, and the third involves events with acknowledgements for controlling remote physical devices. Based on these categories, each event has its appropriate priority value, whether it is assigned statically or dynamically, and is sent to its destination through this event channel. Section VI presents more details on the proposed event channel.

5 Resource Repository and Resource Manager

As shown in Figure 1, the proposed home network consists of logically separated subnets and each subnet a certain number of application agents and proxies for controlling or monitoring physical home devices. Since a home network is a highly sophisticated network, if a tightly coupled process model were used between the application agents and the physical device proxies or a centralized server for looking up resources in the entire network were used, such as the name server in CORBA [10], the reliable operation of the entire home network would become a critical issue as the network could easily go into a failure mode or even shutdown due to too many agents or the sudden failure of the server. Accordingly, to prevent these situations and create a more reliable and fault-tolerant network, a replicated space model for the repository is proposed based on the tuple-space model [10].

As shown in Figure 3, each subnet has its own repository space, which is shared by several agents and proxies, and all repositories in the home network are interconnected to each other through the event-messaging model outlined in the following section. Therefore, an agent can autonomously manage a specific home device, room-net, or several other agents using the repository space wherein the status of all devices can be transparently accessed regardless of the agent's location. Because of the transparency of this location, a newly attached home device or agent can easily obtain information on another agent by either reading or taking the information from the repository space. Similarly, a newly attached home device or agent can equally also announce its information to all other repositories in the home network. Therefore, agents coordinating several devices can provide a more reliable and highly intelligent service as they can easily access information on devices even though the devices may be completely unaware of each other. As a result, since all agents are loosely coupled to each other, this makes the system flexible, scalable, and reliable.

In particular, a replication model of a resource repository for maintaining device status information and resource registry is suggested, meaning that each subnet gateway has its own registry data base and all the device registries in the home network always have the same content. Therefore, whenever any changes to a physical device occur, each device proxy directly sends this information to the resource repository in the immediate subnet, thereafter the repository manager broadcasts the information to all other resource repository managers in the home network.

To synchronize the content of the replicated resource repository space in a home network, a proportion of IEEE1394's higher priority asynchronous stream channel must be allocated whenever the network is initialized. If an IEEE1394 bus is initialized(IEEE1394 buses are used as the backbone in the proposed architecture), the connection manager in the IEEE1394 network selects a channel management node, then the node allocates a channel number for broadcasting the resource repository information over the home network. After the initial broadcasting, an exchange between all device proxies then occurs, item by item, through the normal data transfer channel.

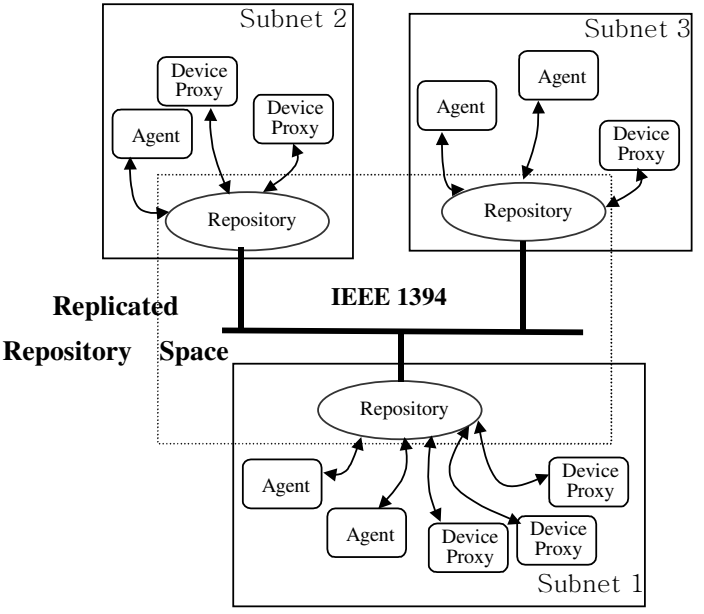


Fig. 3. Replicated Repository Space Mode

Despite the replication of the resource repository, the performance overhead caused by the task of synchronizing the resource repositories is not too high because the contents in the resource repository are highly abstracted and only include high-level information on each device. The following is an example of device status information in a resource repository:

```
("Light",Room2,5,"On",6,Time/Date)
("Light",Room2,7,"On",1,Time/Date)
("TV",Room1,5,"Off",6,Time/Date)
("AirCon",Room1,1,"On",115,Time/Date)
....
```

As shown in the above example, in the proposed resource repositories there is no local network specific information, such as IEEE 1394 bus management or LonTalk node information, only device specific information. Protocol-specific information is man-

aged in each local subnet through specific network managers, such as an IEEE1394 network manager or LonTalk network manager, as shown in Figure 2.

Each device proxy includes some tuple-based operations [11,17] for retrieving and archiving device registry information between other agents in other subnets or device proxies inside a subnet. For example, a light management application agent can control the light status as follows: ***write("Light",Room2,5,"Off")***. To produce a more extensive operation, each operation can be used as a template-based query as follows, ***read("Light",Room2,?X,"On")***. The ?X means a variable in the query, as such, a set of items can be matched into the query, thereby facilitating a group-based query operation, for example, "turn off all lights in room 2". Further details of this resource repository architecture can be found in reference paper [17] written by the current authors. Consequently, this replication resource repository model provides all physical and virtual device proxies with transparent access to all device information in the home network, regardless of the physical location of each device.

6 Priority-Based Event Channel under IEEE1394 Asynchronous Mode

This section presents the internal architecture of the proposed event channel designed to implement an asynchronous message event handling mechanism and enhance the real-time response of events communicated using the proposed middleware. Although based on the publish-and-subscribe messaging model of CORBA and Real-Time CORBA [12,16], the concept is totally redesigned and modified for the IEEE1394 aynchronous transfer mode. As shown in Figure 4, the event channel consists of several priority queues, a dispatcher, and preprocess module. When generated, an event is transferred through a sequence of the preprocess module, priority queues, and dispatcher module.

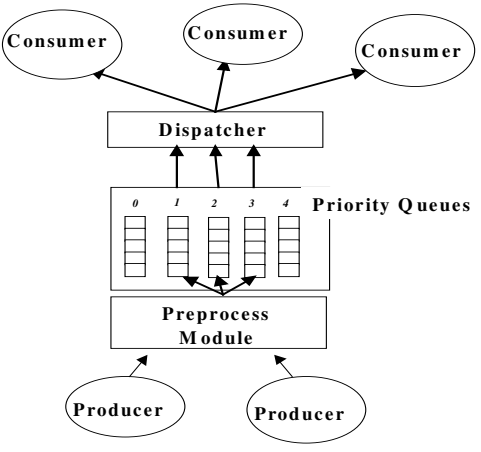


Fig. 4. Event Channel Architecture

6.1 Event Message Format, Producer, and Consumer

All software objects in the proposed home network, including the core components of the middleware and application agents, can be categorized into three object classes; event, producer, and consumer, as shown in Figure 4.

An event contains unit data for communicating messages between the software components in the home network, a producer is a software component that sends an event to others, and a consumer consumes the events that arrive. The following code example shows the class definitions of an event.

```
class Event{
        ObjectID PID;   // the reference of producer object  ID
        ObjectID CID;   // the reference of mapping consumer object ID
        EventType type; // event types such as repository, event with ACK or
                            witout ACK
        Priority value; // priority level of this event
        Int TimeSlice;  // optional, for ACK event
        Int MessageLength;
        Char* Message;
}
```

As shown in the above event definition, the producer decides all the slot values, such as the source and destination ID, event type, and priority, for an event with the help of the resource repository manager before sending the event to the event channel in the subnet. After sending, the successful delivery of the event is left to the event channel. Events can be classified into two types: events with acknowledgement(Figure 5) and events without acknowledgement(Figure 6). The former are used for controlling remote devices safely, whereas the latter are used to send newly updated device status information to all the resource repositories in the network.

6.2 Preprocess Module

The preprocess module has two functions. The first is run-time scheduling and the second is checking the timeout for events requiring ACK, as shown in Figure 5. For scheduling, the preprocess module determines the priority of an event using the event's scheduling information. The checking mechanism for events requiring ACK is also implemented in this module to assure the timing deadline of events. Figure 5 shows an event sequence diagram describing this function.

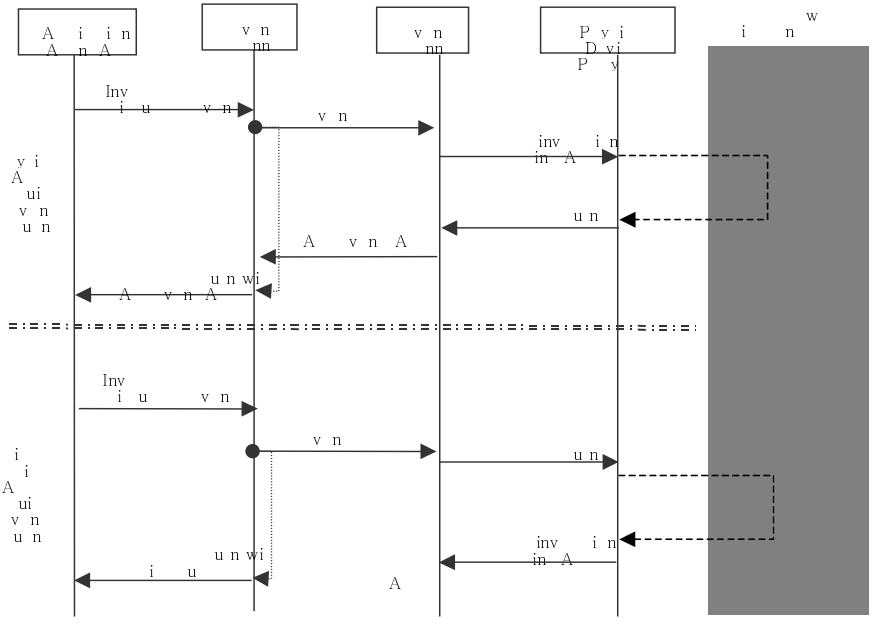


Fig. 5. Event with Acknowledgement

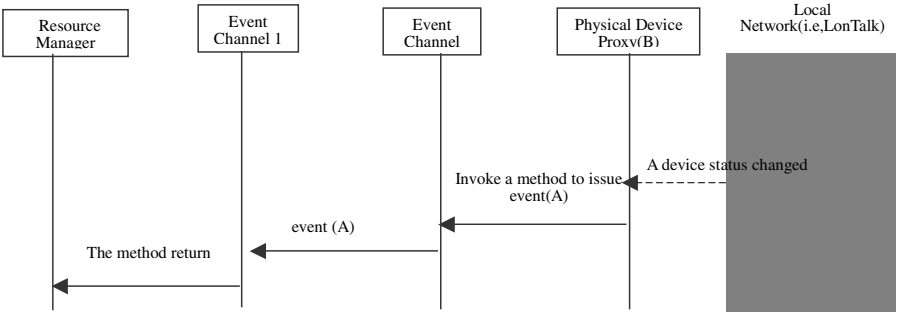


Fig. 6. Event without Acknowledgement

6.3 Priority Queues and Dispatcher

In Figure 7, each queue represents a preemption policy for each consumer priority level. The dispatcher module in Figure 7 uses threads to dispatch the events in the priority queues to the consumer. The proposed architecture allocates a thread or thread pool to each priority queue. Figure 7 shows the relation between the priority queues and the dispatcher.

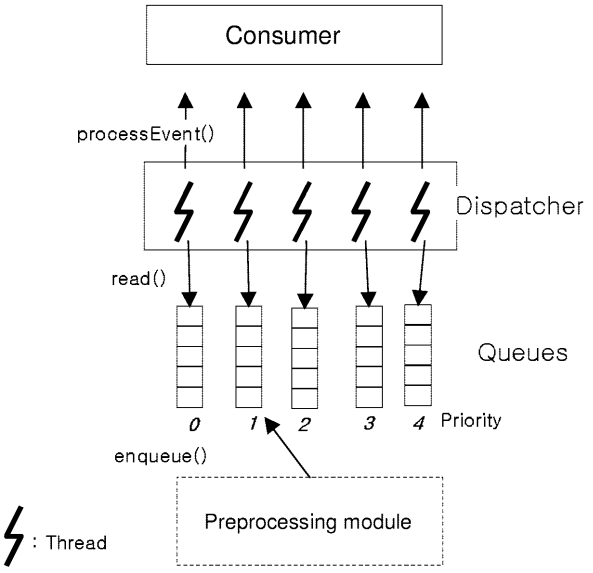


Fig. 7. Architecture of priority queue and dispatcher

In Figure 7, the preprocessing module determines the priority of an event, and then assigns the event to the appropriate priority queue. The dispatcher module consists of threads that have different priorities corresponding to the assigned priority queue. Each thread dispatches an event to a linked consumer and executes an event processing task called the `processEvent()` method of the consumer. Because each thread has associated priorities, higher priority dispatcher threads preempt lower priority dispatcher threads using OS kernel priority scheduling. By associating appropriate priorities to each thread, the dispatcher can take advantage of the OS kernel support of preemption.

7 Real-Time Device Driver for Backbone Protocol Adaptor

IEEE1394 is presently used as the backbone network in the proposed architecture, as a result, various multimedia data plus real-time event data for controlling and monitoring home devices can share an IEEE 1394 bus. Therefore, if a real-time response is not guaranteed in the device driver for the IEEE 1394 network adaptor, the middleware-level real-time features, such as the event channel, will be useless. In particular, because all event data must be sent through an asynchronous channel even though multimedia data can use an isochronous channel in an IEEE 1394 bus, the guarantee of a real-time response for asynchronous channel data is an important issue.

To solve this problem, a new real-time device driver is proposed for the IEEE1394 adaptor used as the backbone adaptor in the proposed architecture. The proposed IEEE1394 device driver architecture prevents packet-level priority inversion using priority-based queues. As such, a high priority packet processing time can be predicted

and the round trip time and jitter time of a high priority packet can be reduced. Furthermore, to reduce interrupt latency, which has an important effect on the real-time guarantee for a device driver communicating with network adapters through hardware interrupts, the processing time in the interrupt context can be reduced by processing received packets in the user-level threads rather than the ISR(Interrupt Service Routine). Accordingly, the proposed IEEE1394 network device driver architecture can guarantee real-time characteristics and be applied to the device driver of a subnet gateway because IEEE1394 is used as the backbone protocol.

In the device driver, to prevent packet-level priority inversion, each unit in the link layer and transaction unit contains priority-based queues that enable packet processing according to the priority of the packet. The asynchronous link unit distinguishes the transmitted packets on the basis of their application priorities and classifies them into appropriate queues and threads. Finally, to reduce any interrupt latency, which has a significant influence on guaranteeing real-time characteristics in a device driver, the packet processing time in the ISR is reduced by processing the received packets in the user-level threads instead of the ISR.

Although the above concept is very important for achieving a real-time response in the proposed architecture, it is not included in the core software components of the proposed middleware layer. Consequently, a detailed discussion on this issue is outwith the scope of the current paper. For a more detailed description of the proposed real-time device driver refer to paper [19] also written by the current authors.

8 Implementation and Experimentation

8.1 Implementation

The core components of ROOM-BRIDGE, including the priority-based event channel [18], replication model of a resource repository [17], and real-time device driver for IEEE1394 [19] have already been developed and their performances individually evaluated. Accordingly, in this study a full-scale version was implemented in a testbed home network with the cooperative support of Kyungpook National University and ETRI(http://www.etri.re.kr/). As shown in Figure 8, three subnets were organized as the real testbed of a home network, and each subnet was connected to the IEEE1394 based backbone network through individual ROOM-BRIDGE servers as the subnet gateways. Two tiny diskless SBCs(Single Board Computers) based on X86 CPU were used as the ROOM-BRIDGE servers and dedicated hardware platform, plus a PC with a TI IEEE1394 adaptor board was used as the third ROOM-BRIDGE server and subnet gateway. The core components of ROOM-BRIDGE in subnets 1 and 2 were implemented under a VxWorks [20] real time operating system with JDK(Tornado for Java). For the backbone and subnet, MotionIO OHCI-104 IEEE1394 adaptors were installed into the ROOM-BRIDGE servers of subnets 1 and 2. In addition, a DI-10(for digital input devices) and DO-10(for digital output devices) were connected into each ROOM-BRIDGE server as the LonTalk-based control network nodes. Gesytec's LonTalk adaptor boards were used for accommodating a LonTalk network in the

subnet. Two digital multimedia devices(Sony DCR-TRV510 digital camcorder and La Cie 37G Firewire Hard Drive as a digital storage for multimedia data) were used as the IEEE1394-based home devices and connected to each subnet, as shown in Figure 8.

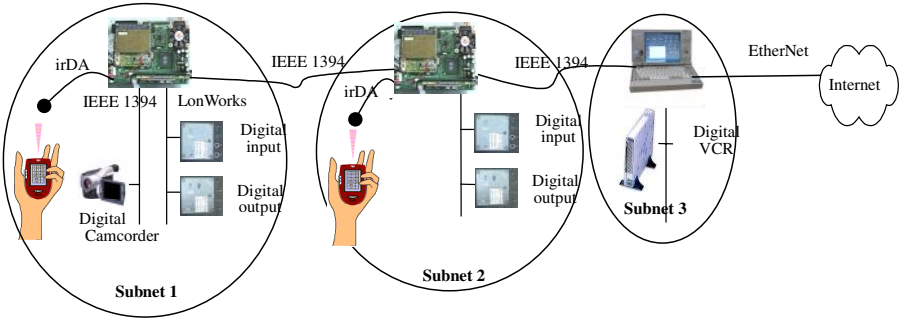


Fig. 8. Hardware configuration of prototype home network

The subnet gateway(subnet3 in Figure 8) used Windows NT with jdk1.3, which was connected to the Internet through an Ethernet adaptor. The core components of the ROOM-BRIDGE middleware, as mentioned in section 3.3, were embedded into each ROOM-BRIDGE server.

To implement the proposed concept more easily, a java-based programming environment was used for implementing the core ROOM-BRIDGE components, such as the event channel and resource repository architecture. To support the replicated repository space, the Space programming model [11] and Beans feature in Java were used. However, all the device drivers for each protocol adaptor were implemented in C and connected into Java using JNI(Java Native Interface).

9 Conclusions and Future Work

This research proposed a new design architecture and related middleware, called ROOM-BRIDGE, for a home network that needs to support ubiquitous computing and consumer devices and application-level multi-agents to meet real-time and distributed constraints. The proposed architecture was tested using a proof-of-concept prototype based on a testbed home network, and showed promise for improving home network service performance under real-time distributed environments. The full functionality of the proposed ROOM-BRIDGE architecture is still being investigated.

According to the evaluation results, the benefits of the proposed architecture can be summarized as follows:

- No extra device-level burden or consideration of consumer devices for supporting the proposed middleware: Since the core components of ROOM-BRIDGE are only installed and work on the level of subnet gateways, called ROOM-BRIDGE servers, ubiquitous devices as leaf nodes of a home network do not require any

extra embedded module, except for a network interface to support the proposed middleware.

- Easy support of heterogeneous protocols and ubiquitous devices whether or not they are a networked device: Even though the necessity of supporting a new protocol occurs in a subnet, only a network manager component and device driver for the protocol adaptor are added to the ROOM-BRIDGE server, thus no modification or reconfiguration is necessary for the entire network.
- Physical and logical group-based device commands: Because a home network consists of several vertical groups of subnets, such as rooms, the kitchen, and so on, an agent or home member can issue logical group-based device controlling commands, for example “turn off all lights in room 5”
- Reliable and fault-isolated architecture: Due to the replication model of the resource repository by room in the proposed architecture, the performance of the entire home network is less affected by the overload or fault of an individual subnet. In addition, the performance of the entire network is not linearly affected by an increasing ratio of application agents.

In further studies, isochronous features, such as those of HAVi spec. [6] in the proposed middleware, will be considered for enhancing the features of multimedia data communication through the isochronous channel of IEEE1394, along with the planning and design of an agent-level middleware as an upper layer application of the ROOM-BRIDGE components.

Acknowledgement. The work reported in this paper was partially supported by ETRI(Electronics and Telecommunications Research Institute) from 1998 to 2001, and IITA(Institute of Information Technology Assessment) from 1998 to 1999.

References

1. Gerard O’ Dricoll, Essential Guide to Home Networking Technologies, Prentice Hall PTR,2000
2. Dutta-Roy.A, "Networks for Home", IEEE Spectrum, Volume 36, December 1999.
3. E. Douglas Jensen,” Scaling up Real-Time Computing to Higher Levels in the Enterprise: A Grand Challenge,” IEEE Workshop on Real-Time Mission-Critical Systems, Nov. 30, 1999
4. LonTalk Protocol Specification Version 3.0, 1994.
5. IEEE1394, Std for High Performance Serial Bus, 1995
6. Specification of the Home Audio/Video Interoperability (HAVi) Architecture Version 1.0, January 18 2000.
7. Tim Kowk,” *A vision for residential broadband services: ATM-to-the-Home*,” IEEE Network, 9(5), Sept.-Oct. 1995.
8. Richard Greenane and Simon Dobson,” Integrating LonWorks into an open systems control environment,” LonWorld’99, Echelon, 1999.
9. Jini Architecture Specification Revision 1.0, Sun microsystems, Jan. 1999.
10. N. Carriero and D. Gelernter, “Linda in Context,” Comm. ACM, vol.32, no.4, Apr. 1989.

11. E. Freeman, S. Hupfer, and K. Arnold , JavaSpaces Principles, Patterns, and Practice, (Addison-Wesley, 1999)
12. T.H. Harrison, D. L. Levine, and D. C. Schmidt, " The design and performance of a real-time CORBA event service," Int'l Conf. On Object-oriented Programming, Systems, Languages and Applications, 1997.
13. L. Bergmans, A. van Halteren, L. Ferreira Pires, M. van Sinderen, M. Aksit," A QoS-control architecture for object middleware,"7th Intl. Conf. on *Interactive Distributed Multimedia Systems and Telecommunication Services* (IDMS 2000)
Enschede, Netherlands, October 2000 LNCS 1905, Springer, 2000, 117-131.
14. L. Bergmans, M. Aksit," Aspects and crosscutting in layered middleware systems," *Reflective Middleware* (RM 2000) workshop held in conjunction with the IFIP/ACM Intl. Conf. on Distributed System Platforms and Open Distributed Processing (Middleware 2000) New York, USA, April 2000.
15. V. Kalogeraki, P.M. Melliar-Smith, and L.E. Moser. *Soft Real-Time Resource Management in CORBA Distributed Systems*. In Proceedings of IEEE Workshop on Middleware for Distributed Real-time Systems and Services, pages 46--51. IEEE, December 1997. San Francisco, CA, USA.
16. G. Banavar, T. Chandra, R. Strom and D. Sturman, "*A case for message oriented middleware*", Proc. Of Symp. On Distributed Systems, DISC99, Bratislava, September 1999, LNCS. Vol. 1693.
17. Jae Chul Moon and Soon Ju Kang, " Multi-Agent Architecture for Intelligent Home Network Service Using Tuple Space Model," IEEE Trans. Consumer Electronics Vol 46, Aug. 2000.
18. Jae Chul Moon, Jun Ho Park, and Soon Ju Kang, "An Event Channel-Based Embedded Software Architecture for Developing Telemetric and Teleoperation Systems on the WWW," Proc. IEEE Real-Time Technology and Applications Symp., Jun. Vanquaver, Canada 1999.
19. Dong Hawn Park and Soon Ju Kang,"IEEE1394 OHCI Device Driver Architecture for Guarantee Real-Time Requirement," 7th Intl' Conf. on RTCSA 2000,Cheju, KOREA
20. VxWorks Programmer's Guide, WindRiver Systems, Mar 1997.
21. L. Wang, S. Balasubbramanian and D. H. Norrie, " Agent-based Intelligent Control System Design for Real-Time Distributed Manufacturing Environments," Agent-based Manufacturing Workshop – Autonomous Agents '98, Minneapolis/St. Paul, May 9-13, 1998

Reducing the Energy Usage of Office Applications

Jason Flinn[1], Eyal de Lara[2], Mahadev Satyanarayanan[1], Dan S. Wallach[3], and Willy Zwaenepoel[3]

[1] School of Computer Science, Carnegie Mellon University
[2] Department of Electrical and Computer Engineering, Rice University
[3] Department of Computer Science, Rice University

Abstract. In this paper, we demonstrate how component-based middleware can reduce the energy usage of closed-source applications. We first describe how the Puppeteer system exploits well-defined interfaces exported by applications to modify their behavior. We then present a detailed study of the energy usage of Microsoft's PowerPoint application and show that adaptive policies can reduce energy expenditure by 49% in some instances. In addition, we use the results of the study to provide general advice to developers of applications and middleware that will enable them to create more energy-efficient software.

1 Introduction

Battery energy is one of the most critical resources for mobile computers. Despite considerable research effort, no silver bullet for reducing energy usage has yet been found. Instead, a comprehensive effort is needed—one that addresses all layers of the system: hardware, operating system, middleware, and applications.

One promising piece of a comprehensive solution is *energy-aware adaptation*, in which applications modify their behavior to reduce their energy usage when battery levels are critical. The potential benefits of energy-aware adaptation were first explored in the context of Odyssey [6]. That work showed that energy-aware applications can often significantly extend the battery lifetimes of the laptop computers on which they operate by trading fidelity, an application-specific metric of quality, for reduced energy usage. However, since only multimedia, open-source applications running on the Linux operating system were studied, it was not clear that this technique would be relevant to the Windows office applications that users commonly run on laptop computers.

Several important questions follow: Can one show significant energy reductions for the type of office applications that users most commonly execute on laptop computers? Is it possible to add energy-awareness to applications for which source code is unavailable? Is this approach valid for applications executing on closed-source operating systems (i.e. Windows)?

In this paper, we describe how Puppeteer [2], a component-based middleware system, allows us to add energy-awareness to applications. Puppeteer takes

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 252–272, 2001.

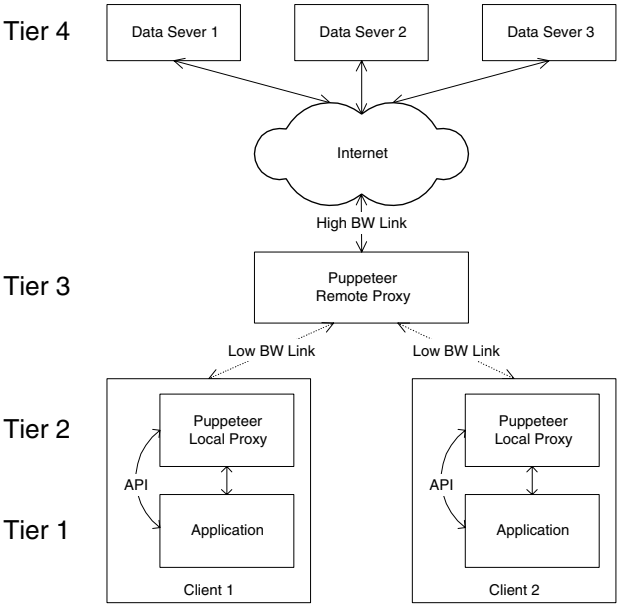


Fig. 1. Puppeteer architecture

advantage of well-defined interfaces exported by applications to modify their behavior without source code modification. We demonstrate the feasibility of this approach by studying energy saving opportunities for Microsoft's popular PowerPoint application. By using Puppeteer to distill multimedia content from presentations stored on remote servers, we reduce the energy needed to load presentations by 49%. Further, the benefits of distillation extend to reducing energy use while the document is being edited and saved.

In addition, we identify several instances where PowerPoint can be made more energy-efficient. From these specific instances, we present general advice for developers of applications and middleware that will enable them to create more energy-efficient software in the future.

In the next section, we provide an overview of Puppeteer. In Section 3, we describe our energy measurement methodology. Section 4 shows how we can modify PowerPoint behavior to reduce energy usage when battery levels are critical. Section 5 discusses opportunities for making PowerPoint and similar applications more energy-efficient. In Section 6, we speculate on the applicability of component-based adaptation to applications other than PowerPoint, and in the remainder of the paper, we discuss related work and conclude.

2 Puppeteer

Puppeteer is a system for extending component-based applications, such as Microsoft Office or Internet Explorer, to support adaptation in mobile environ-

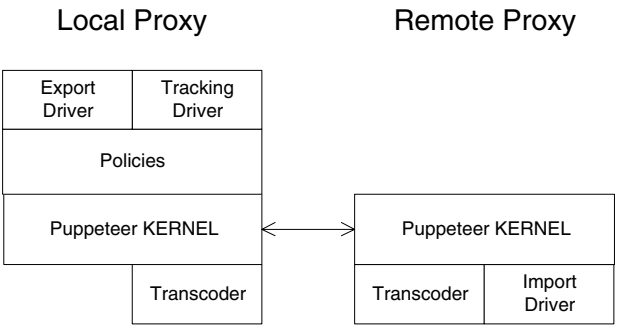


Fig. 2. Puppeteer local and remote proxy architectures

ments. It uses the exported APIs of applications and the structured nature of the documents they manipulate to implement adaptation without changing source code. It supports subsetting adaptation (where only a portion of the elements of a document are provided to the application; for instance, the first page), and versioning adaptation (where a different version of some of the elements is provided to the application; for instance, a low-resolution version of an image). It uses the document structure to extract subsets and versions of the document. Furthermore, Puppeteer uses the exported APIs of the applications to incrementally increase the subset of the document or improve the version of the elements available to the application. For instance, it uses the APIs to insert additional pages or higher-resolution images into the application.

Figure 1 shows the four-tier Puppeteer system architecture. It consists of applications to be adapted, Puppeteer local proxies, the Puppeteer remote proxy, and the data servers. Applications and data servers are completely unmodified. Data servers can be arbitrary repositories of data such as Web servers, file servers or databases. All communication between applications and data servers goes through the Puppeteer local and remote proxies, which work together to perform the adaptation. The Puppeteer local proxy manipulates the running application through a subset of the application's external programming interface.

Figure 2 shows the architecture of the Puppeteer local and remote proxies. It consists of application-specific policies, component-specific drivers, type-specific transcoders and an application-independent kernel. The Puppeteer local proxy is in charge of executing adaptation policies. The Puppeteer remote proxy is responsible for parsing documents, exposing their structure, and transcoding components as requested by the local proxy.

The adaptation process in Puppeteer is divided roughly into three stages: parsing the document to uncover the structure of the data, fetching selected components at specific fidelity levels, and updating the application with the newly fetched data.

When the user opens a document, the Puppeteer remote proxy instantiates an *import driver* for the appropriate document type. The import driver parses the document, and extracts its component structure (the *skeleton*, a tree structure)

and the data associated with the nodes in the tree. Puppeteer then transfers the document's skeleton to the Puppeteer local proxy. The policies running on the local proxy fetch an initial set of elements from within the skeleton at a specified fidelity. These policies may be static, or may depend on *tracking drivers* that detect the occurrence of certain events, such as moving the mouse over an image, causing the image to be loaded.

Puppeteer uses the *export driver* for the particular document type to supply this set of components to the application as though it had the full document at its highest level of fidelity. The application, believing that it has finished loading the document, returns control to the user. Meanwhile, Puppeteer knows that only a fraction of the document has been loaded and will use the application's external programming interface to incrementally fetch remaining components or upgrade their fidelity.

3 Measurement Methodology

All measurements were collected in a client-server environment. Microsoft PowerPoint 2000 executes on the client: a 233 MHz Pentium IBM 560X laptop with 96 MB of memory. The server is a 400 MHz Pentium II desktop with 128 MB of memory. Both machines run the Windows NT 4.0 operating system. The machines communicate using a 2 Mb/s Lucent WaveLan wireless 802.11 network.

We measured client energy usage with a HP3458a digital multimeter. When collecting data, we attached the multimeter's probes in series with the external power input of the client laptop and removed the client's battery to eliminate the effects of charging. We also connected an output pin of the client's parallel port to the external trigger input of the multimeter—this allowed the multimeter and client to coordinate during the taking of measurements.

We created a dynamic library that allows a calling process to precisely indicate the start and end of measurements. The process calls the `start_measuring` function which records the current time and toggles the parallel port pin. Once the pin is toggled, the multimeter samples current levels 1357.5 times per second. When the measured event completes, the application calls `stop_measuring`, which returns the elapsed time since `start_measuring` was called.

To calculate total energy usage, we first derive the number of samples, n, that were taken before `stop_measuring` was called by multiplying the elapsed measurement time by the sample rate. The mean of the first n samples is the average current level. Multiplying this value by the measured voltage for the laptop power supply (which varies by less than 0.25% in the course of an experiment) yields the average power usage. This is multiplied by the elapsed time to calculate total energy usage.

We assume aggressive power management policies. All measurements were taken using a disk-spindown threshold of 30 seconds (the minimum allowed by Windows NT). Unless otherwise noted, the wireless network uses standard 802.11 power management. Audio input and output is disabled. However, the display is

Presentation	Full-Quality Size (MB)	Distilled Size (MB)	Ratio
A	15.02	1.67	0.11
B	11.42	0.47	0.04
C	7.26	0.83	0.11
D	3.11	3.11	1.00
E	2.23	1.32	0.59
F	1.72	0.11	0.07
G	1.07	0.36	0.34
H	0.87	0.75	0.86
I	0.20	0.20	1.00
J	0.08	0.08	1.00

Fig. 3. Sizes of sample presentations

not disabled during measurements since PowerPoint is an interactive application; instead it is reduced to minimum brightness.

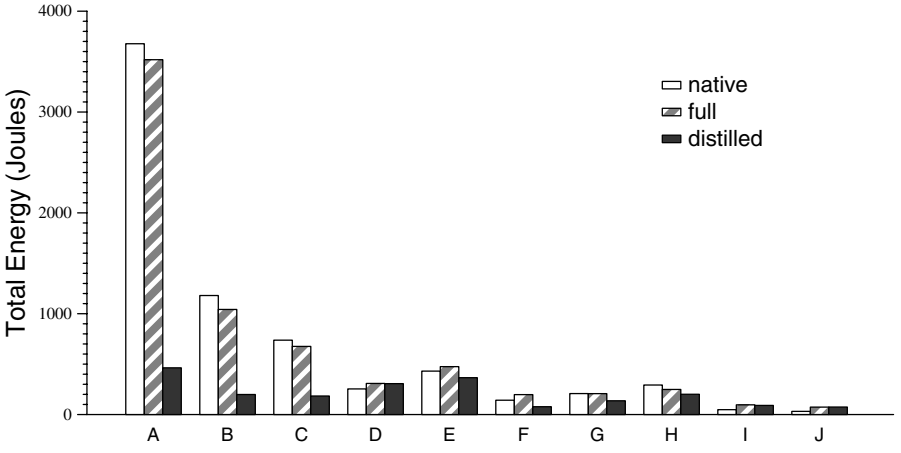
An ideal battery can be modeled as a finite store of energy. If an application expends some amount of energy to perform an activity, the energy supply available for other activities is reduced by that amount. In reality, batteries never behave ideally. As power draw increases, the total energy that can be extracted from the battery decreases. In addition, recovery effects may apply—a reduction in load for a period of time may result in increased battery capacity [11].

In this paper, we assume the ideal model for battery behavior, but note that most of the techniques proposed decrease average power usage. Thus, the gains reported here will be slightly understated.

4 Benefits of Adaptation

In this section, we examine whether component-based adaptation can be used to add energy-awareness to Microsoft PowerPoint. As with many office applications, PowerPoint enables users to incorporate increasing amounts of rich multimedia content into their documents—for example, charts, graphs, and images. Since these objects tend to be quite large, the processor, network, and disk activity needed to manipulate them accounts for significant energy expenditure. Yet, when editing a presentation, a user may only need to modify and view a small subset of these objects. Thus, it may be possible to significantly reduce PowerPoint energy consumption by presenting the user with a distilled version of a presentation: one which contains only the information that the user is interested in viewing or editing.

Puppeteer allows us to load a distilled version of a document when battery levels are critical. The distilled version initially omits all multimedia content not on the first or master slide. Placeholders are inserted into the document to



This figure shows the energy used to load ten PowerPoint presentations from a remote server. Native mode loads presentations from an Apache Web server. Full and distilled modes load presentations from a remote Puppeteer proxy, with full mode loading the entire presentation and distilled mode loading a lower-quality version. On average, distilled mode uses 60% of the energy of native mode. Each bar represents the mean of five trials—90% confidence intervals are sufficiently small so that they would not be visible on the graph.

Fig. 4. Energy used to load presentations

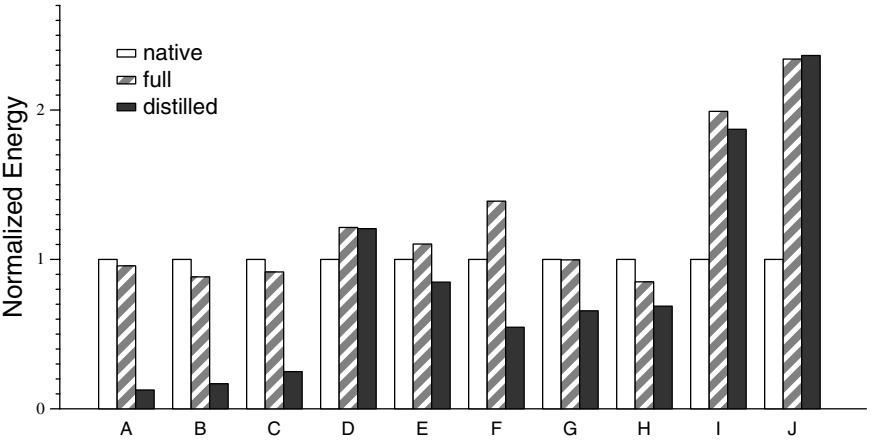
represent omitted objects. Later, if the user wishes to edit or view a component, she may click on the placeholder, and Puppeteer will load the component and dynamically insert it into the document. Thus, when users edit only a subset of a document, the potential for significant energy savings exists.

In Sections 4.1 and 4.2, we measure the impact of distillation on several activities commonly performed in PowerPoint. Then, in Section 4.3 we examine the impact of background tasks such as spell-checking and the Office Assistant. Finally, we quantify the energy benefit of modifying autosave frequency in Section 4.4.

4.1 Loading Presentations

We first examined the potential benefits of loading distilled PowerPoint presentations. We assume that presentations are stored on a remote server. There are many reasons to store documents in a remote, centralized location. Multiple users may wish to collaborate in the production of the document. Also, storage space on mobile clients may be limited. Finally, remote storage offers protection against data loss in the event that a mobile computer is damaged or stolen.

We measured the energy used to fetch presentations from the server and render them on the client. We chose a sample set of documents from a database of 1900 presentations gathered from the Web as described by de Lara et al. [1]. From the database, we selected ten documents relatively evenly distributed in



This figure shows the relative energy used to load ten PowerPoint presentations from a remote server, as described in Figure 4. For each data set, results are normalized to the amount of energy used to load the document in native mode.

Fig. 5. Normalized energy used to load presentations

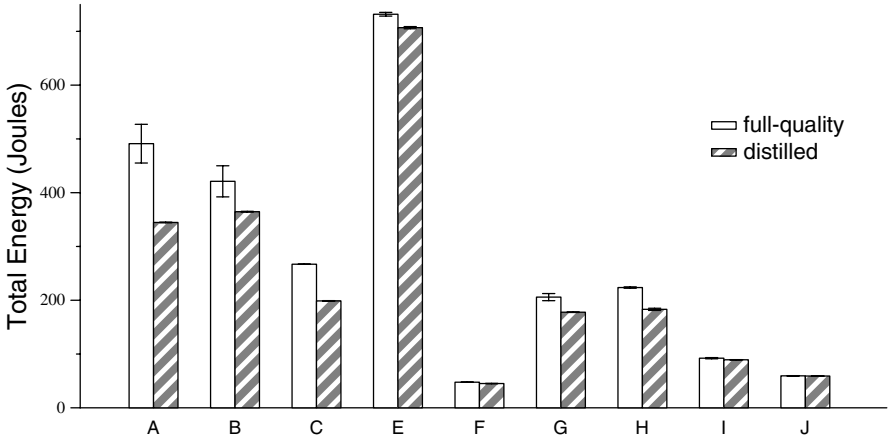
size. Figure 3 shows the sizes of these documents, as well as the reductions achieved by distillation. As might be expected, larger documents tend to have more multimedia content, although there is considerable variation in the data. For three documents (D, I and J), distillation does not reduce document size.

For each document, we first measured the energy used by PowerPoint to load the presentation from a remote Apache Web server (we will refer to this as "native mode"). We also investigated the cost of loading documents from a remote NT file system—these results are not shown since the latency and energy expenditure is significantly greater than when using the Web server.

We then measured the energy used to load each document from a remote Puppeteer proxy running on the same remote machine. We measured two modes of operation: "full", in which the entire document is loaded, and "distilled", in which a degraded version of the document is loaded.

Figure 4 shows the total energy used to fetch the documents using native, full, and distilled modes. In Figure 5 we show the relative impact for each document by normalizing each value to the energy used by native mode. The energy savings achieved by distillation vary widely. Loading a distilled version of document A uses only 13% as much energy as native mode, while distilling document J uses 137% more energy. On average, loading a distilled version of a document uses 60% of the energy of native mode.

Interestingly, full mode sometimes uses less energy to fetch a document than native mode. This is because fetching a presentation with Puppeteer tends to use less power than native mode. Thus, even though native mode takes less time to fetch a document, its total energy usage can sometimes be greater. Without



This figure shows the amount of energy needed to page through a presentation. For each data set, the left bar shows energy use when a full-quality presentation is loaded using native mode, and the right bar shows energy use for a reduced-quality version of the same presentation. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Fig. 6. Energy used to page through presentations

application source code, it is impossible to know for certain why Puppeteer power usage is lower than native mode. One possibility is more efficient scheduling of network transmissions—this issue will be discussed in Section 5.1.

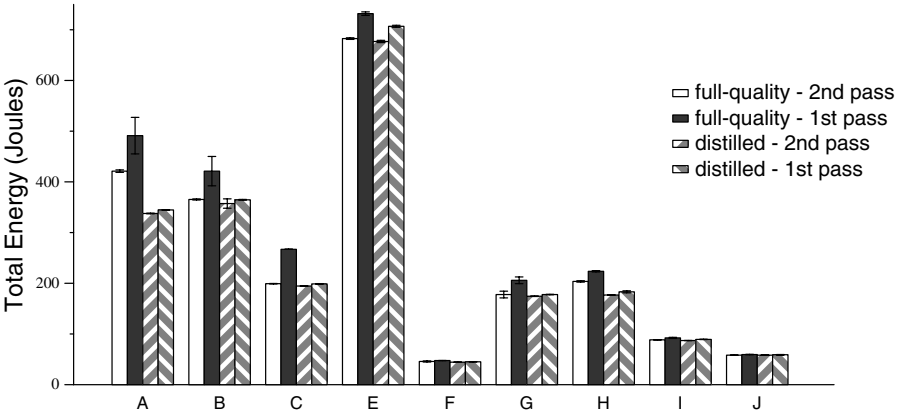
The results in Figures 4 and 5 show that while most documents benefit from distillation, some suffer an energy penalty. If Puppeteer could predict which documents will benefit from distillation, it could distill only those documents and fetch the remaining documents using native mode.

One possible prediction method is to distill only presentations that are larger than a fixed threshold, since small documents are unlikely to contain significant multimedia content. Analysis of the documents used in this study suggests that a reasonable threshold is 0.5 MB. A strategy of distilling only presentations larger than 0.5 MB does not distill documents I and J, and consequently, uses 52% of the energy of native mode to load the ten documents.

Alternatively, Puppeteer could distill only documents that have a percentage of multimedia content greater than a threshold. As shown in Figure 3, distillation does not reduce the size of three documents. If Puppeteer does not distill these documents, it uses only 51% of the energy of native mode.

4.2 Editing Presentations

We next measured how distillation affects the energy needed to edit a presentation. While it is intuitive that loading a smaller, distilled version of a document requires less energy, it is less clear whether distillation also reduces energy use



This figure shows the amount of energy needed to page through a presentation a second time. The energy needed to page through each presentation the first time is also shown for comparison. For each data set, the left two bars show energy use for a full-quality presentation loaded using native mode, and the right two bars show energy use for a reduced-quality version. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Fig. 7. Energy used to re-page through presentations

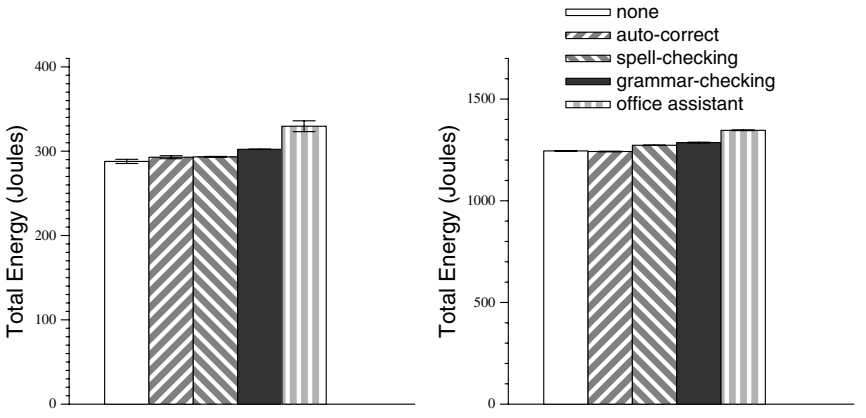
while a document is edited. Although Puppeteer does not yet support reintegration of changes made to a distilled copy of a document, we are currently adding this functionality to the system.

Naturally, energy use depends upon the specific activities that a user performs. While a definitive measurement of potential energy savings would require a detailed analysis of user behavior, we can reasonably estimate such savings by looking at the energy needed to perform common activities.

One very common activity is paging through the slides in a presentation. We created a Visual Basic program which simulates this activity by loading the first slide of a presentation, then sending `PageDown` keystrokes to PowerPoint until all remaining slides are displayed. After each keystroke, the program waits for the new slide to render, then pauses for a second to simulate user think-time.

We measured the energy used to page through both the full-quality and the distilled version of each document. Figure 6 presents these results for nine of the ten presentations in our sample set—presentation D is omitted because it contains only a single slide. As shown by the difference in height between each pair of bars in Figure 6, distilling a document with large amounts of multimedia content can significantly reduce the energy needed to page through the document. Energy savings range from 1% to 30%, with an average of 13%.

After PowerPoint displays a slide, it appears to cache data that allows it to quickly re-render the slide, thereby reducing the energy needed for redisplay. This effect is shown in Figure 7, which displays the energy used to page through each document a second time. For comparison, Figure 7 also shows the energy



(a) no interval (b) 100 ms. interval

This figure shows the amount of energy needed to perform background activities during text entry. The graph on the left shows energy use when text is entered without pause, and the graph on the right shows energy use with a 100 ms. pause between characters. Each bar shows the cumulative effect of performing background activities. The leftmost bar in each graph was measured with no background activities enabled, and the rightmost bar in each graph was measured with all background activities enabled. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Fig. 8. Energy used by background activities during text entry

needed to page through each document initially. Comparing the heights of corresponding bars shows that subsequent slide renderings use less energy than initial renderings. Thus, the benefits of distillation are smaller on subsequent traversals: ranging from negligible to 20% with an average value of 5%.

4.3 Background Activities

We next measured the energy used to perform background activities such as auto-correction and spell-checking. Whenever a user enters text, PowerPoint may perform background processing to analyze the input and offer advice and corrections. When battery levels are critical, Puppeteer could disable background processing to extend battery lifetime.

We measured the effect of auto-correction, spell-checking, style-checking, and the Office Assistant (Paperclip). We created a Visual Basic program which enters a fixed amount of text on a blank slide. The program sends keystrokes to PowerPoint, pausing for a specified amount of time between each one.

Figure 8(a) shows the energy used to enter text with no pause; Figure 8(b) shows energy usage with a 100 ms. pause between keystrokes. We first measured energy usage with no background activities, and then successively enabled auto-correction, spell-checking, style-checking, and the Office Assistant. Thus, the

difference between any bar in Figure 8 and the bar to its left shows the additional energy used by a specific background activity. For example, the difference between the first two bars in each chart shows the effect of auto-correction.

Auto-correction expends negligible energy when entering text—the additional energy cannot be distinguished from experimental error. Spell-checking and style-checking incur a small additional cost. With no pause between entering characters, these options add a 5.0% energy overhead, but with a 100 ms. pause between characters, the overhead is only 3.3%.

The Office Assistant incurs a more significant energy penalty. With no pause between typing characters, enabling the Assistant leads to a 9.1% increase in energy use. With a 100 ms. pause, energy use increases 4.9%. In fact, even when the user is performing no activity, enabling the Office Assistant still consumes an additional 300 mW., increasing power usage 4.4% on the measured system. Adaptively disabling the Office Assistant can therefore lead to a small but significant extension in battery lifetime.

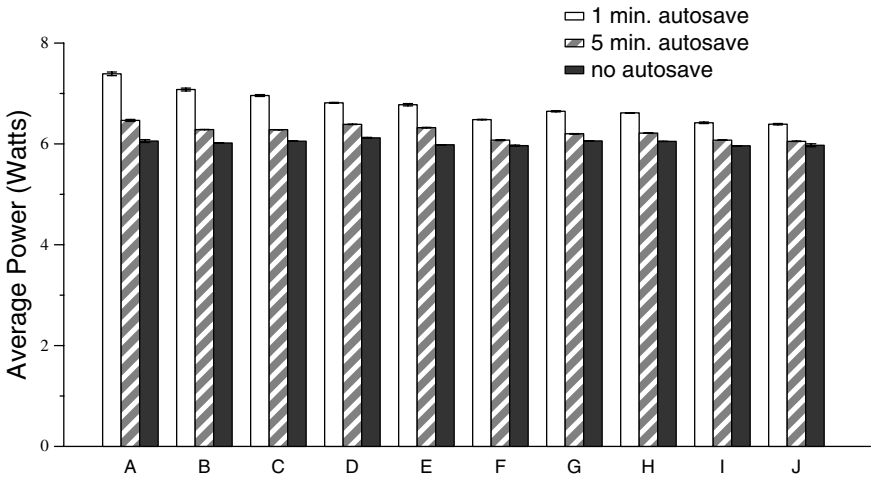
4.4 Autosave

Autosave frequency is another potential dimension of energy-aware adaptation. After a document is modified, PowerPoint periodically saves an AutoRecovery file to disk in order to preserve edits in the event of a system or application crash. Autosave may be optionally enabled or disabled—if it is enabled, the frequency of autosave is configurable. Since autosave is performed as a background activity, it often will have little effect upon perceived application performance. However, the energy cost is not negligible: the disk must be spun up, and, for large documents, a considerable amount of data must be written.

Since periodic autosaves over the wireless network would be prohibitively slow, we assume that documents are stored on local disk. We created a Visual Basic program which loads a PowerPoint document, makes a small modification (adds one slide), and then observes power usage for several minutes, during which no further modifications are made. To avoid spurious measurement of initial activity associated with loading and modifying the presentation, the program waits ten minutes after making the modification before measuring power use.

Figure 9 shows power usage for three autosave frequencies. For each presentation, the first bar shows power usage when the full-quality version of the document is modified and a one minute autosave frequency is specified. The next bar shows the effect of specifying a five minute autosave frequency. The final bar shows the effect of disabling autosave—this represents the minimum power drain that can be achieved by modifying the autosave parameters.

As can be seen by the difference between the first two bars of each data set, changing the autosave frequency from 1 minute to 5 minutes reduces power usage from 5% to 12%, with an average reduction of 8%. The maximum possible benefit is realized when autosave is disabled. As shown by the difference between the first and last bars in each data set, this reduces power usage from 7% to 18% with an average reduction of 11%. Thus, depending upon the user's willingness



This figure shows how the frequency of PowerPoint autosave affects power usage. The three bars in each data set show power use with a 1 minute autosave frequency, with a 5 minute autosave frequency, and with autosave disabled. Each bar represents the mean of five trials—the error bars show 90% confidence intervals.

Fig. 9. Effect of autosave options on application power usage

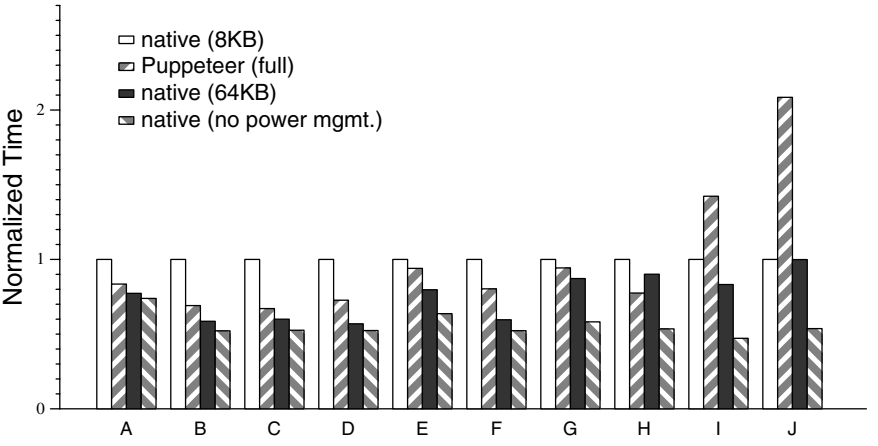
to hazard data loss in the event of crashes, autosave frequency is a potentially useful dimension of energy-aware adaptation.

5 Energy-Efficiency

As shown in the previous section, there are many opportunities to conserve energy usage by modifying PowerPoint behavior. However, these opportunities require one to sacrifice some dimension of application-specific quality in order to extend battery lifetime. Thus, before employing adaptive strategies, one should ensure that the application is as energy-efficient as possible, i.e. that it consumes the minimum amount of energy needed to perform its function.

Increasing software energy-efficiency extends battery lifetime without degrading application quality. Further, it increases the effectiveness of energy-aware adaptation, since the relative savings achieved by adaptive strategies will be greater if the fixed cost of executing the application is lower.

During our study, we discovered several areas in which PowerPoint could be more energy-efficient. In this section, we show that with component-based adaptation, Puppeteer can often increase PowerPoint's energy-efficiency without source code modification. Further, while we observed these areas in the context of a single application, we believe that the principles behind them are general enough so that they can be applied by most software developers. From these



This figure shows how network parameters influence the time needed to load applications from a remote server. The first bar in each data set shows the time needed when a document is loaded from Apache using default network settings. The second bar shows the time needed when the document is loaded from a remote Puppeteer proxy. The third bar shows time to load the document from Apache with 64 KB socket buffer and TCP receive window sizes. The final bar shows time to load the document from Apache with network power management disabled. Each bar represents the mean of five trials—90% confidence intervals are sufficiently small that they would not be visible on the graph.

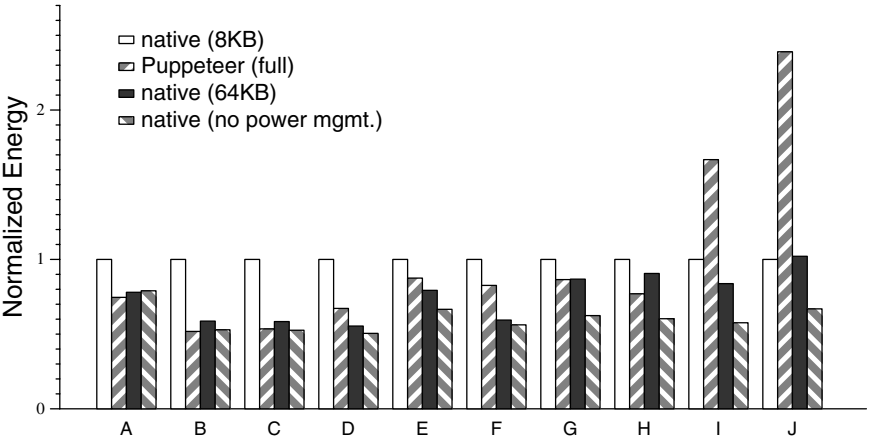
Fig. 10. Effect of power management on time needed to load presentations

specific observations, we formulate some general principles that can be used to make applications and middleware more energy-efficient.

5.1 Transparent Power Management

Layering of functionality is a well-known software engineering principle. It reduces software complexity by allowing developers to implement and modify each layer without needing to know details about other layers. However, layered implementations sometimes exhibit suboptimal performance because opaque layering precludes optimizations that span multiple layers.

Hardware power management is often implemented using a layered approach. Each device (network, disk, CPU, etc.) individually implements power management algorithms without considering application or other system activity. Typically, such algorithms use fixed timeouts—for example, hard disks enter power saving states after periods of inactivity, and wireless network clients periodically disable their receivers. On the other hand, applications normally do not consider power management when scheduling their activities. In our study of PowerPoint behavior, we have found two instances where this layered approach leads to unnecessary energy expenditure and suboptimal performance.



This figure shows how network parameters influence the energy needed to load applications from a remote server. The scenarios used to generate the bars in each data set are identical to those described in Figure 10. Each bar represents the mean of five trials—90% confidence intervals are sufficiently small that they would not be visible on the graph.

Fig. 11. Effect of power management on energy needed to load presentations

Transparency in network power management. Figures 10 and 11 show the normalized time and energy needed to load PowerPoint documents from a remote server. The first bar of each data set shows results for the most naive method—loading the presentation from Apache employing the default network settings, including those for power management. For comparison, the second bar in each data set, shows results when Puppeteer loads the presentation.

Initially, we were greatly surprised that Puppeteer often takes significantly less time and energy to fetch a presentation than when the document is fetched directly from Apache. Puppeteer uses up to 33% less time and 48% less energy than native mode. Since Puppeteer parses the presentation on the server and reconstructs it on the client, the time and energy needed to fetch a document with Puppeteer should be greater. Although this discrepancy puzzled us initially, we now believe that the answer lies in the interaction of network power management and the communication patterns used by the two methods.

To save energy, the wireless network client disables its receiver when no incoming packets are waiting at the network base station. While the receiver is disabled, incoming packets are queued. Every 100 ms., the client restarts its receiver and checks if new packets have arrived. Since the wireless network used in this study has a 2 Mb/s nominal bandwidth, up to 25 KB of data may be queued at the base station while the receiver is disabled. This has the effect of giving the wireless network a high $bandwidth * delay$ product.

Native mode uses a single HTTP/1.1 connection to fetch data from the server, while Puppeteer fetches data using four simultaneous connections. Since the de-

fault Windows and Apache settings specify 8 KB socket buffer and TCP window sizes, a single TCP connection cannot fully utilize the wireless network. However, Puppeteer uses four simultaneous connections—this artifact of its implementation allows it to achieve far better network utilization.

The third bar in each data set shows the effect of increasing the socket buffer and TCP window sizes to 64 KB. This compensates for the high *bandwidth∗delay* product of the wireless network—native mode now fetches presentations, on average, 26% faster and uses 26% less energy than with the 8 KB defaults. Given these results, we chose to use the 64 KB sizes as the default for measurements presented in this paper (i.e., for native mode in Section 4.1).

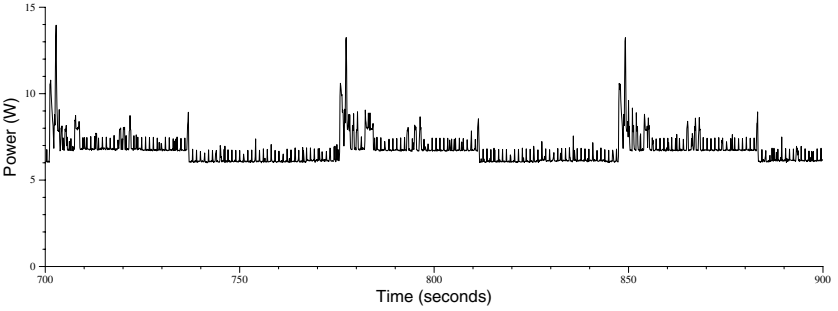
However, adjusting these parameters does not fully compensate for the effect of power management. The last bar in each data set shows results when network power management is disabled. Loading a document without power management uses 22% less time and 18% less energy than when using power management and 64 KB buffers. The remaining performance and energy penalty may be caused by a variety of factors, including TCP ack compression.

These results show the potential benefit of transparent power management. The most naive approach to loading documents incurs a significant time and energy penalty. This penalty can be reduced if the power management layer exposes details about power management strategies. Puppeteer, acting as a proxy for PowerPoint, could then take corrective action by increasing socket buffer and TCP window sizes, or by opening multiple network connections for data transfer. An even more promising approach is for applications and middleware to expose details about their behavior to the power management layer. If Puppeteer were to indicate large data transfers, network power management could be disabled for the duration of the transfer. The benefit of such an approach is shown by the difference between the first and last bars in each data set in Figures 10 and 11.

Transparency in disk power management. Transparency can also improve disk power management. To illustrate this, we specified an autosave frequency of one minute and modified a presentation as described in Section 4.4. Figure 12 shows resulting power drain over time. For clarity, we show only a portion of our measurements, ranging from 700 to 900 seconds after the modification.

The variation in power drain can be attributed to the interaction of PowerPoint autosave and disk power management. At approximately 700 seconds, a large power spike is caused by PowerPoint's writing of the AutoRecovery file and the resulting spin-up of the hard drive. After the write completes, the disk spins for 30 seconds—power usage remains relatively steady since the application is idle. Windows then spins down the hard drive, and power usage remains steady at a lower rate for 30 seconds. At approximately 775 seconds, the pattern repeats, since one minute has passed since PowerPoint last wrote the AutoRecovery file.

At first glance, disk power management appears effective since the energy saved by spinning down the disk for 30 seconds is greater than the excess energy caused by transitions. However, with additional knowledge about application behavior, the power management layer could do even better.



This figure shows how PowerPoint power usage varies over time with a specified autosave frequency of one minute. To generate the measurements, a document was loaded into PowerPoint and modified. The x-axis gives the amount of time that has passed since the modification was performed.

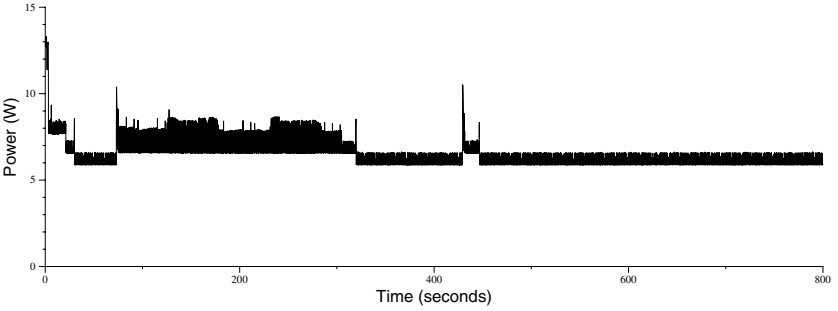
Fig. 12. PowerPoint power usage with 1 minute autosave

The OS saves energy by spinning down the hard drive if the disk remains in a low-power state long enough to counterbalance the energy cost of power-state transitions. Timeout-based power management policies [3,4,9,15] are based on the observation that disk accesses are often closely correlated together in time. Immediately after an access is observed, it is likely that another access will be seen soon. Spinning down the disk is undesirable, since the next access will likely occur before the break-even point for energy savings. However, as time passes without an access, the likelihood of another access happening soon decreases. Eventually, the estimated energy impact of spinning down the disk becomes favorable, and the OS transitions the hard drive to a low-power state.

In Figure 12, the disk power management layer could benefit if Puppeteer were to specify that autosave occurs at regular intervals and is uncorrelated with other activity. If Puppeteer indicates when autosave is occurring, the power management algorithm could omit these disk accesses from its prediction algorithm (since they are uncorrelated with other accesses). In Figure 12, the disk would spin down immediately after autosave completes, saving 19.2 Joules per autosave. This represents a significant 4% relative reduction in energy use. Potentially, the power management layer could also use such knowledge to anticipate future behavior—for example, it might forego spinning down the disk when it anticipates that an autosave will occur in the near future.

5.2 Minimizing Low-Priority Activities

Complex applications such as PowerPoint often perform background activities using low-priority threads. Examples include animation effects that lead to more pleasing user interfaces and data reorganizations that optimize future performance. When a computer operates on wall-power, such activities are scheduled



This figure shows how PowerPoint power usage varies after loading a document and performing no other activity. The x-axis gives the amount of time that has passed since the document was loaded.

Fig. 13. PowerPoint power usage after loading a presentation

whenever system resources such as the CPU are not needed by higher priority tasks—use of these resources is essentially free if they are uncontended.

However, when a computer operates on battery power, background computation is no longer free because it expends the finite supply of energy. Thus, developers should reevaluate these activities and only perform them if the potential benefit is greater than the cost of reduced battery lifetime.

Our study of PowerPoint revealed two instances in which low-priority tasks account for significant energy usage. The first is animation of the Office Assistant as discussed in Section 4.3. Even when the user is performing no activity, enabling the Assistant increases energy usage 4.4%, most likely due to animation of the Assistant's graphic image. When Puppeteer detects that the client is operating on battery power, it could adaptively disable the Office Assistant to save power.

Such animation effects are not restricted to PowerPoint. Web browsers, for instance, usually have a number of graphic devices to indicate activity while Web data is being fetched. While such animation can create more pleasing user experiences, application developers should carefully consider energy costs when designing interfaces. It may be more advisable to curtail unnecessary animation if a computer is operating on battery power.

The second instance of background energy consumption is more complex. After PowerPoint loads a document, we observe a variable amount of background energy consumption. Because this activity is extremely correlated with loading a presentation, we attribute it to application activity or OS activity performed on behalf of the application. Without source code, we cannot confirm the precise nature of the activity, but we believe it is caused by PowerPoint creating a backup of the presentation on disk. However, the duration of the activity is longer than one might expect (about 4 minutes to create a 15 MB file)—this may be due to inefficiency or the desire to prevent interference with foreground activity.

This activity has minimal impact on performance, probably because it is as a low-priority task. However, as shown in Figure 13, the energy costs can be sub-

stantial. We generated this data by loading a presentation and then performing no further activities. The impact of the initial background activity is shown by increased power usage during the first 300 seconds. For the document shown in Figure 13, the activity increases PowerPoint energy usage by 383 Joules. Average power use during the first five minutes increases by over 20%.

Since the energy costs are substantial, it is unlikely that they can be amortized across future activity if PowerPoint does not execute for a long period of time. When on battery power, it may therefore be advisable to forego these activities. Alternatively, one could perform them in a more efficient manner, perhaps by assigning them a higher priority so they would complete faster. This would save energy by allowing the disk to spend more time in low-power states.

There is a strong correlation between presentation size and the magnitude of the initial energy cost. Thus, when Puppeteer distills a document before loading it, as discussed in Section 4, it reduces the impact of initial background activity. For large documents, loading a distilled version reduces initial energy costs 63%.

5.3 Event-Based Programming

Periodic activities are common in application and system software. Such activities include polling for event completion and updates of disk files to preserve modifications in the event of a future crash. The performance impact of periodic activities is often minimal. However, the energy impact can be considerably larger. Each time an activity is performed, it consumes energy. In addition, hardware components such as the CPU, network, and disk must be transitioned from low-power states to perform the activity, wasting further energy.

Where possible, a better alternative is to replace periodic activities with event-based implementations. For example, instead of using a polling loop to check for event completion, one can have the event execution trigger a callback function. Such changes, while quite simple, can dramatically reduce energy use.

Our examination of PowerPoint shows that periodic autosave is a significant energy expenditure. Consider Figure 12 as a motivating example. Three large power spikes at 700, 775, and 850 seconds represent energy used to save the AutoRecovery file. Unfortunately, this activity is performed periodically, even though the presentation is not modified during this time period. A more energy-efficient implementation could use an event-driven model where disk updates are performed only after a fixed amount of data has been modified. Although this implementation artifact is best addressed in the application, Puppeteer can fix this problem without modifying application source. When Puppeteer detects that no activity has occurred for a period greater than the current autosave frequency, it can disable autosave until it detects further activity. In Figure 12, this would completely eliminate the power spikes without sacrificing data consistency.

6 Discussion

The previous results show that we can significantly reduce PowerPoint energy usage by modifying application behavior. The key to realizing these benefits

is Puppeteer's component-based adaptation strategy, which modifies behavior without access to source code. However, we will be able to generalize these results to other, closed-source applications only if component-based adaptation proves to be widely applicable. Thus, it is useful to examine what characteristics of PowerPoint make it a good candidate for component-based adaptation, and to see if these characteristics exist in other applications.

PowerPoint has two main characteristics that allow Puppeteer to modify its behavior. It has a well-defined data format that lets Puppeteer parse the content of presentations. It also has an external API that allows Puppeteer to trigger application events, for example, redisplay of a slide. The API also notifies Puppeteer of external events such as change of focus. On the other hand, PowerPoint has no explicit interfaces for power management or degrading document quality. This is encouraging since it indicates that an application need not explicitly support energy-awareness to realize its benefits. If its interface is sufficiently rich, energy-awareness may be implemented entirely by proxy.

Based on these observations, what other applications are likely candidates for component-based adaptation? The remaining applications in Microsoft's Office suite are obvious candidates, as they have data formats and APIs similar to PowerPoint's. Web browsers such as Netscape Navigator are also good candidates. HTML is a well-defined data format, and Netscape's remote interface allows external applications to manipulate its behavior [8]. Database applications may also benefit, since they have standard interfaces for manipulating data such as ODBC and SQL. Thus, for many common applications, a component-based approach is likely to prove useful in implementing energy-aware adaptation.

When rich external APIs are unavailable, other approaches may suffice. For example the Visual Proxy [13] takes advantage of the structured nature of interactions between applications and window managers to extend the functionality of a Web browser and a word processor. Such approaches are less convenient than component-based adaptation, but, with some sweat, can extend the range of applicability of proxy-based solutions.

7 Related Work

We believe this study is the first to explore how one can reduce the energy usage of office applications. We extend the concept of energy-aware adaptation first introduced in Odyssey [6] to support applications and operating systems where source code is unavailable.

The concept of using distillation to reduce mobile client resource use has been previously explored by Fox et al. [7]. They show that dynamic distillation of data can allow mobile computers to adapt to poor-bandwidth environments.

Several recent research efforts have advocated cooperation between applications and the operating system in managing energy resources. The MilliWatt project [5,14] is developing a power-based API that allows a partnership between applications and the system in setting energy use policy. Neugebauer and McAuley [12] show how the Nemesis operating system can be extended to provide applications with detailed information about energy usage. Lu et al. [10]

propose the development of user-level power managers which could potentially incorporate feedback from applications into power management decisions. Our study of PowerPoint behavior provides further motivation for these projects by demonstrating that significant energy savings could be achieved with transparent power management.

8 Conclusion

We began this paper by asking several questions: Can one show significant energy reductions for the type of office applications that users most commonly execute on laptop computers? Is it possible to add energy-awareness to applications for which source code is unavailable? Is this approach valid for applications executing on closed-source operating systems (i.e. Windows)?

As the results of Section 4 show, the answer to all these questions is "yes". Puppeteer's component-based adaptation approach allows us to modify PowerPoint behavior without access to application or operating system source code. By distilling presentations and excluding multimedia data, we can reduce the energy needed to load presentations from a remote server by up to 49%. Distillation also leads to significant energy savings when editing and saving presentations. Finally, we showed how modifications such as disabling the Office Assistant and lowering autosave frequency can lead to further reductions in energy use.

Our study also revealed several instances where PowerPoint energy-efficiency could be improved. From these specific instances, we extracted general advice for developers who wish to make their applications and middleware more energy-efficient. We showed that transparent power management can lead to reduced energy usage by allowing applications and power management systems to incorporate knowledge of each other's activities. We then showed that applications can significantly reduce energy usage by minimizing low-priority activities and by replacing periodic models of application behavior with event-driven models.

This study has found several powerful tools that allow developers to decrease the energy usage of PowerPoint and similar applications. We believe the next logical step for this work is the development of system support for energy-aware adaptation and transparent power management in closed-source environments such as Windows. Such support will make it easier for software developers to achieve the energy reductions we have discussed. We then hope to expand system support beyond the single application we have studied to handle multiple, concurrently executing applications.

Acknowledgements. This research was supported by the National Science Foundation (NSF) under contract CCR-9901696, the Defense Advanced Research Projects Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918, IBM Corporation, Nokia Corporation and Intel Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, DARPA, USN, IBM, Nokia, Intel, or the U.S. government.

References

1. Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Opportunities for bandwidth adaptation in Microsoft Office documents. In *Procceedings of the 4th USENIX Windows Systems Symposium*, Seattle, WA, August 2000.
2. Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, March 2001.
3. Fred Douglis, P. Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pages 121–137, Ann Arbor, MI, April 1995.
4. Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *Proceedings of 1994 Winter USENIX Conference*, pages 293–307, San Francisco, CA, January 1994.
5. Carla Schlatter Ellis. The case for higher-level power management. In *The 7th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 162–167, Rio Rico, AZ, March 1999.
6. Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems and Principles*, pages 48–63, Kiawah Island, SC, December 1999.
7. A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–170, Cambridge, MA, October 1996.
8. P. James. *Official Netscape Navigator 3.0 Book*. Netscape Press, 1996.
9. Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the 1994 Winter USENIX Conference*, pages 279–291, San Francisco, CA, January 1994.
10. Yung-Hsiang Lu, Tajana Simunic, and Giovanni De Micheli. Software controlled power management. In *Proceedings of the 7th International Workshop on Hardware/Software Codesign*, pages 157–161, Rome, Italy, May 1999.
11. Thomas Martin and Daniel Siewiorek. A power metric for mobile systems. In *Proceedings of the 1996 International Symposium on Lower Power Electronics and Design*, pages 37–42, Monterey, CA, August 1996.
12. Rolf Neugebauer and Derek McAuley. Energy is just another resource: energy accounting and energy pricing in the Nemesis OS. In *Procceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Schloss Elmau, Germany, May 2001.
13. M. Satyanarayanan, Jason Flinn, and Kevin R. Walker. Visual proxy: exploiting OS customizations without application source code. *Operating Systems Review*, 33(3):14–8, July 1999.
14. A. Vahdat, A. R. Lebeck, and C. S. Ellis. Every joule is precious: A case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.
15. John Wilkes. Predictive power conservation. Technical Report HPL-CSP-92-5, Hewlett-Packard Laboratories, February 1992.

System Software for Audio and Visual Networked Home Appliances on Commodity Operating Systems

Tatsuo Nakajima

Department of Information and Computer Science
Waseda University
3-4-1 Okubo Shinjuku Tokyo 169-8555 Japan
tatsuo@mn.waseda.ac.jp

Abstract. This paper reports our ongoing project to build system software for audio and visual networked home appliances. In our system, we have implemented two middleware components for making it easy to build future networked home appliances. The first component is distributed home computing middleware that provides high level abstraction to control respective home appliances. The second component is a user interface middleware that enables us to control home appliances from a variety of interaction devices.
Most of our system have been implemented in Java, but several timing critical programs have been implemented in the C language, which runs on Linux. The combination of Linux and Java will be ubiquitous in future embedded systems. They enable us to port home computing programs developed on PC to target systems without modifying them, and Java's language supports enable us to build complex middleware very easily. Also, our user interface middleware enables us to adopt traditional user interface toolkits to develop home computing applications, but it allows us to use a variety of interaction devices to navigate graphical user interface provided by the applications.

1 Introduction

In the near future, home networks will connect a lot of networked home appliances such as digital TV, digital VCR, and DV cameras[15,20]. As the number of such appliances is increased, it becomes difficult to control the appliances individually. Also, it will not be realistic that each appliance has a different control device since we may not find an expected remote control device among a large number of the devices. Thus, it is desirable to use the nearest device such as PDAs and cellular phones to control the home appliances. On the other hand, these appliances will be connected to the Internet, and the services provided on the Internet will be integrated with the functions of home appliances to provide advanced services such as home shopping. The functionalities provided by the appliances will be very complicated, therefore high level abstraction is required to make it easy to build future advanced home applications that are constructed

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 273–294, 2001.

by the composition of a variety of services executed on the Internet or on the other home and personal information appliances.

We believe that the most important factor to develop future home appliances is the cost of products and the time to develop them. We need to consider a couple of issues to solve the problem. The first issue is to provide high level abstraction to make the development of home applications easy. Distributed middleware providing high level abstraction to control audio and visual home appliances for home appliances is a promising approach to solve the problem. However, it is not easy to develop such complicated middleware. Therefore, it is important to develop the middleware in a very portable way to reduce the cost to develop the middleware. Also, it is desirable to develop such complicated middleware by a sophisticated programming language that supports to build a large scaled software.

This paper reports our ongoing project to build system software for audio and visual networked home appliances. In our system, we have implemented two middleware components for making it easy to build future networked home appliances. The first component is distributed home computing middleware that provides high level abstraction to control respective home appliances. The second component is a user interface middleware that enables us to control home appliances from a variety of interaction devices.

Most of our system have been implemented in Java, but several timing critical programs have been implemented in the C language, which runs on Linux. The combination of Linux and Java will be ubiquitous in future embedded systems. They enable us to port home computing programs developed on PC to target systems without modifying them, and Java's language supports enable us to build complex middleware very easily. Also, our user interface middleware enables us to adopt traditional user interface toolkits such as Java AWT/Swing, GTK+, Qt to develop home computing applications, but it allows us to use a variety of interaction devices to navigate graphical user interface provided by the applications. Therefore, it will make the cost to develop home appliances dramatically cheap.

The remainder of this paper is structured as follows. In Section 2, we present the background of our work. In Section 3, we describe distributed middleware for audio and visual networked home appliances. Section 4 presents a user interface system to control home appliances. In Section 5, we show how our system works, and present the current status in Section 6. In Section 7, we describe several experiences with building our current prototype systems. Finally, in section 8, we conclude the paper.

2 Background and Motivation

Recently, embedded systems such as networked home appliances, internet appliances and personal area networks become very popular, and a lot of companies are developing very complex distributed embedded systems. In the future, the complexity will be increased exponentially, therefore middleware components providing high level abstraction will be very important.

2.1 Middleware for Home Computing Environments

In future computing environments, a variety of objects contain microprocessors and will be connected via various networks such as bluetooth, wireless LAN, and IEEE 1394. In [20], Mark Weiser describes some topics about "*Ubiquitous computing environments*", which promotes invisible embedded devices in every object near us. In his vision, embedded systems play a very important role to realize the goals, and one of the most popular environments to realize the goal is the *home computing environment*.

For building future advanced distributed home computing environments, high level standard interface is very important to reduce the cost of products and the time to develop them. This enables us to build a lot of reusable components that are used for building a variety of systems[12]. In the future, a lot of new standard middleware components will appear according to new requirements such as on-line shopping, smart spaces, and wearable computing. Therefore, it is important to use an infrastructure to build a variety of prototypes very quickly. The infrastructure needs to provide rich programming environments, a variety of device drivers, and various middleware components. We believe that an operating system that can be used universally, and distributed middleware for home appliances are key components to solve the above problems.

2.2 The Benefits of Our Approach

The aim of our project is to show the importance of middleware components to build advanced home computing systems. Our current project focuses on two issues. The first issue is to show the effectiveness of high level abstraction provided by our AV middleware component to build home applications. In future home computing environments, we will need to build large and complex middleware components. Therefore, our project is interested in how to build such complex software, and how to make the software portable. We believe that high level abstraction enables us to build advanced home computing applications quickly. The second issue is how to control home appliances in a very flexible way. The high level abstraction provided by our AV middleware component enables us to build flexible user interface, but it requires another middleware component to control user interface from various interaction devices.

There are several examples to show the effectiveness of our middleware components. We present four examples in this section. In the first example, home computing applications can be implemented on standard PCs, and the applications can be ported to target appliances without modifying them. The scenario can be realized by the adoption of COTS software components for building our middleware components, and shows that our approach makes the development cost very low. The second example is an application that changes graphical user interface according to a user's preference. The third example is an application that composes several home appliances, and the composition can be changed according to a user's preference. These two examples can be realized by the high level abstraction provided by our middleware components. In the fourth example, a home appliance can be controlled by various interaction devices. The

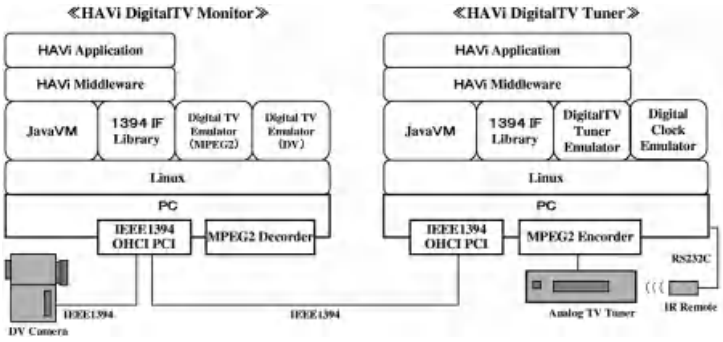


Fig. 1. Home Appliance System Architecture

devices are changed according to the situation of a user. For example, a PDA can be used to show graphical user interface that can be controlled on the PDA. If a user does not have a PDA, the nearest display can be used to display the graphical user interface that is controlled by a user's gesture. These flexible interaction scenarios can be realized by our middleware components although the graphical user interface is generated by using standard GUI toolkits.

3 Distributed Middleware for Networked Home Appliances

In this section, we describe distributed middleware for audio and visual networked home appliances, which we are building.

As shown in Figure 1, in our home computing system, we has implemented HAVi[7,8] that is a distributed middleware component for audio and visual home appliances. HAVi provides standard high level API to control various A/V appliances. The reason to choose HAVi as a core component in our distributed middleware for audio and visual home appliances is that HAVi is documented very well, and some real products will be available in the near future. Therefore, it is possible to use the products in our future experiments. The HAVi component is written in Java, and the Java virtual machine for executing HAVi runs on the Linux operating system. Also, continuous media streams such as DV or MPEG2 are processed in applications written in the C language running on Linux directly.

3.1 Software Designs for Complex Software

Our system provides a couple of high level abstraction for building new services in home computing because the abstraction makes it easy to implement a variety of home applications that need to compose several functionalities provided by different appliances. Also, the abstraction enables us to integrate home appliances with various services on the Internet.

However, it is not easy to implement the high level abstraction. Also, the abstraction should be used on many types of appliances. Thus, the implementation of the abstraction should be ported on various applications in an easy way. One approach to solve the problem is to adopt a software platform that is widely available. The solution allows us to build complex software on various appliances in an easy way if the platform can be used on a wide range of appliances.

However, there is no software platform that is suitable for a various types of software because we need to take into account several requirements to design complex software, and it is necessary to consider the tradeoff among the requirements since it is impossible to satisfy all requirements to develop complex software. Therefore, it is desirable to adopt several software platforms simultaneously according to the characteristics of respective software components. The approach allows us to select a suitable software platform for implementing each program.

In our approach, we configure several software platforms in a layered structure. We call the structuring *multi-layered software platforms*. Software in our system is divided into several components. One component requires high productivity, but it does not require severe timing constraints. On the other hand, it is more important to satisfy timing constraints for another component than to increase productivity. In our current implementation, the former component is implemented on a Java virtual machine, and the latter component is built on Linux directly. Also, the Java virtual machine runs on Linux in a hierarchical fashion. If our system needs to be ported on other operating systems, the component that runs on Linux directly should be ported, but a component implemented in Java needs not to be modified.

3.2 Structure of Our Home Computing System

Our system consists of five components. The first component is the Linux kernel which provides basic functionalities such as thread control, memory management, network systems, and file systems, and device drivers. The kernel also contains IEEE 1394 device drivers and MPEG2 decoder drivers which are required to implement HAVi. The second component is a continuous media management component running on Linux, and written in the C language. The component sends or receives media streams to/from IEEE 1394 networks and MPEG2 encoder/decoder devices, and it sends media streams to a speaker and a window system to simulate various audio and visual home appliances. The component also encodes, decodes or analyzes media elements in a timely manner.

The third component is a Java virtual machine to execute a middleware component such as HAVi. Java provides automatic memory management, thread management, and a lot of class libraries to build an sophisticated middleware components and home computing applications in an easy way. Also, Java Swing and AWT enable us to build sophisticated graphical user interface to write applications for home appliances. Therefore, a lot of standard middleware components for home computing assume to use Java due to the rich class libraries. The fourth component is the HAVi middleware component described in the next section. In our system, the component is written in Java. The last component is home com-

puting application programs that invoke HAVi API to control home appliances. The component adds extra functionalities to actual home appliances. One of the most important roles of the application component is to provide graphical user interface to control home appliances.

3.3 Home Appliances on Linux

This section describes an overview of our home computing system. First, we give a brief description about HAVi which is a distributed middleware for networked home appliances. Next, we show how our system emulates a variety of networked home appliances.

Middleware for Networked Home Appliances: A variety of audio/visual home appliances such as TV and VCR are currently connected by analog cables, and these appliances are usually controlled individually. However, there are several proposals for solving the problem recently. Future audio/visual home appliances will be connected via the IEEE 1394 interface, and the network will be able to deliver multiple audio and video streams that may have different coding schemes. HAVi(Home Audio/Video Interoperability) Architecture proposed by Sony, Matsushita, Philips, Thomson, Hitachi, Toshiba, Sharp, and Grundig is a standard distributed middleware for audio and visual home appliances, and provides an ability to control these appliances in an integrated fashion by offering standard programming interface. Therefore, an application can control a variety of appliances that are compliant to HAVi without taking into account the differences among appliances even if they are developed by different vendors.

The HAVi architecture also provides a mechanism to dynamically load a Java bytecode from a target device. The Java bytecode knows how to control a target appliance since the code implements a protocol to communicate between a client device and a target appliance. Therefore, the client device needs not to know how to control the target appliance, and the target appliance can use the most suitable protocol for controlling it. Also, the bytecode may be retrieved from a server on the Internet, therefore it is easy to update software to control target appliances.

The architecture is very flexible, and has the following three advantages.

- The standardized programming interface that provides high level abstraction hides the details of the appliances. Therefore, it is possible to build applications that are vendor independent. Also, the application can adopt future technologies without modifying target appliances. The feature enables us to build a new appliance by composing existing appliances.
- The software to control target appliances can be replaced by retrieving from the Internet. The feature removes the limitations of traditional home appliances since the functionalities may be updated by modifying software.
- The Java bytecode loaded from a target appliance enables us to use the most appropriate protocol. The feature allows us to choose the state of the art technologies without modifying existing applications when a company releases the appliance.

Especially, the first and the third advantages are suitable for developing a prototype very rapidly since the advantages allow us to adopt future technologies very easily without modifying existing appliances and home application programs.

The HAVi component consists of nine modules defined in the HAVi specification. In HAVi, the module is called *software element. DCM(Device Control Module)* and *FCM(Function Control Module)* are the most important software elements in HAVi. Each appliance has a DCM for controlling the appliance. The DCM is usually downloaded to a HAVi device, and it invokes methods defined in DCM when the appliance needs to be controlled. A DCM abstracts an appliance, and contains several FCMs. Each functionality in an appliance is abstracted as a FCM. Currently, tuner, VCR, clock, camera, AV Disk, amplifier, display, modem, and Web Proxy are defined as FCM. Since these FCMs have a standard API to control them, an application can control appliances without knowing the detailed protocol to access the appliances.

Home Appliance Emulation on Linux: The continuous media management component processes continuous media streams such as MPEG2 and DV streams. To process media elements in a timely manner, the components are implemented in the C language and are running on the Linux kernel directly.

The continuous media management component is controlled by a Java bytecode provided by the appliance implementing the component. As described in the previous section, the bytecode is downloaded to a HAVi device such as set-top boxes, and installed as a DCM software element. The bytecode implements a protocol to transmit a control message to a target appliance, and the message is processed by the continuous media management component to control media streams.

The continuous media management component contains several threads. Let us assume a continuous media management component that processes an audio stream and a video stream. The audio stream is delivered to a speaker, and the video stream is displayed on a window system.

The component contains five threads. The first thread initializes the component, sends a Java bytecode representing a DCM to a HAVi device, and waits for receiving commands from the HAVi device. When an application on the HAVi device invokes a method of the downloaded DCM, a control message is delivered to the component. Although HAVi does not define the protocol between a HAVi device and an appliance, the 1394 AV/C command may make it easy to implement DCMs.

The second thread receives packets from a IEEE 1394 network, and collect them as media elements. The media elements are classified into video frames and audio samples, and they are passed to the third and fourth threads respectively. In our system, threads are communicated through time-driven buffers to carry media elements. The time-driven buffer stores media elements in the FIFO order, but obsolete media in the buffer elements are dropped automatically. The third thread retrieves video frames from a time-driven buffer, and sends them to a window system. Also, the fourth thread retrieves audio samples from a time-driven buffer, and sends them to an audio device. The last thread monitors

the performance of a system, and controls a QOS value of each media stream according to the priority of media streams.

In our system, each media element has a header containing information to identify the media format, and it also contains timestamps to process the media element correctly. Currently, we adopt a RTP header as a header in our system to be assigned to each media element.

To process the timestamp assigned to each media element correctly, each machine's clock should be synchronized. However, there is a clock skew among different clocks, thus we require a mechanism to synchronize them. Also, to control an overload condition, each media stream's QOS should be changed. The detailed description to process continuous media in our system can be found in [11].

3.4 Applications Components

The purpose of an application in our system is to allow us to control home appliances with a variety of devices such as PDAs and cellular phones. In the future, we will have a variety of home appliances, but each appliance traditionally offers a different remote controller to control it. However, the approach has the following problems.

- To control an appliance, we usually need a specific remote controller.
- Everyone needs to use the same user interface to control an appliance.
- When a user controls an appliance, a system does not understand the situation of the user.
- When there are several appliances, a user needs to control them independently.

For solving the problem, an application should have three mechanisms. The first mechanism is to support various types of interaction devices to control an appliance. For example, we like to use an interaction device carried by us to control the appliance. The problem can be solved by the user interface middleware described in the next section. The second mechanism is to take into account our situations. For example, if a system knows the location of a user, the system can control an appliance in a more appropriate way. Also, if a system knows the preference of a user, there may be a better way to control the appliance. The third mechanism is to compose multiple functions in different appliances as one appliance. The mechanism allows us to define a new appliance very easily. The second and third mechanisms should be provided as appropriate high level abstraction to build application components easily. In the future, we like to provide the more high level abstraction to provide the second and third mechanisms on HAVi API.

3.5 A Sample Scenario

In this section, we present a scenario to build an emulated digital TV using our system. The scenario has actually been implemented using our prototype and

give us a lot of experiences with building distributed home computing environments.

In a program emulating a digital TV appliance, a Linux process receives a MPEG2 stream from an IEEE 1394 device driver, and displays the MPEG2 stream on the screen. The sender program that simulates a digital TV tuner receives the MPEG2 stream from a MPEG encoder, and transmits it to the IEEE 1394 network. Currently, the MPEG2 encoder is connected to an analog TV to simulate a digital TV tuner, and the analog TV is controlled by a remote controller connected to a computer with a serial line. A command received from the serial line is transmitted to the analog TV through infrared.

The receiver process contains three threads. The first thread initializes and controls the program. When the thread is started, it sends a DCM containing a Java bytecode to a HAVi device. In our system, the HAVi device is also a Linux based PC system. When the HAVi device receives the bytecode, a HAVi application on the HAVi device shows graphical user interface to control the emulated digital TV appliance. If a user enters a command through the graphical user interface, the HAVi application invokes a method of the DCM representing a tuner. Then, the HAVi device executes the downloaded bytecode to send a command to the emulated digital TV appliance, and the first thread will receive the command.

After the second thread in the emulated digital TV appliance receives packets containing MPEG2 video stream from an IEEE 1394 network, it constructs a video frame, and inserts the frame in a time driven buffer. The third thread retrieves the frame from the buffer, then the video frame is delivered to a MPEG2 decoder according to the timestamp recorded in the video frame. Finally, the decoder delivers the frame to a NTSC line connected to the analog TV display.

4 User Interface System for Networked Home Appliances

Our user interface system is based on the stateless thin-client system such as Citrix Metaframe[3], Microsoft Terminal Server[10], the AT&T VNC system[18], and Sun Microsystems Sun Ray[16].

In the future, a lot of new interaction devices will appear, but it is not realistic to rewrite all existing application programs for the new interaction devices. *Universal interaction* realized by our user interface middleware enables us to use the new interaction devices without modifying the applications programs that adopt traditional GUI toolkits. The benefits of universal interaction enable us to behave uniformly either in home, in offices, or in public spaces when controlling a variety of appliances.

4.1 Design Issues

There are several ways to realize the goals described in the previous section. The most important issue is how to support a variety of interaction devices. Traditional user interface systems assume a few types of interaction devices. For example, mouses, track points, and touch screens are used as input devices.

There are three alternative approaches to build a flexible user interface system as shown below.

- Builds a new user interface system from scratch.
- Builds a new device independent layer that invokes respective traditional user interface systems for a variety of interaction devices.
- Captures input and output events of traditional user interface systems, and transforms the events according to interaction devices.

We have chosen the third approach since our home computing system implementing HAVi requires to use Java AWT, and we like to use various home computing applications developed on HAVi in the near future without modifying them. Recent interests in the use of embedded Linux to build home appliances make our approach more practical since Linux usually adopts the X window system as its basic user interface system, and our system enables a variety of applications on embedded Linux to be controlled by various interaction devices. The approach makes the development of a variety of embedded applications very easy since the applications can use traditional and familiar user interface toolkits.

4.2 Universal Interaction

In our approach, we call the protocol that can be universally used for the communication between input/output interaction devices and appliances *universal interaction*. *Universal interaction* enables us to control a variety of home appliances in a uniform way. This means that our behavior is not restricted by where we are or which appliance we like to control. Therefore, our approach provides very natural interaction with home appliances.

The output events produced by appliances are converted to *universal output interaction events*, and the events are translated for respective output interaction devices. Also, input events generated in input interaction devices are converted to *universal input interaction events*, and the events are processed by applications executed in appliances.

A universal interaction proxy that is called the UniInt proxy as described in the next section plays a role to convert between the universal interaction protocol and input/output events of respective interaction devices in a generic way. The proxy allows us to use any input/output interaction devices to control appliances if the events of the devices are converted to the universal interaction protocol. This approach offers the following three very attractive characteristics.

The first characteristic is that input interaction devices and output interaction devices are chosen independently according to a user's situation and preference. For example, users can select their PDAs for their input/output interaction. Also, the users may choose their cellular phones as their input interaction devices, and television displays as their output interaction devices. For example, users may control appliances by their gesture by navigating augmented real world generated by wearable devices.

The second characteristic is that our approach enables us to choose suitable input/output interaction devices according to a user's preference. Also, these

interaction devices are dynamically changed according to the user's current situation. For example, a user who controls an appliance by his/her cellular phone as an input interaction device will change the interaction device to a voice input system because his both hands are busy for other work currently.

The third characteristic is that any applications executed in appliances can use the any user interface systems if the user interface systems speak the universal interaction protocol. In our approach, we currently adopt keyboard/mouse events as universal input events and bitmap images as universal output events. The approach enables us to use traditional graphical user interface toolkits such as Java AWT, GTK+, and Qt for interfacing with any interaction devices. In fact, a lot of standards for consumer electronics like to recently adopt Java AWT for their GUI standards. Thus, our approach will allow us to control various future consumer electronics from various interaction devices without modifying their application programs. The characteristic is very desirable because it is very difficult to change existing GUI standards.

4.3 System Architecture

Our system uses the stateless thin-client system to transfer bitmap images to draw graphical user interface, and to process mouse/keyboard events for inputs. The usual thin-client system consists of a viewer and a server. The server is executed on a machine where an application is running. The application implements graphical user interface by using a traditional user interface system such as the X window system. The bitmap images generated by the user interface system are transmitted to a viewer that are usually executed on another machine. On the other hand, mouse and keyboard events captured by the viewer are forwarded to the server. The protocol between the viewer and the server are specified as a standard protocol for showing a desktop on various client systems[1]. In the paper, we call the protocol the RFB protocol. The system is usually used to move a user's desktop according to the location of a user[15], or shows multiple desktops on the same display, for instance, both MS-Windows and the X Window system.

In our system, we replace the viewer to a UniInt(Universal Interaction) proxy that forwards bitmap images received from a UniInt server to an output device. In our system, a server of any thin-client systems can be used as the UniInt server. Also, UniInt proxy forwards input events received from an input interaction device to the UniInt server.

Our system consists of the following four components as shown in Figure 2. In the following paragraphs, we explain these components in details.

- Home Appliance Application
- UniInt Server
- UniInt Proxy
- Input/Output Devices

Home appliance applications generate graphical user interface for currently available home appliances to control them. For example, if TV is currently available, the application generates user interface for the TV. On the other hand, the

[1] The AT&T VNC system calls the protocol the RFB(Remote Frame Buffer) protocol.

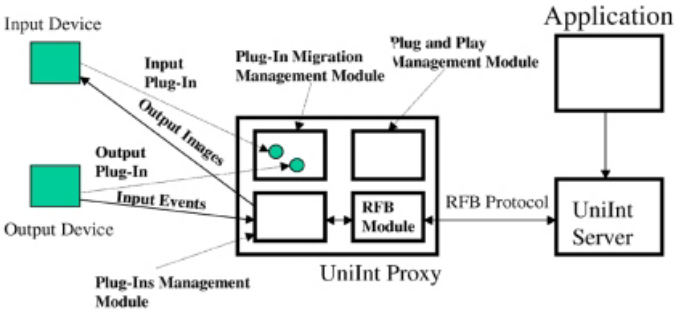


Fig. 2. System Architecture

application generates the composed GUI for TV and VCR if both TV and VCR are currently available.

The UniInt server transmits bitmap images generated by a window system using the RFB protocol to a UniInt proxy. Also, it forwards mouse and keyboard events received from a UniInt proxy to the window system. In our current implementation, we need not to modify existing servers of thin-client systems, and any applications running on window systems supporting a UniInt server can be controlled in our system without modifying them.

The UniInt proxy is the most important component in our system. The UniInt proxy converts bitmap images received from a UniInt server according to the characteristics of output devices. Also, the UniInt proxy converts events received from input devices to mouse or keyboard events that are compliant to the RFB protocol. The UniInt proxy chooses a currently appropriate input and output interaction devices for controlling appliances. Then, the selected input device transmits an input plug-in module, and the selected output device transmits an output plug-in module to the UniInt proxy. The input plug-in module contains a code to translate events received from the input device to mouse or keyboard events. The output plug-in module contains a code to convert bitmap images received from a UniInt server to images that can be displayed on the screen of the target output device.

The last component is input and output interaction devices. An input device supports the interaction with a user. The role of an input device is to deliver commands issued by a user to control home appliances. An output device has a display device to show graphical user interface to control appliances.

In our approach, the UniInt proxy plays a role to deal with the heterogeneity of interaction devices. Also, it can switch interaction devices according to a user's situation or preference. This makes it possible to personalize the interaction between a user and appliances.

4.4 UniInt Proxy

The current version of UniInt proxy is written in Java, and the implementation contains four modules as shown in Figure 2. The first module is the RFB protocol

module that executes the RFB protocol to communicate with a UniInt server. The replacement of the module enables us to use our system with different thin-client systems. The module can use the same module implemented in a viewer of a thin-client system. The second module is the plug-in management module that receives input and output plug-in modules from interaction devices, and dynamically links the modules in the UniInt proxy. The third module is the plug-and-play management module that detects currently available input and output interaction devices. The last module is the plug-in migration management module that manages the migration of input and output plug-in modules between interaction devices and a UniInt proxy.

Management for Available Interaction Devices: The plug and play management module detects the currently available input and output devices near a UniInt proxy. In our system, a unique ID is assigned for each type of input/output devices. The UniInt proxy broadcasts beacon messages periodically. In the current prototype, interaction devices are connected via IEEE802.11b, Ethernet or infrared networks. When each interaction device receives a beacon message, it replies an acknowledgement message. The acknowledgement message contains the unique ID to identify the device type. If the UniInt proxy receives several acknowledgment messages from multiple interaction devices, it chooses one device according to the preference determined by the UniInt proxy. Also, when a newly detected device replies an acknowledgement message, the device may be chosen as a currently used interaction device, if the device is more preferable than the currently used device.

When a UniInt proxy chooses a new device after the detection of the device, it sends an acknowledgement message before using the device. Then, the UniInt proxy sends a terminate message to the device that is used until now. Finally, the UniInt proxy waits for receiving a plug-in module from the new device. The preference to select input and output devices is registered in a UniInt proxy for each user. If the system cannot detect who likes to use an appliance, a default preference is chosen. Also, each plug-in module supports an event to switch the currently used input and output devices. For example, a user can send a command to change a currently used output device to a UniInt proxy. The UniInt proxy switches the current output device to the next one until the user selects his favorite output device.

Migration Management of Plug-In Modules: When receiving an acknowledgement message from the UniInt proxy, the input/output devices send plug-in modules to the UniInt proxy. In our system, we are using the MobileSpaces mobile agent system[14] to transmit plug-in modules between input/output devices and a UniInt proxy. After the plug-in modules are downloaded in the Java virtual machine executed in a UniInt proxy, the plug-in migration module sends a migration complete message to the input/output devices. The MobileSpaces system supports hierarchical agents, and the agents can be communicated when they reside at the same hierarchical level. Therefore, we also implement a UniInt proxy as a mobile agent, but the agent does not be migrated to other hosts.

However, the feature may be used to move a UniInt proxy to a near computer from input and output devices.

Lifecycle Management of Plug-In Modules: The plug-in management module contains an input and an output plug-in module. The input plug-in module receives events from an input device that is currently selected. The event is converted to a mouse or a keyboard event, and the event is transferred by the RFB protocol to a UniInt server. For example, if a user touches a button drawing a right arrow, the event is transmitted to the input plug-in module delivered from a PDA device. The event is translated to a mouse movement event to the right, and the event is finally forwarded to a window system.

Also, after bitmap images are received by the RFB protocol module in a UniInt proxy from a UniInt server, an output plug-in module processes the images before transmitting to an output device. For example, a color image received from a UniInt server is converted to a black and white image. Also, the size of the image is reduced to show on a PDA's screen.

4.5 Context-Awareness in Home Computing Applications

The role of home computing applications is to allow us to control home appliances easily. The interaction between a user and home appliances will become more complex since the functionalities of appliances will be richer and richer. Also, the number of appliances will be increased in the future. Therefore, future home applications should support context aware interaction with a variety of appliances.

There are two types of context awareness that should be taken into account. The first one is to personalize the interaction. The interaction is also customized according to a user's situation. The second type is to deal with currently available multiple appliances as one appliance.

It is usually difficult to personalize the interaction with an appliance since a system does not know who controls the appliance. In our system, we assume that each user has his own control device such as a PDA and a cellular phone. If these devices transmit the identification of a user, the system knows who likes to control the appliance. However, in our system, there is no direct way to deliver such information to a home computing application from interaction devices since we assume that the application adopts traditional user interface systems that do not support the identification of a user. Therefore, in our current implementation, each user has a different UniInt server that executes personalized applications for each user. The application provides customized user interface according to each user's preference. The UniInt proxy chooses an appropriate UniInt server according to the identification of a user acquired from an input device.

Our system needs to know which appliances are currently available according to the current situation. In the current system, we assume that an application knows which appliances can be available. For example, if the application supports three home appliances, the application needs to provide seven graphical user interfaces with the combination of the three appliances. The user interfaces are selected according to the currently available appliances. In our system, we

assume that each home appliance is connected via IEEE 1394 networks. Since IEEE 1394 networks support a mechanism to tell which appliances are currently connected and whose power switches are turned on, it is easy that the application easily knows the currently available appliances, and selects the most suitable user interface.

4.6 Input and Output Interaction Devices

In our system, a variety of input and output devices are available, and the input devices and output devices may be separated or combined. For example, a graphical user interface can be displayed on the screen of a PDA or a large display device on TV. The user interface displayed on TV can be navigated by a PDA device. Therefore, a user can choose a variety of interaction styles according to his preference. Also, these devices can be changed according to the current situation. For example, when the currently used interaction devices are unavailable, another interaction device may be selected to control appliances.

We assume that each device transmits a plug-in module to a UniInt proxy as described in Section 4.4. However, some input devices such as a microphone may not be programmable, and it is difficult to support the communication to a UniInt proxy. In this case, we connect the device to a personal computer, and the computer communicates with a UniInt proxy to deliver a plug-in module. However, it is difficult to know when a program on the personal computer returns an acknowledgement message when a beacon message is received from a UniInt proxy since the computer does not understand whether a microphone is currently available or not. Therefore, in the current prototype, we assume that the device is always connected and available.

5 How Does Our System Work?

In the section, we describe how our system works using our middleware components. When an application recognizes that currently available appliances are a television and a video recorder, it shows a graphical user interface for controlling them. Since the current user does not interested in the TV program reservation, the application draws a user interface containing power control, TV channel selection, and VCR function to its UniInt server. As described in Section 4.5, a UniInt proxy selects a UniInt server executing an application drawing a customized user interface for the user who likes to control these appliances, and the selected UniInt server transmits bitmap images containing the interface to the UniInt proxy.

Let us assume that the UniInt proxy detects that the user has a PDA device. The PDA device delivers input and output plug-in modules to the UniInt proxy. The UniInt proxy transcodes bitmap images transmitted from the UniInt server using the output plug-in module before transmitting the images to the PDA device. In this case, 8 bits color images whose image size is 640x480 are reduced to monochrome images whose size is 180x120. Also, input events on the touch screen of the PDA device are converted to mouse and keyboard events

by the input plug-in module. However, we assume that the user likes to see the graphical user interface on a bigger display now. The user transmits a command to the UniInt proxy by tapping on the screen. When the UniInt proxy detects it, the bitmap images containing the graphical user interface are forwarded to the display system, and the images are converted by the output plug-in module provided by the bigger display before transmitting them. Also, the user interface will be displayed again on the screen of the PDA device by tapping the PDA's screen.

6 Current Status

Currently, we have four PCs and one DV camera connected by an IEEE 1394 network. The first PC executes MS-Window98[2], and has an MPEG2 encoder that is connected to a analog TV tuner via a NTSC cable. The PC also executes a program that controls a remote controller to access the TV tuner. The program translates a command from the PC to an infrared command that enables us to control the commercial analog TV tuner. The DCM for our digital TV emulator transmits commands to the program, then the commands are finally forwarded to the analog TV.

The second PC executes Linux, and has an MPEG2 decoder. The PC executes an MPEG2 receiver process. The MPEG2 decoder has a NTSC interface which is connected to an analog TV display. The continuous media management component on the PC has the DCM for a MPEG display.

The third PC contains the DCM for a DV camera. The DCM contains a code to control a DV camera using the AV/C command. A program to detect a DV camera running on the PC transmits the DCM to a HAVi device when a DV camera is connected to the IEEE 1394 network. The PC also emulates a DV monitor that receives DV frames from an IEEE 1394 broadcast synchronous channel, and the continuous media management component for the DV monitor also has the DCM for the DV monitor. The fourth PC emulates a HAVi device. When the PC receives DCMs from the DV monitor or the digital TV emulation program, an application running on the PC shows graphical user interface. By navigating the interface, the methods provided by the DCMs are invoked by the application, and commands to control continuous media management components emulating home appliances will be delivered.

We found that the approach that building middleware components on Linux and Java is very attractive to develop embedded software because the software can be executed on embedded devices without modifying it. This decreases the development cost of embedded devices dramatically. Also, even if we cannot adopt Linux on target embedded devices, the porting of the software is not difficult. For example, we have ported our implementation of HAVi on an embedded device that has a Pico Java II chip. In the experiment, we are able to port HAVi without modifying it, therefore the porting of the software was dramatically easy.

[2] Currently, we are using MS-Window98 to emulate a digital TV tuner because we could not find a MPEG2 encoder that can be used on Linux.

Our user interface system has adopted the AT&T VNC system as a thin-client system. Our application shows a graphical user interface according to currently available appliances as described in the previous section. Also, the cursor on a screen that displays a graphical user interface can be moved from a PalmPilot as shown in Figure 3. However, if the device is turned off, the cursor is controlled by the keyboard and the mouse of a personal computer. It is also possible to show a graphical user interface on the PDA device according to a user's preference. Also, the current system has integrated cellular phones to control home appliances. NTT Docomo's i-mode phones have Web browsers, and this makes it possible to move a cursor by clicking special links displayed on the cellular phones.

In our home computing system, we have adopted Linux/RT[9] that extends a standard Linux by adding a resource reservation capability. The Linux also provides an IEEE 1394 device driver and an MPEG2 decoder. Also, IBM JDK1.1.8 for the Java virtual machine is used to execute the HAVi middleware component.

Fig. 3. Controlling HAVi with Various Interaction Devices

7 Experiences

This section describes several experiences with building our prototype systems.

Home Appliance Applications: One of the most important issues to build future home appliances is their usability. Especially, the usage of appliances should be customized according to each user's preferences and situations. For example, current home appliances such as TV and VCR provide rich functionalities such as TV program reservation. Since the remote controller has a lot of buttons, usual users confuse how to use these appliances. On the other hand, our system easily changes user interface by replacing applications that invoke HAVi API. This means that it is possible to change user interface according to the preference of a user. Also, there is a possibility to compose multiple functions in several existing appliances, and it enables us to define a new appliance in an ad-hoc way. Therefore, our system is very suitable to do a variety of experiments for developing future advanced home appliances.

However, in HAVi, Registry returns a list of software elements, when an application retrieves an necessary function such as a tuner function and a camera function. The current HAVi does not provide enough information to select the functions. Therefore, a variety of attributes should be stored in Registry to select the most suitable function to develop advanced context-aware home applications.

Also, the customization of user interface requires to represent a variety of aspects of our daily life such as location information of users and various objects or a user's emotion. To make an application to be adapted according to context information portable, the representation of such information should be standardized. However, it is not easy to represent the context information and the preference of a user. To represent such information, we need to model the entire world. This makes the specification of middleware to become too big to use for home appliances, thus we need more efforts in this area.

Distribution Transparency: The issue discussed in this section is how to manage distribution. Home appliances connected by networks store a variety of information that should be consistent whenever some failures occur or the configuration of networks is changed. Also, to build stable distributed systems, it is important to define rigorous semantics for all operations in systems.

The following two issues are completely ignored by the current researches on home computing. The first issue is very important because home computing environments assume that the configuration changed at any moment. There is a lot of work to maintain consistency in distributed systems such as transactions[5] and process groups[2], and these concepts are useful to build distributed home appliances. For example, a digital TV appliance may invoke a transaction for on-line shopping. The database to monitor the behavior of users should keep the consistency even if an application is turned off while modifying the database. We believe that transactions are also a useful concept in home computing environments, and it is important to develop an appropriate transaction model for home computing environments. Also, fault tolerance is an important topic in

home computing environments since it is not desirable to fail to record a broadcast program due to the crash of an appliance. In this case, an application should be implemented by using a process group concept to store critical information.

The second issue is important when multiple functions in different appliances are composed. These functions provide standard interface to access them. For example, each tuner function provides the same interface even if the vendors are different. However, since the semantics of the interface is not define rigorously, it is not ensured to behave in the same way. Because the distribution of these functions is hidden from an application in a transparent way, it is not easy to build stable applications when a failure occurs[6,19]. We should take into account the difficulties of distributed systems when designing distributed middleware components.

High Level Abstraction: We have implemented several home computing applications on our implementation of HAVi. One of these applications is a remote control application that is integrated with a Web server. It enables the application to convert commands from the embedded Web server to commands to access HAVi API. This means that it is easy to build an application to control home appliances from a variety of control devices embedding Web browsers. We believe that the key point to realize fantastic home computing environments is to provide high level API that enables us to build a variety of home applications.

The high level API provided by HAVi allows us to build advanced home applications that are personalized for each person. For example, it is easy to build a program customizing graphical user interface according to the preference of a user. The program identifies a user, and changes the layout of of graphical user interface. However, it is not easy to identify the current user who likes to use an appliance. Also, it is not easy to represent context information to customize the behavior of applications. For example, it is not simple to generate graphical user interface according to the currently available appliances.

The high level API enables us to compose several appliances, but it is not realistic to create an application for respective compositions. Therefore, it is important to build an application to compose appliances from a composition specification that is a high level language specific for describing the composition of appliances. The specification should be declarative, and it should be possible to compose several specifications to reuse existing specifications. Our experience shows that the current high level API does not have enough power to represent the composition. Especially, the naming of functions in appliances and a variety of services is a key issue to develop a useful composition specification.

COTS Platforms: We have implemented HAVi using Java, and continuous media management components on Linux. Our platform increases the portability of our middleware by adopting COTS components. A variety of open software make us to develop prototype applications very quickly. For example, free DV decoder software enables us to create our DV monitor in a very short time, and a free Web server enables us to create a remote control application easily. We believe that the adoption of open platform allows us to build a variety of prototype

applications very cheaply. Also, Linux allows us to use the same operating system both for development machines and target devices. Especially, recent Linux kernel provides a variety of functions that are suitable for embedded systems such as ROM file systems and flash file systems. Therefore, the development cost of home appliances will be dramatically decreased by the adoption of Linux.

Object-orientation, automatic memory management, sophisticated synchronization support and rich class libraries enable us to develop complex middleware such as HAVi very quickly. Especially, the development of traditional embedded applications is very difficult by synchronization misuse and memory leak. However, Java's synchronization mechanism is well structured, therefore concurrent programs become robust quickly. Also, Java's automatic memory management prevents us from memory leaks that are especially serious in multi-threaded programming. The fact is very important for developing future home appliances since the development time should become shorter although the complexity will be increased exponentially.

Limitation of Our User Interface System: In our system, a bitmap image that contains a graphical user interface is transferred from a UniInt server to a UniInt proxy. Since the image does not contain semantic information about its content, the UniInt proxy does not understand the content. For example, it is difficult to extract the layout of each GUI component from the image. Therefore, it is not easy to change the layout according to the characteristics of output devices or a user's preference. Also, our system can deal with only mouse and keyboard events. Thus, the navigation of a graphical user interface can be done by emulating the movement of a cursor or pressing a keyboard and mouse buttons. If the limitation makes the usability of a system bad, other approaches should be chosen. However, navigating a graphical user interface from a PDA and a cellular phone provides very flexible interaction with home appliances. Our experiences show that home appliances usually allow us to use a large display and show graphical user interfaces on the display to control the appliances. Thus, we believe that our system has enough power to make future middleware components for home appliances flexible.

Better Control for Home Appliances: Our system can control any applications executed on standard window systems such as the X window system. In our home computing system, traditional applications coexist with home computing applications, and these applications can also be controlled in an integrated way by our system. For example, we can navigate an MP3 player or a Netscape browser running with home computing applications via our system. However, the overlapping window layout is painful to be navigated by our user interface system. We consider that the tiled window strategy is more suitable for controlling home appliances. Also, our experience shows that we can control both home appliances and traditional applications such as presentation software and web browsers by using our system in a comfortable way if our system supports the movement of a mouse cursor at a variety of speeds.

Porting Issues in Embedded Systems: In embedded systems, underlying platforms provide a variety of advanced features. The features are different on respective platforms, and it is impossible to standardize these features since the standardization may hide some details of respective platforms. Also, it is important to support Quality of Service(QOS) parameters to accommodate a variety of resource constraints. However, a general QOS specification is not suitable for embedded systems since the specification will be too complex and big. Therefore, it is desirable to provide a specialized QOS specification for each platform. The QOS specification should take into account the predictability of underlying software such as scheduling behavior and worst case execution time.

However, the approach makes the portability of software bad since it is necessary to rewrite programs when they are ported on other platforms. We need to develop a new methodology to develop a portable program that can be adapted to respective platforms. We have a plan to enhance our middleware components with our new methodology[12]. The methodology is based on famous separation of concerns principle. In the methodology, we are working on transparently incorporating advanced features provided as non standardized API and a domain specific QOS specification on existing programs. Also, the methodology includes a method to predict the behavior of programs, which is important to predict the behavior of a program when it is enhanced to access advanced underlying features.

8 Conclusion

This paper has described system software for audio and visual networked home appliances. Our system consists of two components. The first component is distributed middleware for audio and visual networked home appliances. The component provides high level abstraction to control home appliances. The second component is a user interface system to allow us to use a variety of interaction devices to control graphical user interface provided by home applications.

Our system increases the portability of system software for home appliances dramatically since we have adopted commodity software such as Java and Linux. Also, our user interface system allows home computing applications to adopt standard user interface middleware, but it enables us to use a variety of interaction devices instead of a mouse and a keyboard. In the future, we like to prove that our approach to adopt commodity software as much as possible is useful to build other complex future embedded systems such as personal appliances like cellular phones and PDAs.

Ubiquitous computing environments that are located in any spaces will be integrated by connecting them via the Internet in the near future[13]. The environments are extremely heterogeneous, but requires applications to be extremely portable. Also, in the environments, we need to connect different distributed middleware components such as HAVi, Jini and UPnP. We consider that network systems that are customized according to the characteristics of respective applications are very important in the near future[1].

References

1. H.Aizu, I.Satoh, D.Ueno, T.Nakajima, "A Virtual Overlay Network for Integrating Home Appliances", In the Proceedings of the 2nd International Symposium on Applications and the Internet, 2002.
2. K.Birman, "The Process Group Approach to Reliable Distributed Computing", Communications of the ACM, Vol.36, No.12, 1993.
3. Citrix Systems, "Citrix Metaframe 1.8 Background", Citrix White Paper, 1998.
4. T. Hodes, and R. H. Katz, "A Document-based Framework for Internet Application Control", In Proceedings of the Second USENIX Symposium on Internet Technologies and Systems, 1999.
5. J. Gray and A.Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, 1993.
6. R. Guerraoui, "What Object Oriented Distributed Programming Does not Have to be, and What it may be", Informatik, 2 1999.
7. HAVi Consortium, "HAVi Specification: Specification of the Home Audio/Video Interoperability(HAVi) Architecture, http://www.havi.org/
8. R.Lea, S.Gibbs, A.Dara-Abrams, E. Eytchson, "Networking Home Entertainment Devices with HAVi", IEEE Computer, Vol.33, No.9, 2000.
9. Timesys, "Linux/RT", http://www.timesys.com.
10. Microsoft Corporation, "Microsoft WIndows NT Server 4.0: Technical Server Edition, An Architecture Overview", Technical White Paper 1998.
11. T. Nakajima, et. al., "A Framework for Building Audio and Visual Home Appliances on Commodity Software, In the Proceedings of IASTED International Conference on Internet and Multimedia Systems and Applications(IMSA2001), 2001.
12. T. Nakajima, "Towards Universal Software Substrate for Distributed Embedded Systems", In Proceedings of the International Workshop on Object-Oriented Real-Time Dependable Systems, 2000.
13. T. Nakajima, et. al., "Technical Challenges for Building Internet-Scale Ubiquitous Computing", Technical Report, Waseda University, 2001.
14. I. Sato, "MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System", In Proceedings of IEEE International Conference on Distributed Computing Systems, 2000.
15. Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, Paul Webster, "The Anatomy of a Context-Aware Application", In Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1999.
16. Sun Microsystems, "Sun Ray 1 Enterprise Appliance", http://www.sun.com/products/sunray1/.
17. User Interface Markup Language, http://www.uiml.org/
18. T.Richardson, et al., "Virtual Network Computing", IEEE Internet Computing, Vol.2, No.1, 1998.
19. J.Waldo, G.Wyant, A.Wollrath and S.Kendall, "A Note on Distributed Computing", Technical Report, Sun Microsystems Labolatories, Inc., 1994.
20. Mark Weiser, "The Computer for the 21st Century", Scientific American, Vol. 265, No.3, 1991.

Access Control and Trust in the Use of Widely Distributed Services

Jean Bacon, Ken Moody, and Walt Yao

University of Cambridge Computer Laboratory
New Museum Site, Pembroke Street
Cambridge CB2 3QG, United Kingdom
{jean.bacon, ken.moody, walt.yao}@cl.cam.ac.uk

Abstract. OASIS is a role-based access control architecture for achieving secure interoperation of independently managed services in an open, distributed environment. OASIS differs from other RBAC schemes in a number of ways: role management is decentralised, roles are parametrised, and privileges are not delegated. OASIS depends on an active middleware platform to notify services of any relevant changes in their environment.
Services define roles and establish formally specified policy for role activation and service use; users must present the required credentials and satisfy specified constraints in order to activate a role or invoke a service. The membership rule of a role indicates which of the role activation conditions must remain true while the role is active. A role is deactivated immediately if any of the conditions of the membership rule associated with its activation become false.
Instead of privilege delegation OASIS introduces the notion of appointment, whereby being active in certain roles carries the privilege of issuing appointment certificates to other users. Appointment certificates capture the notion of long lived credentials such as academic and professional qualification or membership of an organisation. The role activation conditions of a service may include appointment certificates, prerequisite roles and environmental constraints.
We define the model and architecture and discuss engineering details, including security issues. We illustrate how an OASIS session can span multiple domains, and discuss how it can be used in a global environment where roving principals, in possession of appointment certificates, encounter and wish to use services. We propose a minimal infrastructure to enable widely distributed, independently developed services to enter into agreements to respect each other's credentials.
We speculate on a further extension to mutually unknown, and therefore untrusted, parties. Each party will accumulate audit certificates which embody its interaction history and which may form the basis of a web of trust.

1 Introduction

OASIS is an access control system for open, interworking services in a distributed environment modelled as domains of services. Services may be developed inde-

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 295–310, 2001.

pendently but service level agreements allow their secure interoperation. OASIS is closely integrated with an active, event-based middleware infrastructure. In this way we can notify applications of any change in their environment, making it possible to ensure that security policy is satisfied at all times.

OASIS is role based but has important differences from other RBAC schemes [3,4,5,6,7,8,11,12,13,14,15,16]:

- Roles are service-specific; there is no notion of globally centralised administration of role naming and privilege management.
- Roles may be parametrised, as required by applications.
- Roles are activated within sessions. A session is started by activating an initial role such as *logged_in_user*. Most roles have activation conditions that require prerequisite roles and a session of active roles is built up.
- All privileges are associated with roles. We use appointment instead of privilege delegation; the activation conditions of roles which convey the privileges to be granted may require appointment certificates.
- We provide an *active security environment*. Constraints on the context can be checked during role activation; the role may be deactivated if particular conditions become false subsequently.

Services name their client roles and enforce policy for role activation and service invocation, expressed in terms of their own and other services' roles. Section 2 gives the details of role activation conditions. An encryption protected role membership certificate (RMC) is returned to the user on successful role activation and this may be used as proof of authorisation to use this and other services and as a credential for activating other roles, according to services' policy. Engineering details are in Sect. 4.

In contrast with some RBAC schemes we do not support the delegation of privileges. Instead we use *appointment*, in which one function of certain roles is to issue appointment certificates which may be used, together with any other credentials required by policy, to activate one or more roles. There is no reason why the holder of the appointer role should be entitled to the privileges conferred by the certificates; for example, a hospital administrator need not be medically qualified. Appointment is discussed in Sect. 2.

RBAC, in associating privileges with roles, provides a means of expressing access control which is scalable to large numbers of principals. The detailed management of large numbers of access control lists, as people change their employment or function, is avoided. However, pure RBAC associates privileges only with roles, whereas applications often require more fine-grained access control. Parametrised roles extend the functionality to meet this need. Role activation conditions can check parameter values and the relationship between parameters, so that policy can express exceptions to the default access controls. OASIS role membership certificates are always principal specific but may or may not be parametrised, depending on the requirements of the application. Some applications may require anonymous certificates. Section 5 discusses this issue in more detail.

Appointment meets the requirement for long-lived credentials; in contrast privileges are associated with roles, which are activated in the context of a session. This allows an active security environment to be maintained, in which any breach of role membership conditions can be notified. Sections 4 and 5 discuss the architecture and engineering of sessions.

In practice, distributed systems contain many domains; for example the healthcare domain comprises subdomains of public and private hospitals, primary care practices, research institutes, clinics, etc. as well as national services such as electronic health record management. Access control policy may be dictated by national law and/or may come from organisational decisions. In [1] we present an early attempt at pseudo-natural language policy expression and its automatic translation into first-order predicate calculus. Such a mechanism is crucial for any large-scale deployment of policy; it is essential to maintain consistency as policies evolve. The formal expression of policy and its automatic deployment is an independent thread of research which is not the main focus here.

Widely distributed services may establish agreements on the use of one another's appointment certificates. Within a session at a user's place of work it is often necessary to make cross-domain invocations of remote services. For example, a doctor carrying out emergency treatment for a patient who is away from home may need access to the patient's health record. Another example is the negotiated use of remote digital libraries or databases. This is a natural part of the architecture and is described in Sect. 3.

Sections 5 and 6 discuss extended uses of the OASIS approach. Suppose someone temporarily leaves their home base to work as a known principal in a known domain. For example, a doctor employed in a hospital may need to work temporarily in a research institute, perhaps in another country. There may be a reciprocal agreement between the hospital and the research institute to accept electronic evidence of medical qualifications, subject of course to validation by the issuer.

A different scenario is that a member of some organisation may have the right to use the services of another, negotiated between the two organisations. An analogy is that the English and Scottish National Trusts may give privileges to each other's members. Here, any paid-up member of a local organisation may apply to use a known remote organisation.

The above scenarios assumed trusted services inhabiting mutually aware domains, such as the healthcare domains of different regions or nations. More generally, we may wish to set up an infrastructure for a world in which roving computational entities encounter previously unknown, and therefore untrusted, services. Both parties should be able to present checkable credentials to provide evidence of previous successful interactions. We discuss the difficulties of establishing such an infrastructure and look at some of the risks involved.

In summary: in Sect.s 2 and 3 we outline the OASIS model and architecture. In Sect. 4 we highlight some significant engineering issues, including the integration of OASIS access control with authentication and secure communication. In

Sect. 5 we discuss the use of OASIS for widely distributed but mutually aware domains. In Sect. 6 we speculate on how principals and services might have their interactions certified in order to establish a more general scenario that includes mutually untrusted parties.

2 The OASIS Access Control Model

[17] presents the OASIS model in detail including the formal semantics. Here we outline the basic model.

OASIS embodies an open, decentralised approach, appropriate to deployment in distributed systems. Roles are defined by services and services may interoperate, recognising one another's roles, according to service-level agreements. Central to the OASIS model is the idea of **credential-based role activation** at a service. The credentials that a user possesses, together with side conditions which depend on the state of the environment, will authorise him or her to activate roles.

Activation of any role in OASIS is explicitly controlled by a **role activation rule**. A role activation rule specifies, in Horn clause logic, the conditions that a user must meet in order to activate the role. The conditions may include **prerequisite roles, appointment credentials** and **environmental constraints**.

A **prerequisite role** as a condition for a target role means that a principal must already have activated the prerequisite role before it can activate the target role. Some services define roles which do not include prerequisite roles in their role activation rule. An example is a login service which defines a role *logged_in_user*. The activation conditions might include proof that the user is employed by or registered at the organisation to which the system belongs, and can supply authentication evidence, but a prerequisite role is not required. By activating such a role, called an initial role, a user may start an OASIS session; other roles' activation rules may include the prerequisite role *logged_in_user*. Figure 1 illustrates role dependency, with a role activation rule for service C that includes only prerequisite roles.

Appointment credentials were motivated in Sect. 1. Being active in certain roles gives the principal the right to issue appointment certificates to one or more other principals. Appointment certificates may be used, together with any other credentials required by policy, to activate one or more roles. They are certificates whose lifetime is independent of the duration of the session of activation of the appointer role. They may be long-lived, such as when they are used to certify academic or professional qualification, employment or membership of an organisation. They may be transient, for example when certifying that someone is authorised to stand in for a colleague who is called away while on duty. In some cases the appointment is made with the intention of allowing the appointee to activate a specific role. An example is that a screening nurse in an Accident and Emergency (A&E) Department may allocate a patient to a particular doctor. He/she issues an appointment certificate to the doctor who may then activate the role *treating_doctor* for that patient. In other cases the

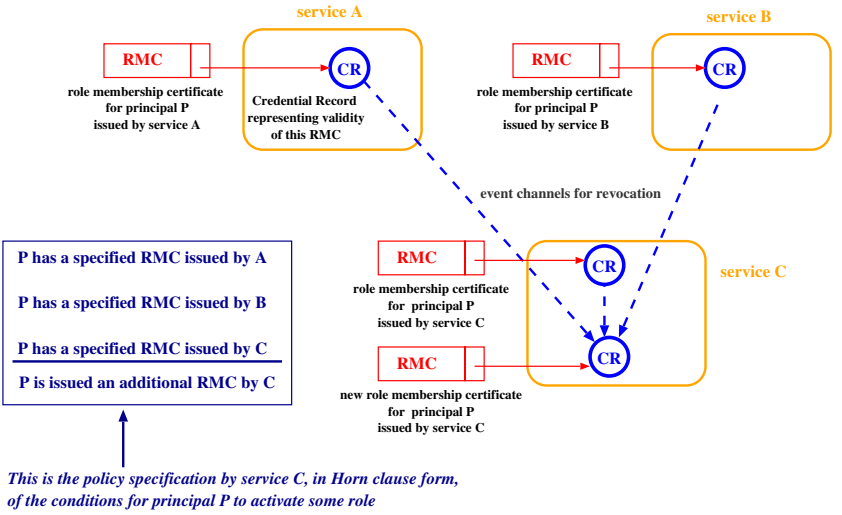


Fig. 1. Role dependency through prerequisite roles

appointer may have no reason to know which roles will include the appointment certificate in their activation rules. In general, there is no reason why the appointer should be entitled to the privileges conferred (through subsequent role activation) by the appointment certificates it issues. In a computerised system it is natural that an administrator should enter a role and issue academic, professional or membership appointment certificates; a hospital administrator need not be medically qualified. If an application requires delegation then it can be built using appointment. The role of the *delegator* must be granted the privilege of issuing appointment certificates, and a role must be established to hold the privileges to be assigned. Finally an activation rule must be defined to ensure that the appointment certificate is presented in an appropriate context.

OASIS roles are parametrised; without parameters RBAC cannot meet the requirements of some application domains. For example, the English "Patients' Charter" allows patients to specify who may see which parts of their health records; the policy deployed at any service which provides health care should take account of such directions by the patient. Default policy is role based for scalability; for example "doctors may access the records of patients registered with them". Policy as implemented must respect individual exceptions indicated by patients; for example "Fred Smith" (although a doctor) "may not access my health record". Constraint checking during service use allows such exceptions to be enforced; it is vital that doctors who access patient records may be identified individually. For access control in a file service it is necessary to indicate individual owners of files as well as groups of users. In general, OASIS role parameters might be the identifier or location of the computer, the name of the activator of

the role, some identifier of the activator, such as a public key or health service identifier, the patient the activator is treating, and so on.

As noted above, in addition to prerequisite roles and appointment credentials, role activation rules may include **environmental constraints**. These may be user-independent constraints or conditions on user-dependent parameters. Examples of user-independent constraints are the time of day and the location or name of a computer. An example of a condition on user-dependent parameters is that the user is a member of a group; this may be ascertained by database lookup at some service. Another is that parameters are related in a specified way; for example the doctor has the patient registered as under his/her care. Again, a check can be made against a database. A simple parameter check may ascertain that the user is a specified exception to a general category who may activate the role.

The membership rule for a role indicates those predicates for activating the role that must continue to be true for the role to remain active. In Sect. 4 we show how membership rules are monitored and enforced in our active environment.

As in traditional RBAC, roles convey privileges; specifically, the privilege of method invocation (including object access) at services. The conditions for service invocation are possession of role membership certificates of this and other services together with environmental constraints. Since roles are parametrised we can enforce policies on object access such as "doctors may access patients' health records" together with exclusions on individual objects: "Joe Bloggs' health record may not be accessed by Fred Smith".

3 OASIS Architecture

Figure 2 shows the architecture of an OASIS service which defines roles. Services may also be OASIS-aware and specify roles of other services as credentials to authorise their use, without themselves defining roles, see [10].

A client activates a role by presenting credentials to a service that enable the service to *prove* that the client conforms to its policy for entry to that role (path 1). The service validates the credentials. This may involve checking back with certificate-issuing services and checking environmental conditions. If the checks succeed a *role membership certificate* (RMC) is issued (path 2). This may be presented with subsequent requests to use that service, together with any other credentials specified by the service's policy (path 3). The service validates the credentials, checks any constraints and, if all is well, the invocation proceeds (path 4). The design of RMCs and how they are validated is discussed in Sect. 4.

An OASIS session may include remote, cross-domain service invocations based on prior service-level agreements. Figure 3 gives an example, taken from a national electronic health record (EHR) service. We assume a national EHR domain and many local healthcare domains such as primary care groups and hospitals. A hospital domain is likely to include many computerised services such as those which record pharmacy and X-ray usage. Figure 3 shows one of

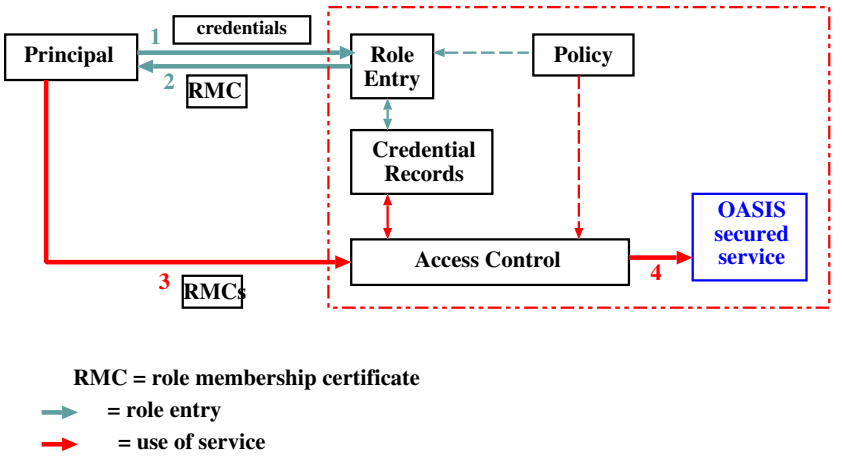


Fig. 2. A service secured by OASIS access control

these services; that which delivers EHR components as requested by authorised principals.

The scenario is that a doctor has successfully activated the parametrised role *treating_doctor (doctor_id, patient_id)* within the hospital domain where he/she is employed. The doctor needs to refer to certain records of treatment within the patient's EHR and invokes the local EHR service to make the request, with the RMC *treating_doctor* as credential. The local EHR service is OASIS aware, validates the credential by callback to the hospital administration, and the invocation proceeds. The local EHR service must now invoke the national EHR service to locate and acquire the EHR components. The national EHR service has a rolename which principals from authorised healthcare domains can activate, say *hospital(hospital_id)*. In the figure the role is active and the invocation **request-EHR** is made (path 1). Service level agreements between the national service and individual health care domains would establish a protocol to validate local RMCs so that the identity of the original requester can be recorded for audit. The detailed design of the national EHR service would specify the credentials and parameters required for the invocation. For example, the hospital certificate is the required credential and the *treating_doctor (doctor_id, patient_id)* certificate would form part of the audit record. The doctor and patient ids are shown embedded in the *treating_doctor* certificate but they could be included, in addition, as arguments of the calls.

The national EHR service validates the hospital certificate, notes that the requester is in the role *treating_doctor* and uses the parameter *patient_id* to find information on where components of the EHR are stored. Before returning the data to the hospital it is essential to check that the role is sufficient to permit access to the fields requested and that this particular doctor has not been excluded from access by the patient. If all is well the data is returned

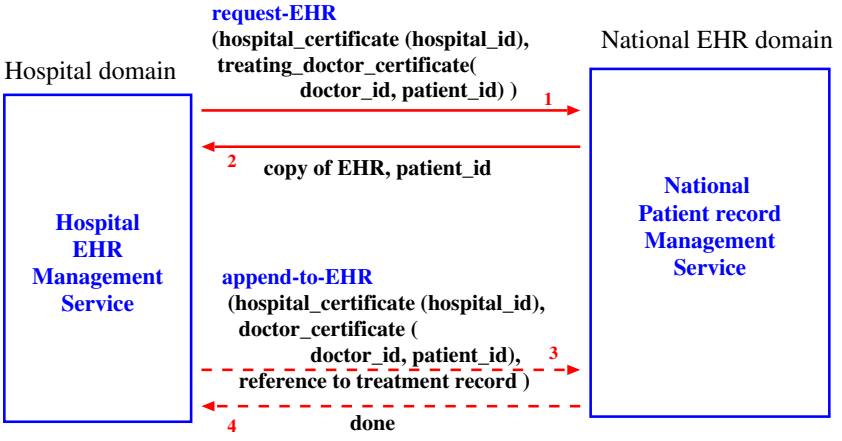


Fig. 3. An OASIS session with cross-domain calls

(path 2). The doctor carries out an item of treatment. Paths 3 and 4 show the authorised and audited inclusion of the record of treatment in the patient's EHR. The architecture must guarantee that these steps complete successfully, but the response time is not critical. This is a simplification of the way in which EHRs are likely to be managed.

The above examples illustrate how OASIS access control operates within and across domains. We now expand on some engineering issues.

4 OASIS Engineering

There need not be a centrally dictated design of role membership certificates, although there is likely to be a unified design within each domain. An Oasis-aware service will validate a certificate presented as an argument via callback to the issuer. The service may cache the certificate and the result of validation in order to reduce the communication overhead of repeated callback. This requires an event channel so that the issuer can notify the service should the certificate be invalidated for any reason, see [2].

Figure 4 presents a possible RMC design. RMCs are encryption-protected to guard against tampering and are principal-specific to guard against theft, as shown in the figure. The *principal_id* is discussed further in Sect. 4.1. The role-name and any parameters may be recorded in the RMC and protected. The owner's personal identification may or may not be one of the parameters, depending on application requirements.

The issuer keeps information on the RMC, including its current validity, in a credential record (CR). The credential record reference (CRR) in the RMC allows the issuer and the CR to be located. Details of how the CR might be

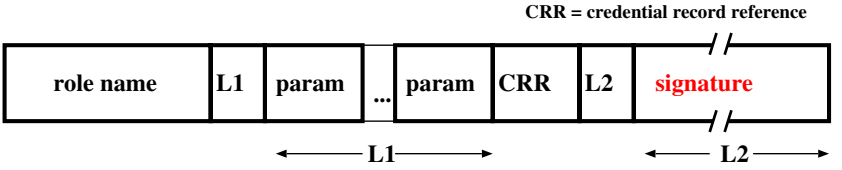


Fig. 4. A possible role membership certificate (RMC) design

designed were given in [9]. [10] discussed engineering issues for OASIS implementations in administrative domains that span many individual services. It is likely that certificates will not be issued and validated by each individual service as is possible in the architecture. Rather, a domain will contain one highly available service to carry out the functions of certificate issuing and validation. The paper outlined the design of such a service, including replication for availability together with consistency management.

It is important to note that OASIS is integrated with an event infrastructure [2]; this allows services protected by OASIS to communicate asynchronously, so that one service can be notified of a change of state at another without any requirement for periodic polling.

An OASIS session typically starts from the activation of an initial role, such as authenticated, *logged_in_user*. A user may activate further roles by submitting the credentials required to satisfy an activation rule for each. Active roles therefore form trees of role dependencies rooted on initial roles. If a single initial role is deactivated, for example the user logs out, all the active roles dependent on it collapse and that session terminates.

The event-based middleware infrastructure is used to monitor the membership conditions of active roles. Should any membership condition for a role become false the role is deactivated and the dependent subtree is collapsed. Figure 5 shows event channels that capture role dependencies in an OASIS session.

4.1 Engineering Authentication and Secure Communication

Here we discuss how OASIS might be integrated with standard authentication and encrypted communication. Within a firewall-protected domain OASIS might be used without encrypted communication and yet still provide acceptable access control. Colleagues are unlikely to wire-tap, although looking at a colleague's screen may well reveal confidential data. Appointment certificates might be copied from file spaces if care is not taken to protect them, but a thief should not be able to exploit them if the activation rules are well designed. Although call and return parameters are potentially visible "on the wire" the persistent data is protected from uncontrolled update.

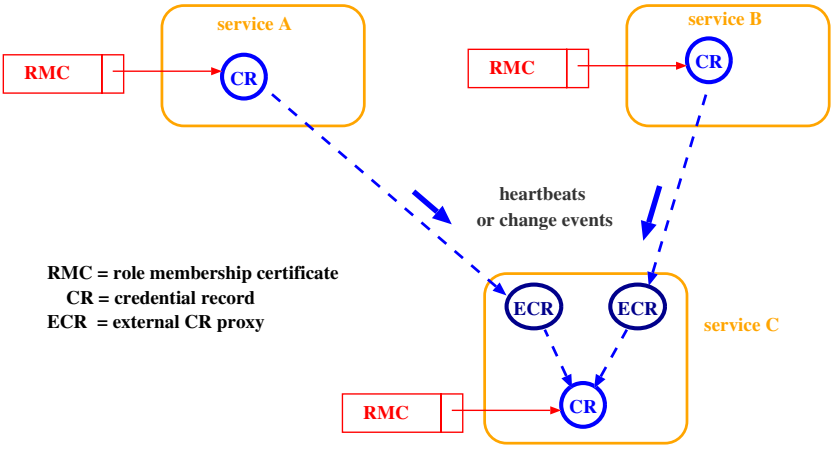


Fig. 5. Active security via an event infrastructure

If any visibility of data and certificates "on the wire" is unacceptable to an application, which must be assumed to be the case with cross-domain interworking, then encrypted communication must be used. Sensitive data might be encrypted selectively within a trusted domain. Data sent to a service can be encrypted with the service's public key and the public key of the caller can be included for encrypting the reply.

First, suppose that certificates are sent and stored in clear in a local domain. Security properties and related issues are as follows:

- **Protection from tampering.** The fields of role membership and appointment certificates are protected by their signatures.
- **Protection from forgery.** A secret held by the issuing service is an argument to the encryption function of role membership and appointment certificates. A correct signature cannot be generated except by knowledge of the secret.
- **Protection of RMCs from theft.** OASIS RMCs are principal-specific. Although not visible as a parameter field in the RMC, a principal_id is an argument to the encryption function that generates the signature, see Fig. 4. A stolen RMC therefore cannot successfully be used by an adversary unless the principal_id can be guessed and forged. The choice of a suitable id is discussed below. In addition, an RMC can contain principal-related information as parameters in protected fields; this can help to prevent thieves from abusing appointment certificates.
- **Choice of a principal_id for RMCs.** Since OASIS is session-based a session-specific principal_id could be used, which would provide better security than a persistent principal_id. An unforgeable location-dependent principal_id would be ideal but that may not be easy to create. The principal, of course, would need to run the whole session from that location. The host_id,

IP address and process_id of the caller are perhaps too easy to forge. An alternative is that a key-pair can be created by the principal and the public key sent to the service to be bound into the certificate. The service can establish at any time that the caller holds the corresponding private key by running a challenge-response protocol. This is discussed further below.

– **Protection of appointment certificates from theft.** Unlike RMCs, appointment certificates have a lifetime which is independent of a session. They cannot be made principal-specific by using a session-dependent principal identifier. Like RMCs, the detailed design is issuer-dependent and the issuer validates them on demand. They can be made principal-specific by including a persistent principal_id as an argument to the encryption function, such as a long-lived public key of the principal. A long-lived appointment certificate is more vulnerable to attack than an RMC and it is likely that appointment certificates would be re-issued, encrypted with a new server secret, from time to time.

Many RBAC schemes do not have the OASIS distinction between potentially long-lived appointment certificates and session-based RMCs. An implementation of long-lived role membership would carry the same vulnerability to attack as OASIS appointment certificates. Session-based role activation is more secure, provided that we have strong authentication when an initial role is activated. Authentication is discussed further below.

Authentication

OASIS assumes that principals have been authenticated when appointment certificates and RMCs are granted to them. When a certificate is subsequently presented as a credential the assumption is that the principal presenting it is the principal to whom it was issued.

Authentication of a principal at the start of an OASIS session, when a long-lived appointment credential may be used to activate an initial role, is therefore an important issue. Subsequently, the RMC for the initial role is effectively an authentication token and all subsequent role activations and service invocations are provably from the same principal, as discussed above. Authentication might be through password-protected login in a working environment. In the future, biometrics will be used increasingly and might be included in an appointment certificate.

Integration with PKC

OASIS can be integrated with public/private key cryptography for authentication as well as secure comunication. A public key of the activator of an initial role could be used as the session key described above, bound into the signature of every subsequent RMC and sent to the service under the service's public key. The service can check that the activator has the corresponding private key by using a challenge-response protocol, such as ISO/9798. The issuing service produces

a random challenge, encrypted with the public key presented by the activator, and a nonce. The client must respond with the challenge in plaintext encrypted with the nonce. Upon receiving this, the service can conclude that the activator has access to the private key corresponding to the public key presented, and can safely bind it into the signature. A similar challenge can be made at any time; in the extreme, every time a certificate is presented. In practice the challenge might be made at random during a session, and at selected times such as before sensitive data is sent.

5 OASIS for Multiple, Mutually-Aware Domains

Section 4 covered the case where a session within someone's normal workplace domain includes the invocation of services in other domains. We assumed prior agreement between domains/services on the use of each other's RMCs as credentials for role entry and/or service invocation. Now suppose someone wishes to work away from their home base temporarily, in a known domain. For example, a doctor employed in a hospital may need to work for a short time in a research institute, without changing employment.

The hospital and research institute trust each other; they are subdomains within a national healthcare domain. They agree that the home domain's administrative service will issue an appointment certificate to the doctor. This will serve as a credential for entering the role *visiting_doctor* in the research institute; a role which has more privileges than a minimal visitor's role such as *guest*.

In the home domain appointment certificates *employed_as_doctor (hospital_id)* are issued only to members of staff who can prove that they are academically and professionally qualified in medicine. Their credentials are checked before the *employed_as_doctor* appointment certificate is issued. The doctor can enter the role *visiting_doctor* in the research institute through an activation rule which recognises the home domain appointment certificate as a precondition that proves that the doctor is medically qualified; this activation rule is part of the policy established by the service level agreement between the hospital and the research institute. The research institute would check the validity of the appointment certificate during role activation by callback to the hospital.

The reciprocal side of the agreement would allow medical research workers to work temporarily in the hospital. The appointment certificate *research_medic (research_institute_id)* might similarly serve as a credential for activating medical roles at the hospital, subsuming *employed_as_doctor (hospital_id)* etc.

The fields of appointment certificates (and RMCs) are readable, although protected from tampering and theft as described above. Parameters recorded there may be read and any environmental conditions checked.

Group Membership Negotiations

A different scenario is that a member of some organisation may have rights to use the services of another, negotiated between the two services. An analogy

is that an art lover can become a friend of the Tate Gallery, and receive the newsletter of Tate London, Tate St Ives or Tate Liverpool. A friend registered at one gallery will also receive the privileges of a friend at any other. Here, any paid-up member of a local organisation may apply to use a known remote organisation. The identity of the principal is not needed if proof of membership is securely provable. An appointment certificate (the electronic equivalent of a membership card) might indicate the organisation and the period of membership and might or might not include personal details.

Anonymity

Suppose that privacy legislation has been passed whereby someone who has paid for medical insurance may take certain genetic tests anonymously. The insurance company's membership database contains the members' data; the genetic clinic has no access to this. The insurance company must not know the results of the genetic test, or even that it has taken place. The clinic, for accounting purposes, must ensure that the test is authorised under the scheme.

A member of the scheme is issued a computer-readable membership card containing an appointment certificate and the expiry date. The member activates the role *paid_up_patient* at the clinic and proceeds to take the genetic test. The activation rule for the role comprises the appointment certificate and an environmental constraint requiring that the date of the (start of the) test is before the expiry date of the insurance scheme membership. The appointment certificate is validated at the issuing service (a trusted third party) before role activation can proceed.

6 Untrusted Environments and Principals

The above scenarios assumed trusted services inhabiting mutually aware domains, such as the healthcare domains of different regions or nations. Credentials issued by one service are accepted by another, and the role activation rule enforces the policy established when an agreement was set up between the services. This mechanism is only possible when the issuer of a credential is already known to the service receiving it.

More generally, we may wish to set up a minimal infrastructure, sufficient for a world in which roving computational entities encounter previously unknown, and therefore untrusted, services. Both parties should be able to present checkable credentials which provide evidence of previous successful interactions. This is analogous to the check on a person's credit record that is made before major purchases are authorised.

A certified record of an interaction between a principal and a service could contribute to the evidence of the trustworthiness of both parties. Such certificates might be exchanged and validated before a principal uses a previously unknown service. Each party may then take a calculated risk on whether to proceed: the service risks the client exploiting its resources in unintended ways, or failing to

pay an agreed charge; the client risks breach of confidentiality, and poor or partial fulfilment. A formal approach might be for the parties to negotiate a contract before the service is undertaken, and together sign a certificate recording the outcome.

If a certificate issuing and validation (CIV) service already exists in a domain its function might be extended to generate such a certificate. After an interaction subject to contract the CIV service creates an *audit_certificate* which it issues to both parties and validates on request. An *audit_certificate* could be engineered, as described in Sect. 4, to contain sufficient information for the issuing service to be located.

There are of course snags associated with this proposal. It is made on the assumption that CIV services work only on behalf of trustworthy services, and that they are themselves trustworthy. In practice, a client and service might collude to build up a false history of trustworthiness. Similarly, a rogue domain might provide valueless audit certificates, or repudiate those issued to clients who had acted in good faith. The domain of the auditing service for a certificate is a factor that must be taken into account when assessing the risk.

These are deep waters. OASIS RBAC has been designed for deployment in a distributed environment, and allows mutually trusting services to interact securely. Ubiquitous computing, electronic business and mobile agents together promise great benefits, if only a means can be found to bound the risks inherent in computational interaction between unknowns. What is needed is an approach which will allow a trust infrastructure to evolve despite Byzantine behaviour by a minority of the principals.

7 Conclusion

We have brought together the various aspects of OASIS, outlining its model and architecture and discussing engineering issues, with a view to extending its use. We have outlined the security properties of OASIS certificates and have shown how OASIS can be integrated with a standard authentication and encryption infrastructure.

OASIS has important differences from other RBAC schemes which reflect our application-driven, engineering approach. Decentralised role management is essential for widely distributed systems with independently managed components. Parametrised roles are essential for many applications. It is often necessary to express exceptions from generic role-based access and the relationships between parameters may be an essential part of an access control policy. A session based approach provides stronger security than is possible with long-lived roles; a session key can be bound with role membership certificates. Further, a session context allows role activation rules to include environmental constraints. We also monitor role membership conditions throughout a session by implementing OASIS above event-based middleware. This allows us to deactivate roles immediately as dictated by policy. All of these factors contribute to an active security environment.

Appointment subsumes the delegation of privilege and captures the requirement for long-lived credentials. The distinction between session-based role activation and potentially long-lived appointment credentials adds to the security of the system.

We have shown how OASIS is used within and across domains and how principals might be supported in moving to known but remote domains. Service level agreements, with check-back to the issuing service for validation, are the supporting mechanisms. We have shown that anonymous service invocation is possible. Finally we have proposed that audit certificates might be issued as a record of the invocation of a service by a principal. Such certificates provide a distributed record of the histories of services and principals and might form the basis for interaction between mutually unknown parties.

Acknowledgements. We acknowledge the support of the Engineering and Physical Sciences Research Council (EPSRC) under the grant *OASIS Access Control: Implementation and Evaluation*. Members of the Opera research group in the Computer Laboratory made helpful comments on earlier drafts of this paper. We acknowledge many stimulating discussions with our colleagues on the "Secure" proposal, submitted to the EU's Global Computing Initiative.

References

1. J. Bacon, M. Lloyd, and K. Moody. Translating role-based access control policy within context. In *Proceedings of Policy 2001, Policies for Distributed Systems and Networks*, volume 1995 of Lecture Notes in Computer Science, pages 107–119. Springer-Verlag, 2001
2. J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri. Generic support for distributed applications. *IEEE Computer*, pages 68–76, March 2000.
3. E. Barka and R. Sandhu. A role-based delegation model and some extensions. In *23rd National Information Systems Security Conference*, Baltimore, MD, October 2000.
4. E. Barka and R. Sandhu. Framework for role-based delegation models. In *16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 2000.
5. M. J. Covington, M. J. Moyer, and M. Ahamad. Generalized role-based access control for securing future applications. In *23rd National Information Systems Security Conference*, Baltimore, MD, October 2000.
6. D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security*, 2(1):34–64, Feb 1999.
7. L. Giuri and P. Iglio. Role templates for content-based access control. In *Second ACM Workshop on Role-Based Access Control*, pages 153–159, Fairfax, Virginia, November 1997.
8. C. Goh and A. Baldwin. Towards a more complete model of role. In *Third ACM Workshop on Role-Based Access Control*, pages 55–61, Fairfax, Virginia, October 1998.

9. R. Hayton, J. Bacon, and K. Moody. OASIS: Access Control in an Open, Distributed Environment. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 3–14, Oakland, CA, May 1998. IEEE.
10. J. Hine, W. Yao, J. Bacon, and K. Moody. An architecture for distributed OASIS services. In *Middleware 2000*, volume 1795 of Lecture Notes in Computer Science, pages 104–120, 2000.
11. J. D. Moffett and E. C. Lupu. The uses of role hierarchies in access control. In *Fourth ACM Workshop on Role-Based Access Control*, pages 153–160, Fairfax, Virginia, October 1999.
12. M. Nyanchama and S. Osborn. Access rights administration in role-based security systems. In J. Biskup, M. Morgernstern, and C. Landwehr, editors, *Database Security VIII: Status and Prospects*, 1995.
13. M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security*, 2(1):3–33, Feb 1999.
14. R. Sandhu. Role activation hierarchies. In *Third ACM Workshop on Role-Based Access Control*, pages 33–40, Fairfax, Virginia, October 1998.
15. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, Feb. 1996.
16. R. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *10th IEEE Computer Security Foundations Workshop*, pages 183–194, Rockport, Massachusetts, June 1997.
17. W. Yao, K. Moody, J. Bacon. A Model of OASIS Role-Based Access Control and its Support for Active Security. In *Proceedings, Sixth ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 171–181, Chantilly, VA, May 2001

Preserving Causality in a Scalable Message-Oriented Middleware

Philippe Laumay, Eric Bruneton, Noël De Palma, and Sacha Krakowiak

Laboratoire Sirac**,
INRIA Rhône-Alpes ; 655, Avenue de l'Europe ; 38334 Saint-Ismier Cedex ; France
Tel : +33 4 76 61 53 89 - Fax : +33 4 76 61 52 52
Philippe.Laumay@inrialpes.fr

Abstract. We present a solution to guarantee scalable causal ordering through matrix clocks in Message Oriented Middleware (MOM). This solution is based on a decomposition of the MOM in domains of causality, i.e. small groups of servers interconnected by router servers. We prove that, provided the domain interconnection graph has no cycles, global causal order on message delivery is guaranteed through purely local order (within domains). This allows the cost of matrix clocks maintenance to be kept linear, instead of quadratic, in the size of the application. We have implemented this algorithm in a MOM, and the performance measurements confirm the predictions.

1 Introduction

Message-oriented middleware (MOM) is growing in importance with the development of new applications made of loosely coupled autonomous components that communicate on large-scale networks. This class of applications (including stock exchange quotations, electronic commerce services, web-content provisioning) is characterized by asynchronous exchanges, heterogeneity, and requirements for scalability and evolution. MOM provides an infrastructure that transmits messages and events to the widely spread components of a service, gluing them together through logical coupling [1].

Asynchronous interaction over large-scale networks is a major source of non-determinism. Message ordering provides a way of reducing this non-determinism, as it allows the application designer to assert properties such as causal delivery of messages or atomic (uniform) broadcast. Industrial specifications and implementations of MOM are now integrating this aspect. For example, the CORBA Messaging reference specification [2] defines the ordering policy as part of the messaging Quality of Service.

However, usual message ordering mechanisms pose scalability problems, as an increase of the number of nodes and/or the distance between them degrades

** Sirac is a joint laboratory of Institut National Polytechnique de Grenoble, INRIA and Université Joseph Fourier.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 311–328, 2001.

the performance of classical clock-synchronization algorithms. A common ordering mechanism uses logical time [3] to order events according to the causal order [4]. The causal precedence relation induces a partial order on the events of a distributed computation. It is a powerful concept, which helps to solve a variety of problems in distributed systems like algorithms design, concurrency measurement, tracking of dependent events and observation [5].

In many applications, causal order based on logical time is not enough to capture the dependencies needed by the application's logic. Vector clocks bring progress over logical (scalar) timestamps by inducing an order that exactly reflects causal precedence [6,7,8]. Matrix clocks [9,10] extend vector clocks by capturing a global property, namely "what A knows about what B knows about C". Such shared knowledge is needed in many instances involving close cooperation, such as replica update management and collaborative work. However, matrix clocks require $O(n^3)$ global state size for a n-node system, and change propagation still needs $O(n^2)$ message size [11]. This precludes the use of matrix clocks for large-scale systems (several hundreds or thousands of nodes).

A solution to achieve scalability on a MOM is to divide the node interconnection graph in several smaller interconnected groups. Inter-group communication is performed by special nodes called causal router servers, which link two or more groups and are responsible for transmitting messages while maintaining global causal consistency. This solution reduces the amount of necessary information that needs to be stored and exchanged to maintain causality. However, to be scalable, it requires that all computations should be kept local; an algorithm that requires the cooperation of all nodes does not scale well.

This paper proposes a solution based on the splitting of the MOM in domains of causality, which improves performance while reducing causality related costs. We have proved the following property : **iff there is no cycle in the domain interconnection graph[1], local causal ordering in each domain guarantees global causal ordering**. Thus, through a purely local (domain-wise) algorithm, the scalability of causal ordering through matrix clocks in a MOM is greatly improved. With a suitable choice of domains, the communication costs are now linear (instead of quadratic) with respect to the application size (number of nodes). We have implemented this algorithm in AAA[2], a MOM developed in our group. The results of the performance measurements confirm the predictions.

This paper is organized as follows. Related work on MOM scalability is surveyed in Section 2. Section 3 presents the AAA MOM and Section 4 introduces the domains of causality and presents the proof of the main result on domain-based causality. Section 5 describes the implementation of causality domains in the AAA MOM. Section refPerformance-evaluation presents performance results. We conclude in Section 7.

[1] For a precise characterization of this property, see Section 4.2

[2] The AAA MOM was developed in the Sirac laboratory in collaboration with the Bull-INRIA Dyade consortium, and is freely available on the Web with an implementation of the JMS interface. `http://www.objectweb.org/joram/`

2 Related Work

Many solutions have been proposed to lower the cost of causal ordering by reducing the amount of control information. A first group of solutions is based on vector clocks, which require causal broadcast and therefore do not scale well. These solutions include [12], in which nodes are grouped in hierarchically structured clusters, and [13] in which nodes are organized in a Daisy architecture.

A solution based on Hierarchical Matrix Timestamps (HMT) is proposed in [14]. The HMT stores information about other nodes in the same domain and summarizes information about other domains. But this technique is specially adapted to update propagation in replicated databases using a weak consistency algorithm[3] and is not suitable for causal communication.

An original solution for causal delivery is introduced in [15]. This solution does not use a logical clock and implements the causal history relation with lists of causally linked messages. The nodes interconnection graph is split in *subnets* separated by *vertex separators*. This approach allows cycles in the subnet interconnection graph and reduces the size of exchanged information. It does not reduce the amount of control information necessary within a subnet, which is detrimental to scalability.

A last set of solutions is based on the interprocess communication topology. In [16], processes are assumed to communicate mostly with processes of the same group, and the causal history can be omitted for messages exceptionally addressed to a process of another group. The same idea was developed in [17]. The algorithm proposed improves the FM class of algorithms[4] and brings up some solutions to reduce the clock size. One solution uses the communication path : a process only keeps the information about the set of processes with which it may communicate. But this algorithm does not ensure the global causal delivery of messages.

From this brief survey, we may conclude that both the message size (on the network) and control information size (on the nodes) are crucial as far as scalability is concerned and must be treated with the same importance. The next two sections present our solution, using the AAA MOM as a test bed.

3 The AAA Environment

The AAA (Agent Anytime Anywhere) MOM [18] is a fault-tolerant platform that combines asynchronous message communication with a programming model using distributed, persistent software entities called agents. Agents are autonomous reactive objects executing concurrently, and communicating through an event-reaction pattern [19]. Agents are persistent and their reaction is atomic, allowing

[3] Each replica autonomously updates local copies and periodically propagates the log of update operations to other replicas.

[4] The authors define the FM algorithms as all the causal ordering algorithms which use a logical clock (counter, vector or matrix) and mark each event (message) with a timestamp information.

recovery in case of node failure. The Message Bus (i.e. the MOM) guarantees the reliable, causal delivery of messages. The MOM is represented by a set of agent servers (or servers for short) organized in a bus architecture (see Fig. 1).

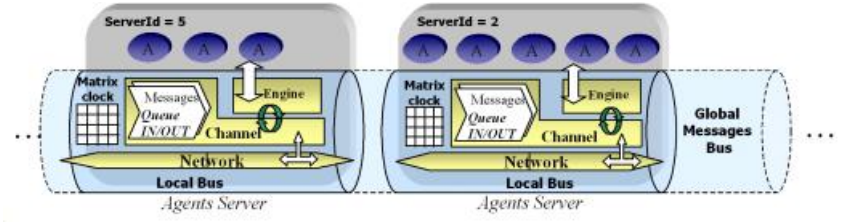


Fig. 1. Two interconnected Agent Servers details.

The combination of the agents and bus properties provides a solution to transient nodes or network failures, while causal ordering decreases the non-determinism of asynchronous exchanges. Each server is made up of three components, the *Engine*, the *Channel* and the *Network*, which are implemented in the Java language. The *Engine* guarantees the Agents' properties and the *Channel* ensures reliable message delivery and causal order. The causal order algorithm uses a matrix clock on each server, which has a size of n^2 for n servers. This causes two problems :

- Network overload, due to timestamp data exchange for clock updates. Even if only modifications of the matrix are sent instead of the full matrix (the *Updates* optimized algorithm described in Appendix A, which a similar approach can be found in [20].), the message size is $O(n^2)$ in the worst case.
- High disk I/O activity to maintain a persistent image of the matrix on each server in order to recover communication in case of failure.

The *Network* component is responsible to send the messages (basic communication layer). Our solution, presented in the next section, uses a decomposition approach to solve these two problems.

4 Domain-Based Causality

To solve the problem of the control information size for matrix clocks, we propose to replace single bus architecture by a virtual multi-bus (or Snow Flake) architecture [21]. Causality is only maintained on each individual bus, also called a *domain of causality*. The idea of splitting the set of servers is not new, but the published solutions either use vector clocks and broadcast [12,13], or use matrix clocks but only reduce the message timestamp size [14,15]. A quite similar solution based on group composition was used in [22], but the authors only prove that if the groups topology is a tree the end-to end ordering is respected, but

not the opposite. Moreover the ordering algorithm was not given, and no implementation exists. In our paper we prove that if the domains interconnection graph is cyclic then the global causality is not respected and vice-versa.

Our solution, combined with the *Updates* optimization, reduces both the message timestamp size and the control information size on servers. In addition, the modularity of the architecture allows it to be adapted to any physical or logical topology.

We have found that if causality is enforced in each domain, then it is automatically respected globally, provided **there is no cycle** in the domain interconnection graph. The proof of this property is presented in Section 4.3.

4.1 Domain of Causality

A domain of causality is a group of servers in which the causal order is respected. Adjacent domains are interconnected by a specific server, which has the function of router and is responsible of message transmission between domains while respecting causality. Such a server, which is in at least two domains, is called a *causal router-server*. In our system, the architecture is not imposed[5]. Domains may be freely organized in any acyclic graph and the logical architecture can be easily mapped on the real network topology[6] to improve the delivery algorithm. This technique reduces the size of the information exchanged on the network and furthermore decreases the size of the control information on the servers. Both factors improve the scalability of the MOM (a performance analysis and experimental results are given in section 6).

As an example (see Fig. 2) an 8-server MOM is logically split in four domains. Domain A includes {S1,S2,S3}, domain B includes {S4,S5}, domain C includes {S7,S8} and domain D and E includes respectively {S3, S6} and {S1,S4}.

Causal message ordering is enforced in each single domain. When a client (agent) connected e.g. to server 1 needs to communicate with a client connected to server 8, the message must be routed (like an IP packet on a network) using paths S1→S3 (domain A), S3→S6 (domain D), S6→S8 (domain C). Message routing is ensured by the system and is completely invisible to the clients, which are not aware of the interconnection topology. A server that needs to send a message to a server in another domain must send it to a causal-router-server (servers S1, S3, S4 and S6 in Fig. 2).

4.2 Domain-Based Computations

A distributed domain-based system is defined by a set $\mathbb{P} = \{p_1, ..., p_n\}$ of processes and a set $\mathbb{D} = \{d_1, ..., d_n\}$ of domains. The distribution of processes among domains is defined by a subset $\mathbb{R}$ of $\mathbb{P} \times \mathbb{D}$: process p is in domain d (noted $p \in d$)

[5] As opposed to the solution of [13], which imposes a daisy architecture and [12], which imposes a hierarchic architecture.

[6] In practice, most network are connected to form a spanning tree specifically to avoid cycles.

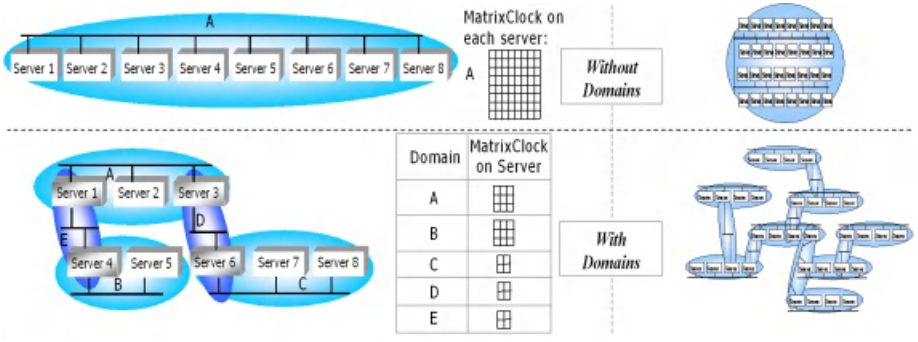


Fig. 2. Example of domains of causality

iff $(p, d) \in \mathbb{R}$. Processes communicate through messages. A computation is defined by a set of messages $\mathbb{M} = \{m_1, \ldots, m_q\}$. For each message m, there is a process $src(m)$, the sender, and a different process $dst(m)$, the receiver. The global history (or *trace*) of the computation is defined by the set of events associated with message send and receive. We assume that a process can only exchange messages with processes in the same domain (interdomain communication is examined later). Within a process p, we define a local order on messages sent or received by p : let m, m' be two such messages; then $m <_p m'$ means that (in the local time of process p) the sending or receiving of m precedes the sending or receiving of m'.

Causal precedence is usually defined between events; we extend it to messages as follows. Let m, m' be messages; m *causally precedes* m' (noted $m \prec m'$) iff one of the following three conditions holds :

- m and m' are sent by a process p, and m is sent before m' : $src(m) = p \wedge src(m') = p \wedge m <_p m'$.
- m is received by a process p, and p later send m' : $dst(m) = p \wedge src(m') = p \wedge m <_p m'$.
- there is a message n such that $m \prec n \wedge n \prec m'$.

We are only interested in *correct* traces, i.e. traces for which relation $\prec$ is a partial order ($m \not= m' \wedge m \prec m' \Rightarrow \neg(m' \prec m)$). A correct trace *respects causality* iff the order in which messages are received by each process p agrees with the causal order : $dst(m) = p \wedge dst(m') = p \wedge m \prec m' \Rightarrow m <_p m'$. A trace respects causality in domain d iff its restriction to d (i.e. its subset restricted to messages with source and destination in d) respects causality. As mentioned before, messages can only be exchanged between processes in the same domain. However, since a process may be included in several domains, indirect communication is possible, through a chain of messages, between processes in different domains. In order to formalize this indirect communication, we introduce the notions of path and chain.

A (process) *path* (of length c) from process p_1 to process p_c is a nonempty sequence $(p_1, ..., p_c)$ of processes such that any two consecutive processes in this

sequence are in the same domain : $\forall i < c, \exists d \in \mathbb{D}, p_i \in d \wedge p_{i+1} \in d$. We call p_1 the source of the path and p_c its destination. A *direct* path is one in which all processes are different (no loops); a *minimal* path is a direct path such that $i+1 < j \Rightarrow \neg(\exists d \mid p_i \in d \wedge p_j \in d)$ (the path does not "linger" in a domain). As a direct consequence, a minimal path of length > 2 has its origin and destination in different domains. A *cycle* is a direct path such that there is a domain including the source and the destination of the path, and there is no domain including all processes in the path. This generally corresponds to a cycle in the domain graph (two domains are connected in this graph if there is a process included in both domains) but not always (for example, if a domain is included in another one, a situation that does not occur in practice).

A *chain* (of length k) in a trace is a nonempty sequence $(m_1, ..., m_k)$ of messages such that each message (after the first one) is sent by the process that received the preceding message, after the receive : $\forall i < k, \exists p \in \mathbb{P}$, $dst(m_i) = p \wedge src(m_{i+1}) = p \wedge m_i <_p m_{i+1}$. We call $src(m_1)$ the source of the path and $dst(m_k)$ its destination. The path associated with a chain is $(src(m_1), src(m_2), ..., src(m_k), dst(m_k))$; this sequence is indeed a path since any two consecutive processes are in the same domain (since messages are local to a domain). A direct (resp. minimal) chain is a chain whose associated path is direct (resp. minimal).

Communication between processes in different domains is performed by means of "virtual messages". A *virtual message* between process p in domain d and process p' in domain d' is represented by a chain of (real) messages, with source p and destination p'. We can then define "virtual traces" in which all messages are virtual. A virtual trace may be formally defined as follows. Let T be a trace. We define T' as a *virtual trace* associated with T by defining a set $C = c_1, ..., c_k$ of minimal chains of T that do not "crossover" : if m_i and m_{i+1} are two consecutive messages of a chain $c \in C$, then no message of another chain $c' \in C$ can be sent by $p = dst(m_i)$ after m_i is received and before m_{i+1} is sent (see Fig. 3). Then T' is the trace derived[7] from T by considering each chain $(m_1, ..., m_k) \in C$ as a direct message from $src(m_1)$ to $dst(m_k)$. Several virtual traces may be derived from a (real) trace, including the real trace itself (by defining $C = \{(m_1), ..., (m_q)\}$. A correct virtual trace is one associated with a correct real trace.

4.3 The Main Theorem

Our main result gives the condition under which a virtual trace respects causality globally. It provides the base for an efficient, scalable causal message system based on domain decomposition.

Theorem 1. *The following two propositions are equivalent :*

P1 Any virtual trace associated with a correct trace that respects causality in each domain respects.

P2 The domain interconnection graph is acyclic.

[7] This derivation is straightforward, but its formal description is cumbersome.

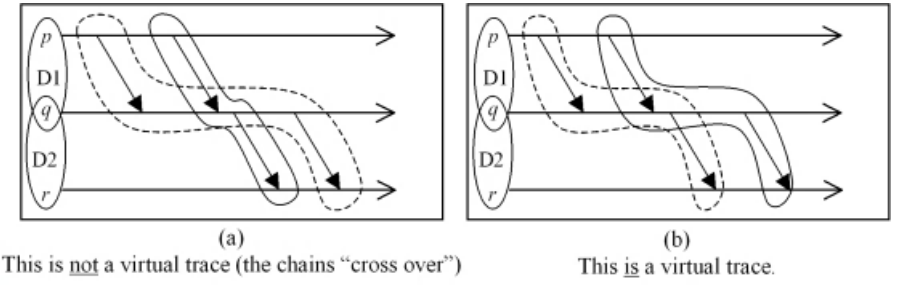


Fig. 3. Virtual traces and real traces

The proof of the theorem uses the following lemmas (see Appendix B for demonstrations).

Lemma 1 *In a correct trace, for any chain* $(m_1, ..., m_k)$ *whose source* p *and destination* q *are different, there exists a direct chain* $(n_1, ..., n_L)$ *with the same source and destination, such that* $m_1 \leq_p n_1$ *and* $n_L \leq_q m_k$.

Lemma 2 *If a correct trace does not respect causality, then there are two processes* p *and* q*, a message* n *from* p *to* q*, and a chain* $(m_1, ..., m_n)$ *from* p *to* q *such that* $n <_p m_1$ *and* $m_n <_q n$.

The proof of the main theorem is in two parts: we first prove $P1 \Rightarrow P2$ (by proving $\neg P2 \Rightarrow \neg P1$); then we prove $P2 \rightarrow P1$.

Part 1 : $\neg P2 \Rightarrow \neg P1$. If there is a cycle in the domain graph, there exists a correct virtual trace, associated with a correct real trace that respects causality in each domain, which does not respect causality globally.

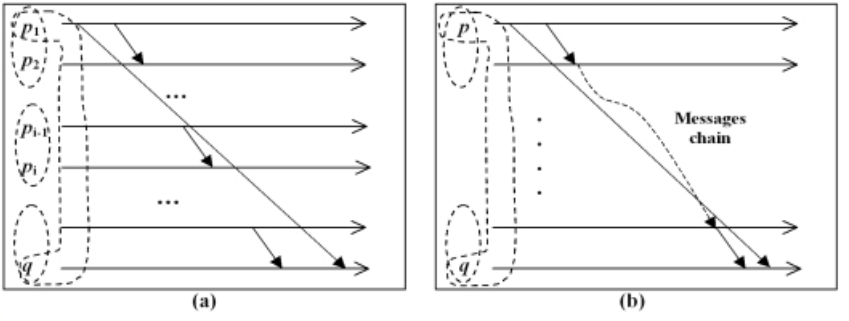


Fig. 4. Break of causality through a cycle of domains.

Since there is a cycle, there exists a direct path $(p, ..., p_i, ..., q)$ such that there is a domain including p and q, and there is no domain including all processes

of the path. Then consider the trace represented on Fig. 4 (a) (taking the real trace itself as a virtual trace).

This trace respects causality in each domain, because (a) no single domain includes all processes of the path; therefore, the restriction of the trace to any domain does not include all messages of the global trace; and (b) a trace derived from the original global trace by removing any single message respects causality. However, the trace does not respect global causality (consider the situation between p and q). This completes Part 1 of the proof.

Part 2 : $P2 \Rightarrow P1$. Lemma 2 may be reformulated as follows : if in a correct trace T there does not exist two processes p and q, a message n from p to q, and a chain $(m_1, ..., m_n)$ from p to q such that $n <_p m_1$ and $m_n <_q n$, then T respects causality. Therefore, in order to prove $P2 \Rightarrow P1$, we shall prove $P2 \Rightarrow P(x)$, where $P(x)$ is the following property : for any virtual trace associated with a correct trace that respects causality in each domain, there does not exist two processes p and q, a virtual message n from p to q represented by a minimal chain of length x, and a chain $(m_1, ..., m_n)$ of virtual messages from p to q such that $n <_p m_1$ and $m_n <_q n$. The proof is by induction on x.

Proof of $P(1)$ **:** By contradiction. Consider a correct virtual trace associated with a correct trace that respects causality in each domain. Assume there exists, for such a trace, two processes p and q, a virtual message n' from p to q represented by a chain of length 1 (the real message n), and a chain $(m'_1, ..., m'_n)$ of virtual messages from p to q such that $n' \leq_p m'_1$ and $m'_n \leq_q n'$ (this is the situation represented on Fig. 4 (b)).

Let $(m_1, ..., m_k)$ be the real chain representing the virtual chain $(m'_1, ..., m'_n)$. By Lemma 1, there is a direct chain $(n_1, ..., n_h)$ such that $m_1 \leq_p n_1$ and $n_h \leq_q m_k$. This chain cannot have length 1, otherwise causality would be violated in the domain including p and q (such a domain exists since there is a real message from p to q). Let then $(p, r, ..., q)$ be the (direct) path associated with this chain. No domain includes all processes of this path, because causality would be violated in such a domain. But p and q are in the same domain, hence $(p, r, ..., q)$ is a cycle, which contradicts $P2$. Therefore $P(1)$ is true.

Induction step : Assuming $P(y)$ holds for any $y < x$, we shall prove $P(x)$. The proof is by contradiction. Consider a correct virtual trace associated with a correct trace that respects causality in each domain. Assume there exists, for such a trace, two processes p and q, a virtual message n' from p to q associated with a minimal chain $(n_1, ..., n_x)$ of length $x > 1$, and a virtual chain $(m'_1, ..., m'_n)$ from p to q such that $n' < pm'_1$ and $m'_n <_q n'$ (this is similar to Fig. 4 (b), replacing the message from p to q by a virtual message).

Let $(p, a_1, ..., a_{x-1}, q)$ be the (minimal) path associated with chain $(n_1, .., n_x)$. Let $(m_1, ..., m_u)$ be the real chain corresponding to the virtual message chain $(m'_1, ..., m'_n)$. By Lemma 1, there exists a direct chain $(L_1, ..., L_v)$ such that $m1 \leq_p L_1$ and $L_v \leq_q m_u$. Let $(p, b_1, ..., b_{v-1}, q)$ be the corresponding direct path (note that $v > 1$, otherwise $(n_1, ..., n_x)$ would not be minimal). Now consider the path $(a_{x-1}, ..., a_1, p, b_1, ..., b_{v-1}, q)$. There are two possible cases.

Case 1 : all processes of path $(a_{x-1}, ..., a_1, p, b_1, ..., b_{v-1}, q)$ are different. In that case, there is a domain including a_{x-1} and q, and there is no domain including p and q (because the minimal path $(p, a_1, ..., a_{x-1}, q)$ has length > 2), and hence no domain includes all processes of the path. Therefore the path is a cycle, which contradicts $P2$.

Case 2 : not all processes of path $(a_{x-1}, ..., a_1, p, b_1, ..., b_{v-1}, q)$ are different. Since the ai are all different and distinct from p and q, and so are the b_j, there must be i and j such that $a_i = b_j$. By definition of a virtual trace, it is not possible for message L_{j+1} to be sent after n_i is received and before n_{i+1} is sent. Therefore, only the two cases represented on Fig. 5 may occur. In case (a), the virtual trace associated with the (non crossing) minimal chains $\{(n_1, ..., n_i), (L_1), ...\}$ does not satisfy $P(i)$, with $i < x$. In case (b), the virtual trace associated with the (non crossing) minimal chains $\{(n_{i+1}, ..., n_x), ...(L_v)\}$ does not satisfy $P(x - i)$, with $x - i < x$. Therefore, in both cases (a) and (b), there is a virtual trace that does not satisfy $P(y)$, $y < x$, which contradicts the induction hypothesis.

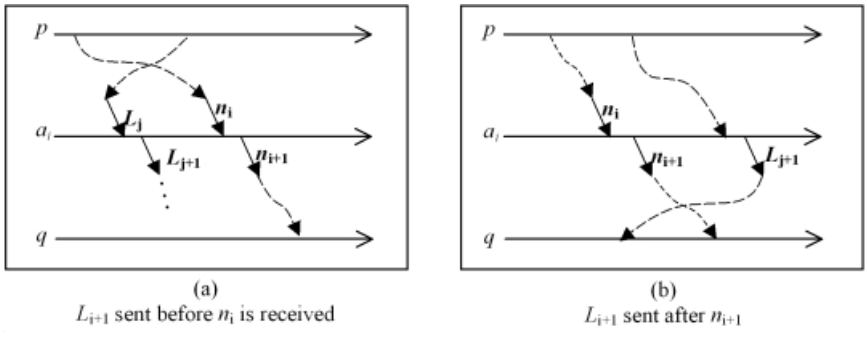


Fig. 5. "Cross over" of virtual traces.

In both cases 1 and 2, we find a contradiction. Therefore $P(x)$ is true, and so is $P(n)$ for any $n = 1$. This completes part 2 of the proof, $P2 \Rightarrow P1$.

5 Implementation

Recall that the MOM is logically split in a set of interconnected domains; causality is enforced within each domain and there is no cycle in the domain interconnection graph. This transformation of the MOM must be transparent for the clients; i.e. agent names must remain unchanged at the application level. Two problems need to be solved: the management of multiple matrix clocks (in the case of causal-router-servers, which belong to more than one domain), and message routing.

Server Modification

To solve the first problem, we have created on each server a local bus of domain (structure called *Message Consumer*), one for each domain that includes

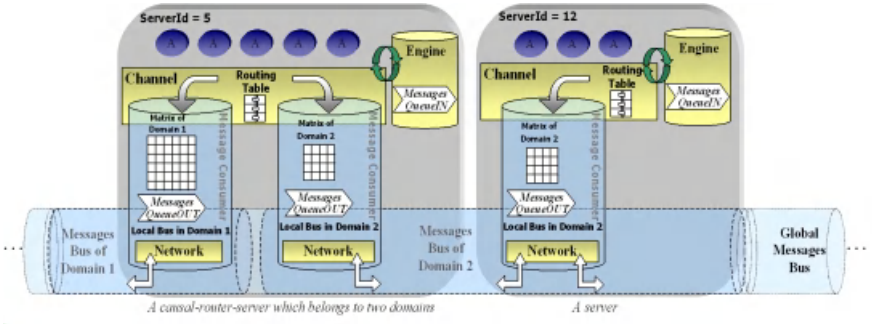


Fig. 6. Two examples of the new agent server structure.

the server. To solve the problem of message routing we have followed the classical network protocol approach, using a routing table. These new structures are represented in Fig. 6.

An agent server now has as many *MessageConsumer* as domain, which own a message queue and the matrix clock associated to its domain. With this implementation, a server can belong to an arbitrary number of domains, and any server can be a causal-router-server. The routing table gives, for each destination server, the identifier of the server to which the message should be sent: the destination server, within a domain, and a router server otherwise. The routing table is built statically at boot time. During server initialization, the servers and the routing tables are constructed, based on a shortest path algorithm.

The *Channel* ensures the transmission of messages and guarantees reliability and causal ordering (see section 3). The *Channel* put messages into the proper *MessageConsumer* using the routing table, then piggybacks messages with a matrix timestamp corresponding to the domain to which the message is sent. At reception, the recipient *Channel* checks the message timestamp and redirects the message to either the local queue (*QueueIN*) or the proper *MessageConsumer* according to its destination domain (see Fig. 7).

6 Performance Evaluation

6.1 Protocol Description

The simplest performance indicator for the AAA MOM is the turn-around time of a message between servers, which is the sum of two terms: the first one related to transfer itself (serialization-deserialization, transfer time, agent saving), the second one related to causal ordering (checking, updating and saving the matrix clock). The first term is nearly constant under our experimental conditions. Therefore the results are a good indicator of the efficiency of the causal ordering algorithm.

For the experiments, we have created an agent on each agent server, which sends back received messages (ping-pong). Messages are sent by a main agent on

Sender Message_Consumer

```
evt = Get_Agent_Sent_Event()
// an agent send an event
domainDestServer= RoutingTable[evt.dest]
messCons = MessageConsumer(domainDestServer)
// get the message consumer of
// the destination domain server
stamp = messCons.matrixclock(domainDestServer)
// get the stamp of the message
// = the updated matrix of the domain
msg = evt + stamp
messCons.network.Send(msg) ⟶
// the message is saved into
// MessageQueueOut and sent
Recv(ACK) ⟵
Remove(evt)
// from the MessageQueueOUT
```

Receiver Message_Consumer

```
⟶ msg = messCons.Recv
// receive a message from a MessageConsumer
// message is saved into LocalQueue
⟵ Send(ACK)
Check(messCons.matrixclock)
messCons.matrixclock.update(msg.stamp)
IF (evt.dest == this.server)
 | Push(evt, QueueIN)
 | // push event to message queue QueueIN
 | // the Agent destination will react to it
ELSE
 | systemDest= RoutingTable[evt.dest]
 | messCons = MessageConsumer(systemDest)
 | // get the message consumer of
 | // the destination domain
 | stamp = messCons.matrixclock(systemDest)
 | // get the stamp of the message
 | // = the updated matrix of the domain
 | msg = evt + stamp
 | messCons.network.Send(msg)
 | // the message is transferred into
 | // MessageQueueOut and sent to the
 | // next router-server or server
FI
```

Fig. 7. Channel Modification

server 0, which computes the round-trip average time for 100 sends. We did three series of tests: unicast on the local server, unicast on a remote server, broadcast on all servers. We did a series of experiments on a single host, but the number of servers was limited to 50 (because a single host cannot support more than 50 Java Virtual Machines). We then set up a network of ten hosts in order to increase the number of servers. We used PC Bi-Pentium II 450MHz with 512 Mo and a 9 Go SCSI hard drive, connected by a 100Mbit/s Ethernet adapter, running Linux kernel 2.0 or 2.2 (depending on machines). We only present the main results (full results are in [23]).

6.2 Experiments and Results

The initial measurements (with no causality domains) clearly show the quadratic increase of the message ordering cost with the number of servers, for both single host and multiple hosts experiments. Fig. 8 and Fig. 9 show typical results.

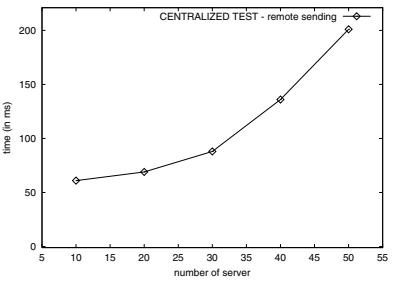


Fig. 8. Remote unicast without *domains of causality.*

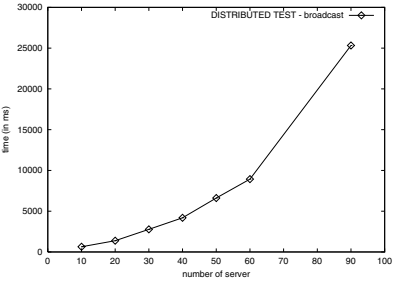


Fig. 9. Broadcast without *domains of causality.*

For the experiments with causality domains, we used a bus-like domain organization (Fig. 10). Other possible organizations are daisy and tree.

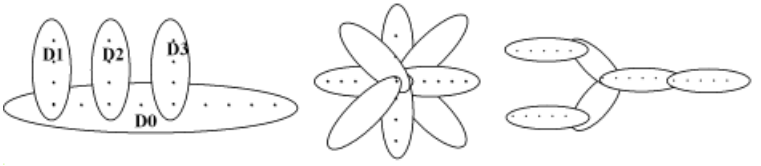


Fig. 10. A Bus, a Daisy and a Hierarchical division of domains.

With the bus-like organization, Fig. 11 shows a linear increase of the communication time (up to 150 servers, the limit of our experiment). To explain the linear dependency, consider a tree of domains of depth d, where each domain has k sub-domains exactly and s servers ($2 \leq k \leq s-1$), then the total number of servers is $n = 1 + (s-1)(k(d+1)-1)/(k-1) \approx sk^d$ and the maximum cost of sending a message is $C \approx (2d+1)s^2$ (the cost of sending a message in a domain of s servers is supposed to be s^2).

The linear cost results from our splitting in $\sqrt{n}$ domains of $\sqrt{n}$ servers with a fixed depth $d = 1$ (bus) which causes a cost of $C \approx K \times n$. For a general tree in which s and k are fixed (and $d > 1$), it would be possible to obtain logarithmic cost, because $C \approx 2ds^2 \approx 2s^2(\ln(n) - \ln(s))/\ln(k) \leq 2s^2 \ln(n)/\ln(k)$, so $C \approx K' \times \ln(n)$. However, $K' > K$ (in particular if we take into account the cost of message routing, proportional to d), therefore, a tree may be less efficient than a bus in some cases.

Fig. 12 clearly shows the performance gain brought by the use of causality domains.

All these results slightly depend on our peculiar test bed but nevertheless we believe that they can be considered sufficiently general for MOMs using a MatrixClock-based causal order with persistency.

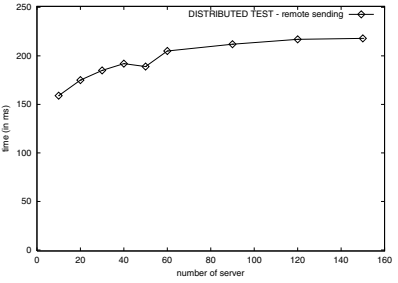


Fig. 11. Remote unicast with *domains of causality.*

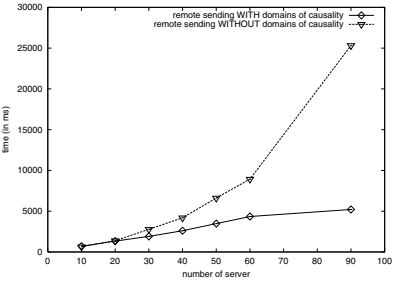


Fig. 12. Comparison of the cost with and without *domains of causality.*

7 Conclusion and Future Work

We have presented a solution based on *domains of causality* to reduce the cost of causal ordering in a scalable Message-Oriented Middleware based on matrix clocks. A domain of causality is a group of servers in which causal message delivery is enforced. The domains are interconnected by specific servers called causal router-servers which transmit messages between domains while guaranteeing the respect of causality. We have proved that the causality is globally respected in the entire network iff there is no cycle in the domains interconnection graph. Our solution has been successfully implemented on the AAA MOM and generates linear increase of the causal ordering costs with the number of servers, instead of quadratic increase with the classical causal ordering algorithm.

The modularity of this solution has many benefits. It is well adapted to a mobile environment (a group of mobile phones is represented by a domain and a station by a causal-router-server) and to LANs interconnection. However, the division of the MOM in domains needs to be done carefully and the new problem is to find an optimal splitting. Two directions may be followed: first, the splitting can be made according to the network architecture, secondly it can be made according to the application's topology. This latter solution exploits the description of applications (e.g. with an Architecture Description Language [24]) to obtain the application graph connectivity and to determine an optimal split of the communication architecture.

Our future work will further investigate the problem of optimal splitting of a MOM into domains, taking into account application needs and communication costs.

Acknowledgements. The authors gratefully acknowledge Luc Bellissard and André Freyssinet for their valuable and very useful advices on this work. We would also like to thank Jacques Mossière and the anonymous reviewers for their precious comments on earlier versions of this paper.

References

1. G. Banavar, T. Chandra, R. Strom, and D. Sturman. A case for message oriented middleware. In *Lecture Notes in Computer Science*, volume 1693, pages 1–18. Distributed Computing 13th International Symposium, September 1999. ISBN 3-540-66531-5.
2. Object Management Group. *CORBA Messaging*, May 1998. White Paper, http://www.omg.org.
3. L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21, pages 558–565, 1978.
4. K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. In *ACM Transactions on Computer Systems*, volume 9, pages 272–314, August 1991.
5. M. Raynal, A. Schiper, and S. Toueg. The causal ordering and a simple way to implement it. In *Information Proceeding Letters*, volume 39, pages 343–350, 1991.
6. O. Babaoglu and K. Marzullo. *Consistent Global States of Distributed Systems: Fundamental Consepts and Mechanisms*, chapter 3. S. Mullenderr, addison-wesley edition, 1993.
7. V. Hadzilacos and S. Toueg. Fault-tolerant broadcast and related problems. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.
8. F. Mattern. Virtual time and global states of distributed systems. In *the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.
9. M.J. Fischer and A. Michael. Sacrifying serializability to attain high availability of data in an unreliable network. In *ACM Symposium on Principles of Database Systems*, pages 70–75, March 1982.
10. G.T.J. Wuu and A.J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *3nd ACM Symposium on PODC*, volume 99, pages 233–242, 1984.
11. M. Raynal and M. Singhal. Logical time: A way to capture causality in distributed systems. In *IEEE Computer*, pages 49–56, 1995.
12. N. Adly, Nagi M., and J. Bacon. A hierarchical asynchronous replication protocol for large-scale systems. In *IEEE Workshop on Parallel and Distributed Systems*, pages 152–157, October 1993.
13. R. Baldoni, R. Freidman, and R. Van Renesse. The hierarchical daisy architecture for causal delivery. In *IEEE International Conference on Distributed Systems*, pages 570–577, May 1997.
14. T. Johnson and K. Jeong. Hierarchical matrix timestamps for scalable update propagation. submitted to the 10th Workshop on Distributed Algorithms, June 1996.
15. L. Rodrigues and P. Veríssimo. Causal separators and topological timestamping: an approach to support causal multicast in large scale systems. In *15th International Conference on Distributed Systems*, May 1995.
16. Rodrigues L. and P. Veríssimo. *Topology-Aware Algorithms for Large-Scale Communication*, volume 1752, pages 127–156. Springer Verlag LNCS, 2000.
17. S. Meldal, S. Sankar, and J. Vera. Exploiting locality in maintaining potential causality. In *the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 231–239, 1991.
18. L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. *An Agent Platform for Reliable Asynchronous Distributed Programming.* Symposium on Reliable Distributed Systems, October 1999.

19. G. Agha, P. Wegner, and A. Yonezawa. *Research Directions in Concurrent Object-Oriented Systems*. MIT Press, Cambridge, 1993.
20. A.D. Kshemkalyani and M. Singhal. An efficient implementation of vector clocks. In *Information Processing Letters*, volume 43, pages 47–52, August 1992.
21. A. Homer and D. Sussman. *Programmation MTS et MSMQ avec Visual Basic et ASP*, chapter 5. Eyrolles, Mars 1999.
22. S. Johnson, F. Jahanian, and J. Shah. The inter-group router approach to scalable group composition. In *19th ICDCS*, pages 4–14, June 1999.
23. Philippe Laumay. Déploiement d'un bus à messages sur un réseau à grande échelle. Rapport de DEA, University Joseph Fourier, June 2000.
24. L. Bellissard, S. Ben Atallah, F. Boyer, and M. Riveill. Distributed application configuration. In *International Conference on Distributed Computing Systems*, pages 579–585. IEEE Computer Society, May 1996.

Appendix

A The *Updates* Algorithm

State // an int value.
Mat[][] // an 2-dimension table representing the Matrix clock.
Mat$[x, y]$*.value // the value of the clock for [x,y], i.e. the real clock.*
Mat$[x, y]$*.state // the last modified state of the (x,y) matrix clock value.*
Mat$[x, y]$*.node // source node of last modification.*
Node$[x]$*.state // value of the x channel state during the last sending.*
Updates // timestamp sent; = a set of [line,column,value]
// (i.e. Mat$[line, column]$*.value).*

Initially :

$$\left|\begin{array}{l} State = 0; \\ \forall i, j Mat[i, j].state = 0; \\ \forall i, j Node[i, j].state = 0; \end{array}\right.$$

Sending from S_i to S_j :

$$\left|\begin{array}{l} Node[j].state = State; \\ Mat[i, j].value+ = 1; \\ Mat[i, j].state = State; \\ Mat[i, j].node = j; \\ State+ = 1; \\ Updates = \left\{ (k, l, Mat[k, l].value) \left| \begin{array}{l} Mat[k, l].state > Node[j].state \\ Mat[k, l].node \not= j \end{array} \right. \right\} \end{array}\right.$$

Receiving on S_i from S_j :

$$\left| \forall (k, l, v) \in Updates, Mat[k, l].value < v \rightarrow \left| \begin{array}{l} Mat[k, l].value = v \\ Mat[k, l].state = State \end{array} \right. \right.$$

B Proofs of Lemma 1 and Lemma 2

Lemma 1. In a correct trace, for any chain $(m_1, ..., m_k)$ whose source p and destination q are different, there exists a direct chain $(n_1, ..., n_L)$ with the same source and destination, such that $m_1 \leq_p n_1$ and $n_L \leq_q m_k$.

Proof : by induction on k. The property is trivially true for $k = 1$. Assume it holds for chains of length $k < K$, and consider a chain $(m_1, ..., m_K)$ of length K. If this is a direct chain, the proof is done. If not, then by definition the path $(p_1, ..., p_{K+1})$ is not direct, i.e. there exists $i < j$ such that $p_i = p_j$. Consider the chain $(n_1, ..., n_L)$ defined as follows[8] :

a) $(m_j, ..., m_K)$ if $i = 1 \wedge j < K + 1$.
b) $(m_1, ..., m_{i-1})$ if $i = 1 \wedge j = K + 1$.
c) $(m_1, ..., m_{i-1}, m_j, ..., m_K)$ if $i > 1 \wedge j < K + 1$.

This chain has the same source and destination as $(m_1, ..., m_n)$, and has length $< K$. By the induction hypothesis, there exists a direct chain $(n'_1, ..., n'_h)$ with the same source and destination, and such that $n_1 \leq_p n'_1$ and $n'_h \leq_q n_L$.

In addition, $m_1 \leq_p n_1$ and $n_L \leq_q m_k$. The first inequality trivially holds for case b) and c). It also holds in case a), for $m_j <_p m_1 \Rightarrow m_{j-1} <_p m_1 \Rightarrow m_{j-1} \prec m_1$ which is impossible because the trace is correct and we already have $m_1 \prec m_{j-1}$. Likewise, the second inequality also holds in the three cases. Hence, $m_1 \leq_p n'_1$ and $n'_h \leq_q m_k$. The property therefore holds for chains of length K, which concludes the induction step, and the proof.

Lemma 1' (needed for the proof of Lemma 2). If $n \prec m$, then either there is a chain $(n, ..., m)$, or there is a chain $(l, ..., m)$, where l is a message sent after n by the sender of n.

Proof : by recursive descent, using a case analysis at each step. If $n \prec m$, there are 3 possible cases :

- m and n are sent by the same process p, and m is sent after n. Then there is a chain of the form $(l, ..., m)$, where l is a message sent after n (take e.g. the chain (m)).
- m is sent by a process that previously received n. Then there is a chain of the form $(n, ..., m)$ (take e.g. the chain (n, m)).
- There exists a message k such that $n \prec k \wedge k \prec m$. By recursion (using the analysis of the 2 previous cases), one finds four possible cases, and in each case there exists either a chain $(n, ..., m)$ or a chain $(l, ..., m)$, where l is a message sent after n by the sender of n. The recursion terminates since all chains are finite.

Lemma 2. If a correct trace does not respect causality, then there are two processes p and q, a message n from p to q, and a chain $(m_1, ..., m_n)$ from p to q such that $n <_p m_1$ and $m_n <_q n$.

Proof : if a correct trace does not respect causality, then some process q received message m before message n, and $n \prec m$. By Lemma 1', there exists

[8] The case $i = 1 \wedge j = K + 1$ does not occur, because $p \neq q$.

either a chain $(n, ..., m)$ or a chain $(l, ..., m)$, where l is a message sent after n by the sender of n.

In the first case, since $dst(n) \neq src(m)$, chain $(n, ..., m)$ has the form $(n, l, ..., m)$. Then, by considering chain $(l, ..., m)$, $l \prec m \wedge m \prec l$ (since $m <_q n$ and $n <_q l$), which contradicts the assumption that the trace is correct (Fig. 13 (a)).

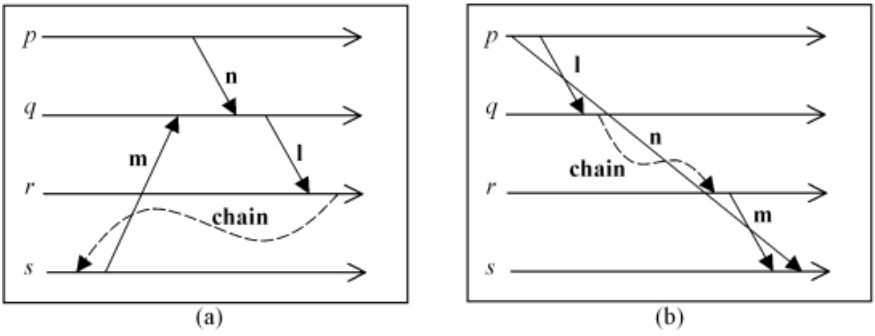


Fig. 13. Break of correctness (a) or causality (b).

In the second case, there is a message n from $p = src(n)$ to q, and a chain $(l, ..., m)$ from p to q such that $n <_p l$ and $m <_q n$ (Fig. 13 (b)). This concludes the proof.

Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems

Antony Rowstron[1] and Peter Druschel[2⋆]

[1] Microsoft Research Ltd, St. George House,
Guildhall Street, Cambridge, CB2 3NH, UK.
antr@microsoft.com
[2] Rice University MS-132, 6100 Main Street,
Houston, TX 77005-1892, USA.
druschel@cs.rice.edu

Abstract. This paper presents the design and evaluation of Pastry, a scalable, distributed object location and routing substrate for wide-area peer-to-peer applications. Pastry performs application-level routing and object location in a potentially very large overlay network of nodes connected via the Internet. It can be used to support a variety of peer-to-peer applications, including global data storage, data sharing, group communication and naming.

Each node in the Pastry network has a unique identifier (nodeId). When presented with a message and a key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries. Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a to scalar proximity metric like the number of IP routing hops.

Pastry is completely decentralized, scalable, and self-organizing; it automatically adapts to the arrival, departure and failure of nodes. Experimental results obtained with a prototype implementation on an emulated network of up to 100,000 nodes confirm Pastry's scalability and efficiency, its ability to self-organize and adapt to node failures, and its good network locality properties.

1 Introduction

Peer-to-peer Internet applications have recently been popularized through file sharing applications like Napster, Gnutella and FreeNet [1,2,8]. While much of the attention has been focused on the copyright issues raised by these particular applications, peer-to-peer systems have many interesting technical aspects like decentralized control, self-organization, adaptation and scalability. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric.

There are currently many projects aimed at constructing peer-to-peer applications and understanding more of the issues and requirements of such applications and systems [1,2,5,8,10,15]. One of the key problems in large-scale peer-to-peer applications

⋆ Work done in part while visiting Microsoft Research, Cambridge, UK.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 329–350, 2001.

is to provide efficient algorithms for object location and routing within the network. This paper presents Pastry, a generic peer-to-peer object location and routing scheme, based on a self-organizing overlay network of nodes connected to the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliable. Moreover, Pastry has good route locality properties.

Pastry is intended as general substrate for the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems. Several application have been built on top of Pastry to date, including a global, persistent storage utility called PAST [11,21] and a scalable publish/subscribe system called SCRIBE [22]. Other applications are under development.

Pastry provides the following capability. Each node in the Pastry network has a unique numeric identifier (nodeId). When presented with a message and a numeric key, a Pastry node efficiently routes the message to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. The expected number of routing steps is O(log N), where N is the number of Pastry nodes in the network. At each Pastry node along the route that a message takes, the application is notified and may perform application-specific computations related to the message.

Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a scalar proximity metric like the number of IP routing hops. Each Pastry node keeps track of its immediate neighbors in the nodeId space, and notifies applications of new node arrivals, node failures and recoveries. Because nodeIds are randomly assigned, with high probability, the set of nodes with adjacent nodeId is diverse in geography, ownership, jurisdiction, etc. Applications can leverage this, as Pastry can route to one of k nodes that are numerically closest to the key. A heuristic ensures that among a set of nodes with the k closest nodeIds to the key, the message is likely to first reach a node "near" the node from which the message originates, in terms of the proximity metric.

Applications use these capabilities in different ways. PAST, for instance, uses a fileId, computed as the hash of the file's name and owner, as a Pastry key for a file. Replicas of the file are stored on the k Pastry nodes with nodeIds numerically closest to the fileId. A file can be looked up by sending a message via Pastry, using the fileId as the key. By definition, the lookup is guaranteed to reach a node that stores the file as long as one of the k nodes is live. Moreover, it follows that the message is likely to first reach a node near the client, among the k nodes; that node delivers the file and consumes the message. Pastry's notification mechanisms allow PAST to maintain replicas of a file on the k nodes closest to the key, despite node failure and node arrivals, and using only local coordination among nodes with adjacent nodeIds. Details on PAST's use of Pastry can be found in [11,21].

As another sample application, in the SCRIBE publish/subscribe System, a list of subscribers is stored on the node with nodeId numerically closest to the topicId of a topic, where the topicId is a hash of the topic name. That node forms a rendez-vous point for publishers and subscribers. Subscribers send a message via Pastry using the topicId as the key; the registration is recorded at each node along the path. A publisher sends data to the rendez-vous point via Pastry, again using the topicId as the key. The rendez-vous point forwards the data along the multicast tree formed by the reverse paths

from the rendez-vous point to all subscribers. Full details of Scribe's use of Pastry can be found in [22].

These and other applications currently under development were all built with little effort on top of the basic capability provided by Pastry. The rest of this paper is organized as follows. Section 2 presents the design of Pastry, including a description of the API. Experimental results with a prototype implementation of Pastry are presented in Section 3. Related work is discussed in Section 4 and Section 5 concludes.

2 Design of Pastry

A Pastry system is a self-organizing overlay network of nodes, where each node routes client requests and interacts with local instances of one or more applications. Any computer that is connected to the Internet and runs the Pastry node software can act as a Pastry node, subject only to application-specific security policies.

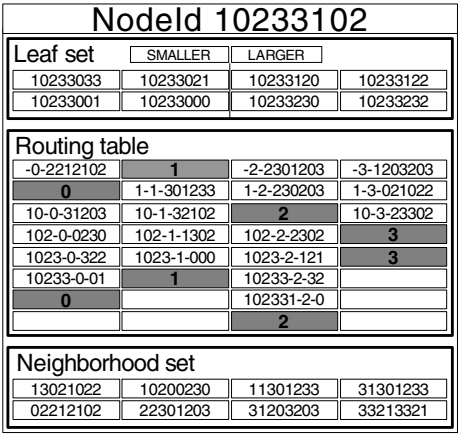
Each node in the Pastry peer-to-peer overlay network is assigned a 128-bit node identifier (nodeId). The nodeId is used to indicate a node's position in a circular nodeId space, which ranges from 0 to $2^{128} - 1$. The nodeId is assigned randomly when a node joins the system. It is assumed that nodeIds are generated such that the resulting set of nodeIds is uniformly distributed in the 128-bit nodeId space. For instance, nodeIds could be generated by computing a cryptographic hash of the node's public key or its IP address. As a result of this random assignment of nodeIds, with high probability, nodes with adjacent nodeIds are diverse in geography, ownership, jurisdiction, network attachment, etc.

Assuming a network consisting of N nodes, Pastry can route to the numerically closest node to a given key in less than $\lceil log_{2^b} N \rceil$ steps under normal operation (b is a configuration parameter with typical value 4). Despite concurrent node failures, eventual delivery is guaranteed unless $\lfloor |L|/2 \rfloor$ nodes with *adjacent* nodeIds fail simultaneously ($|L|$ is a configuration parameter with a typical value of 16 or 32). In the following, we present the Pastry scheme.

For the purpose of routing, nodeIds and keys are thought of as a sequence of digits with base 2^b. Pastry routes messages to the node whose nodeId is numerically closest to the given key. This is accomplished as follows. In each routing step, a node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the present node's id. If no such node is known, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's id. To support this routing procedure, each node maintains some routing state, which we describe next.

2.1 Pastry Node State

Each Pastry node maintains a *routing table*, a *neighborhood set* and a *leaf set*. We begin with a description of the routing table. A node's routing table, R, is organized into $\lceil log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. The $2^b - 1$ entries at row n of the routing table each refer to a node whose nodeId shares the present node's nodeId in the first n digits,



NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	**1**	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	**2**	10-3-23302
102-0-0230	102-1-1302	102-2-2302	**3**
1023-0-322	1023-1-000	1023-2-121	**3**
10233-0-01	**1**	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Fig. 1. State of a hypothetical Pastry node with nodeId 10233102, $b = 2$, and $l = 8$. All numbers are in base 4. The top row of the routing table is row zero. The shaded cell in each row of the routing table shows the corresponding digit of the present node's nodeId. The nodeIds in each entry have been split to show the *common prefix with 10233102 - next digit - rest of nodeId*. The associated IP addresses are not shown.

but whose $n+1$th digit has one of the $2^b - 1$ possible values other than the $n+1$th digit in the present node's id.

Each entry in the routing table contains the IP address of one of potentially many nodes whose nodeId have the appropriate prefix; in practice, a node is chosen that is close to the present node, according to the proximity metric. We will show in Section 2.5 that this choice provides good locality properties. If no node is known with a suitable nodeId, then the routing table entry is left empty. The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, on average, only $\lceil log_{2^b} N \rceil$ rows are populated in the routing table.

The choice of b involves a trade-off between the size of the populated portion of the routing table (approximately $\lceil log_{2^b} N \rceil \times (2^b - 1)$ entries) and the maximum number of hops required to route between any pair of nodes ($\lceil log_{2^b} N \rceil$). With a value of $b = 4$ and 10^6 nodes, a routing table contains on average 75 entries and the expected number of routing hops is 5, whilst with 10^9 nodes, the routing table contains on average 105 entries, and the expected number of routing hops in 7.

The neighborhood set M contains the nodeIds and IP addresses of the $|M|$ nodes that are closest (according the proximity metric) to the local node. The neighborhood set is not normally used in routing messages; it is useful in maintaining locality properties, as discussed in Section 2.5. The leaf set L is the set of nodes with the $|L|/2$ numerically closest larger nodeIds, and the $|L|/2$ nodes with numerically closest smaller nodeIds, relative to the present node's nodeId. The leaf set is used during the message routing, as described below. Typical values for $|L|$ and $|M|$ are 2^b or 2×2^b.

How the various tables of a Pastry node are initialized and maintained is the subject of Section 2.4. Figure 1 depicts the state of a hypothetical Pastry node with the nodeId 10233102 (base 4), in a system that uses 16 bit nodeIds and a value of $b = 2$.

2.2 Routing

The Pastry routing procedure is shown in pseudo code form in Table 1. The procedure is executed whenever a message with key D arrives at a node with nodeId A. We begin by defining some notation.
R_l^i: the entry in the routing table R at column $i, 0 \leq i < 2^b$ and row $l, 0 \leq l < \lfloor 128/b \rfloor$.
L_i: the i-th closest nodeId in the leaf set L, $-\lfloor |L|/2 \rfloor \leq i \leq \lfloor |L|/2 \rfloor$, where negative/positive indices indicate nodeIds smaller/larger than the present nodeId, respectively.
D_l: the value of the l's digit in the key D.
$shl(A, B)$: the length of the prefix shared among A and B, in digits.

Table 1. Pseudo code for Pastry core routing algorithm.

```
(1)  if (L_{-⌊|L|/2⌋} ≤ D ≤ L_{⌊|L|/2⌋}) {
(2)      // D is within range of our leaf set
(3)      forward to L_i, s.th. |D − L_i| is minimal;
(4)  } else {
(5)      // use the routing table
(6)      Let l = shl(D, A);
(7)      if (R_l^{D_l} ≠ null) {
(8)          forward to R_l^{D_l};
(9)      }
(10)     else {
(11)         // rare case
(12)         forward to T ∈ L ∪ R ∪ M, s.th.
(13)             shl(T, D) ≥ l,
(14)             |T − D| < |A − D|
(15)     }
(16) }
```

Given a message, the node first checks to see if the key falls within the range of nodeIds covered by its leaf set (line 1). If so, the message is forwarded directly to the destination node, namely the node in the leaf set whose nodeId is closest to the key (possibly the present node) (line 3).

If the key is not covered by the leaf set, then the routing table is used and the message is forwarded to a node that shares a common prefix with the key by at least one more digit (lines 6–8). In certain cases, it is possible that the appropriate entry in the routing table is empty or the associated node is not reachable (line 11–14), in which case the message is forwarded to a node that shares a prefix with the key at least as long as the local node,

and is numerically closer to the key than the present node's id. Such a node must be in the leaf set unless the message has already arrived at the node with numerically closest nodeId. And, unless $\lfloor |L|/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously, at least one of those nodes must be live.

This simple routing procedure always converges, because each step takes the message to a node that either (1) shares a longer prefix with the key than the local node, or (2) shares as long a prefix with, but is numerically closer to the key than the local node.

Routing performance. It can be shown that the expected number of routing steps is $\lceil log_{2^b} N \rceil$ steps, assuming accurate routing tables and no recent node failures. Briefly, consider the three cases in the routing procedure. If a message is forwarded using the routing table (lines 6–8), then the set of nodes whose ids have a longer prefix match with the key is reduced by a factor of 2^b in each step, which means the destination is reached in $\lceil log_{2^b} N \rceil$ steps. If the key is within range of the leaf set (lines 2–3), then the destination node is at most one hop away.

The third case arises when the key is not covered by the leaf set (i.e., it is still more than one hop away from the destination), but there is no routing table entry. Assuming accurate routing tables and no recent node failures, this means that a node with the appropriate prefix does not exist (lines 11–14). The likelihood of this case, given the uniform distribution of nodeIds, depends on $|L|$. Analysis shows that with $|L| = 2^b$ and $|L| = 2 \times 2^b$, the probability that this case arises during a given message transmission is less than .02 and 0.006, respectively. When it happens, no more than one additional routing step results with high probability.

In the event of many simultaneous node failures, the number of routing steps required may be at worst linear in N, while the nodes are updating their state. This is a loose upper bound; in practice, routing performance degrades gradually with the number of recent node failures, as we will show experimentally in Section 3.1. Eventual message delivery is guaranteed unless $\lfloor |L|/2 \rfloor$ nodes with consecutive nodeIds fail simultaneously. Due to the expected diversity of nodes with adjacent nodeIds, and with a reasonable choice for $|L|$ (e.g. 2^b), the probability of such a failure can be made very low.

2.3 Pastry API

Next, we briefly outline Pastry's application programming interface (API). The presented API is slightly simplified for clarity. Pastry exports the following operations:

nodeId = pastryInit(Credentials, Application) causes the local node to join an existing Pastry network (or start a new one), initialize all relevant state, and return the local node's nodeId. The application-specific credentials contain information needed to authenticate the local node. The application argument is a handle to the application object that provides the Pastry node with the procedures to invoke when certain events happen, e.g., a message arrival.

route(msg,key) causes Pastry to route the given message to the node with nodeId numerically closest to the key, among all live Pastry nodes.

Applications layered on top of Pastry must export the following operations:

deliver(msg,key) called by Pastry when a message is received and the local node's nodeId is numerically closest to key, among all live nodes.

forward(msg,key,nextId) called by Pastry just before a message is forwarded to the node with nodeId = nextId. The application may change the contents of the message or the value of nextId. Setting the nextId to NULL terminates the message at the local node.

newLeafs(leafSet) called by Pastry whenever there is a change in the local node's leaf set. This provides the application with an opportunity to adjust application-specific invariants based on the leaf set.

Several applications have been built on top of Pastry using this simple API, including PAST [11,21] and SCRIBE [22], and several applications are under development.

2.4 Self-Organization and Adaptation

In this section, we describe Pastry's protocols for handling the arrival and departure of nodes in the Pastry network. We begin with the arrival of a new node that joins the system. Aspects of this process pertaining to the locality properties of the routing tables are discussed in Section 2.5.

Node arrival. When a new node arrives, it needs to initialize its state tables, and then inform other nodes of its presence. We assume the new node knows initially about a nearby Pastry node A, according to the proximity metric, that is already part of the system. Such a node can be located automatically, for instance, using "expanding ring" IP multicast, or be obtained by the system administrator through outside channels.

Let us assume the new node's nodeId is X. (The assignment of nodeIds is application-specific; typically it is computed as the SHA-1 hash of its IP address or its public key). Node X then asks A to route a special "join" message with the key equal to X. Like any message, Pastry routes the join message to the existing node Z whose id is numerically closest to X.

In response to receiving the "join" request, nodes A, Z, and all nodes encountered on the path from A to Z send their state tables to X. The new node X inspects this information, may request state from additional nodes, and then initializes its own state tables, using a procedure describe below. Finally, X informs any nodes that need to be aware of its arrival. This procedure ensures that X initializes its state with appropriate values, and that the state in all other affected nodes is updated.

Since node A is assumed to be in proximity to the new node X, A's neighborhood set to initialize X's neighborhood set. Moreover, Z has the closest existing nodeId to X, thus its leaf set is the basis for X's leaf set. Next, we consider the routing table, starting at row zero. We consider the most general case, where the nodeIds of A and X share no common prefix. Let A_i denote node A's row of the routing table at level i. Note that the entries in row zero of the routing table are independent of a node's nodeId. Thus, A_0 contains appropriate values for X_0. Other levels of A's routing table are of no use to X, since A's and X's ids share no common prefix.

However, appropriate values for X_1 can be taken from B_1, where B is the first node encountered along the route from A to Z. To see this, observe that entries in B_1 and X_1 share the same prefix, because X and B have the same first digit in their nodeId. Similarly, X obtains appropriate entries for X_2 from node C, the next node encountered along the route from A to Z, and so on.

Finally, X transmits a copy of its resulting state to each of the nodes found in its neighborhood set, leaf set, and routing table. Those nodes in turn update their own state based on the information received. One can show that at this stage, the new node X is able to route and receive messages, and participate in the Pastry network. The total cost for a node join, in terms of the number of messages exchanged, is $O(log_{2^b} N)$. The constant is about 3×2^b.

Pastry uses an optimistic approach to controlling concurrent node arrivals and departures. Since the arrival/departure of a node affects only a small number of existing nodes in the system, contention is rare and an optimistic approach is appropriate. Briefly, whenever a node A provides state information to a node B, it attaches a timestamp to the message. B adjusts its own state based on this information and eventually sends an update message to A (e.g., notifying A of its arrival). B attaches the original timestamp, which allows A to check if its state has since changed. In the event that its state has changed, it responds with its updated state and B restarts its operation.

Node departure. Nodes in the Pastry network may fail or depart without warning. In this section, we discuss how the Pastry network handles such node departures. A Pastry node is considered failed when its immediate neighbors in the nodeId space can no longer communicate with the node.

To replace a failed node in the leaf set, its neighbor in the nodeId space contacts the live node with the largest index on the side of the failed node, and asks that node for its leaf table. For instance, if L_i failed for $\lfloor |L|/2 \rfloor < i < 0$, it requests the leaf set from $L_{-\lfloor |L|/2 \rfloor}$. Let the received leaf set be L'. This set partly overlaps the present node's leaf set L, and it contains nodes with nearby ids not presently in L. Among these new nodes, the appropriate one is then chosen to insert into L, verifying that the node is actually alive by contacting it. This procedure guarantees that each node lazily repairs its leaf set unless $\lfloor |L|/2 \rfloor$ nodes with adjacent nodeIds have failed simultaneously. Due to the diversity of nodes with adjacent nodeIds, such a failure is very unlikely even for modest values of $|L|$.

The failure of a node that appears in the routing table of another node is detected when that node attempts to contact the failed node and there is no response. As explained in Section 2.2, this event does not normally delay the routing of a message, since the message can be forwarded to another node. However, a replacement entry must be found to preserve the integrity of the routing table.

To repair a failed routing table entry R_l^d, a node contacts first the node referred to by another entry $R_l^i, i \neq d$ of the same row, and asks for that node's entry for R_l^d. In the event that none of the entries in row l have a pointer to a live node with the appropriate prefix, the node next contacts an entry $R_{l+1}^i, i \neq d$, thereby casting a wider net. This procedure is highly likely to eventually find an appropriate node if one exists.

The neighborhood set is not normally used in the routing of messages, yet it is important to keep it current, because the set plays an important role in exchanging information

about nearby nodes. For this purpose, a node attempts to contact each member of the neighborhood set periodically to see if it is still alive. If a member is not responding, the node asks other members for their neighborhood tables, checks the distance of each of the newly discovered nodes, and updates it own neighborhood set accordingly.

Experimental results in Section 3.2 demonstrate Pastry's effectiveness in repairing the node state in the presences of node failures, and quantify the cost of this repair in terms of the number of messages exchanged.

2.5 Locality

In the previous sections, we discussed Pastry's basic routing properties and discussed its performance in terms of the expected number of routing hops and the number of messages exchanged as part of a node join operation. This section focuses on another aspect of Pastry's routing performance, namely its properties with respect to locality. We will show that the route chosen for a message is likely to be "good" with respect to the proximity metric.

Pastry's notion of network proximity is based on a scalar proximity metric, such as the number of IP routing hops or geographic distance. It is assumed that the application provides a function that allows each Pastry node to determine the "distance" of a node with a given IP address to itself. A node with a lower distance value is assumed to be more desirable. An application is expected to implements this function depending on its choice of a proximity metric, using network services like traceroute or Internet subnet maps, and appropriate caching and approximation techniques to minimize overhead.

Throughout this discussion, we assume that the proximity space defined by the chosen proximity metric is Euclidean; that is, the triangulation inequality holds for distances among Pastry nodes. This assumption does not hold in practice for some proximity metrics, such as the number of IP routing hops in the Internet. If the triangulation inequality does not hold, Pastry's basic routing is not affected; however, the locality properties of Pastry routes may suffer. Quantifying the impact of such deviations is the subject of ongoing work.

We begin by describing how the previously described procedure for node arrival is augmented with a heuristic that ensures that routing table entries are chosen to provide good locality properties.

Locality in the routing table. In Section 2.4, we described how a newly joining node initializes its routing table. Recall that a newly joining node X asks an existing node A to route a join message using X as the key. The message follows a paths through nodes A, B, etc., and eventually reaches node Z, which is the live node with the numerically closest nodeId to X. Node X initialized its routing table by obtaining the i-th row of its routing table from the i-th node encountered along the route from A to Z.

The property we wish to maintain is that all routing table entries refer to a node that is near the present node, according to the proximity metric, among all live nodes with a prefix appropriate for the entry. Let us assume that this property holds prior to node X's joining the system, and show how we can maintains the property as node X joins.

First, we require that node A is near X, according to the proximity metric. Since the entries in row zero of A's routing table are close to A, A is close to X, and we assume

that the triangulation inequality holds in the proximity space, it follows that the entries are relatively near A. Therefore, the desired property is preserved. Likewise, obtaining X's neighborhood set from A is appropriate.

Let us next consider row one of X's routing table, which is obtained from node B. The entries in this row are near B, however, it is not clear how close B is to X. Intuitively, it would appear that for X to take row one of its routing table from node B does not preserve the desired property, since the entries are close to B, but not necessarily to X. In reality, the entries tend to be reasonably close to X. Recall that the entries in each successive row are chosen from an exponentially decreasing set size. Therefore, the expected distance from B to its row one entries (B_1) is much larger than the expected distance traveled from node A to B. As a result, B_1 is a reasonable choice for X_1. This same argument applies for each successive level and routing step, as depicted in Figure 2.

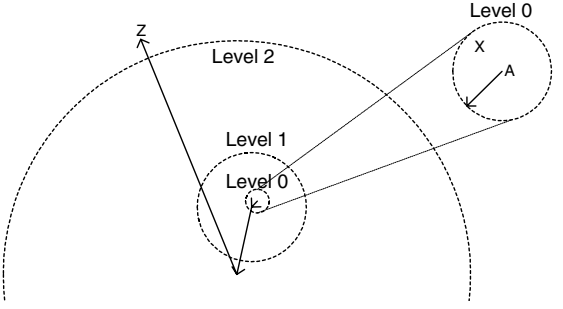


Fig. 2. Routing step distance versus distance of the representatives at each level (based on experimental data). The circles around the n-th node along the route from A to Z indicate the average distance of the node's representatives at level n. Note that X lies within each circle.

After X has initialized its state in this fashion, its routing table and neighborhood set approximate the desired locality property. However, the quality of this approximation must be improved to avoid cascading errors that could eventually lead to poor route locality. For this purpose, there is a second stage in which X requests the state from each of the nodes in its routing table and neighborhood set. It then compares the distance of corresponding entries found in those nodes' routing tables and neighborhood sets, respectively, and updates its own state with any closer nodes it finds. The neighborhood set contributes valuable information in this process, because it maintains and propagates information about nearby nodes regardless of their nodeId prefix.

Intuitively, a look at Figure 2 illuminates why incorporating the state of nodes mentioned in the routing and neighborhood tables from stage one provides good representatives for X. The circles show the average distance of the entry from each node along the route, corresponding to the rows in the routing table. Observe that X lies within each circle, albeit off-center. In the second stage, X obtains the state from the entries discovered in stage one, which are located at an average distance equal to the perimeter of each respective circle. These states must include entries that are appropriate for X, but were not discovered by X in stage one, due to its off-center location.

Experimental results in Section 3.2 show that this procedure maintains the locality property in the routing table and neighborhood sets with high fidelity. Next, we discuss how the locality in Pastry's routing tables affects Pastry routes.

Route locality. The entries in the routing table of each Pastry node are chosen to be close to the present node, according to the proximity metric, among all nodes with the desired nodeId prefix. As a result, in each routing step, a message is forwarded to a relatively close node with a nodeId that shares a longer common prefix or is numerically closer to the key than the local node. That is, each step moves the message closer to the destination in the nodeId space, while traveling the least possible distance in the proximity space.

Since only local information is used, Pastry minimizes the distance of the next routing step with no sense of global direction. This procedure clearly does not guarantee that the shortest path from source to destination is chosen; however, it does give rise to relatively good routes. Two facts are relevant to this statement. First, given a message was routed from node A to node B at distance d from A, the message cannot subsequently be routed to a node with a distance of less than d from A. This follows directly from the routing procedure, assuming accurate routing tables.

Second, the expected distance traveled by a messages during each successive routing step is exponentially increasing. To see this, observe that an entry in the routing table in row l is chosen from a set of nodes of size $N/2^{bl}$. That is, the entries in successive rows are chosen from an exponentially decreasing number of nodes. Given the random and uniform distribution of nodeIds in the network, this means that the expected distance of the closest entry in each successive row is exponentially increasing.

Jointly, these two facts imply that although it cannot be guaranteed that the distance of a message from its source increases monotonically at each step, a message tends to make larger and larger strides with no possibility of returning to a node within d_i of any node i encountered on the route, where d_i is the distance of the routing step taken away from node i. Therefore, the message has nowhere to go but towards its destination. Figure 3 illustrates this effect.

Locating the nearest among k nodes. Some peer-to-peer application we have built using Pastry replicate information on the k Pastry nodes with the numerically closest nodeIds to a key in the Pastry nodeId space. PAST, for instance, replicates files in this way to ensure high availability despite node failures. Pastry naturally routes a message with the given key to the live node with the numerically closest nodeId, thus ensuring that the message reaches one of the k nodes as long as at least one of them is live.

Moreover, Pastry's locality properties make it likely that, along the route from a client to the numerically closest node, the message first reaches a node near the client, in terms of the proximity metric, among the k numerically closest nodes. This is useful in applications such as PAST, because retrieving a file from a nearby node minimizes client latency and network load. Moreover, observe that due to the random assignment of nodeIds, nodes with adjacent nodeIds are likely to be widely dispersed in the network. Thus, it is important to direct a lookup query towards a node that is located relatively near the client.

Recall that Pastry routes messages towards the node with the nodeId closest to the key, while attempting to travel the smallest possible distance in each step. Therefore,

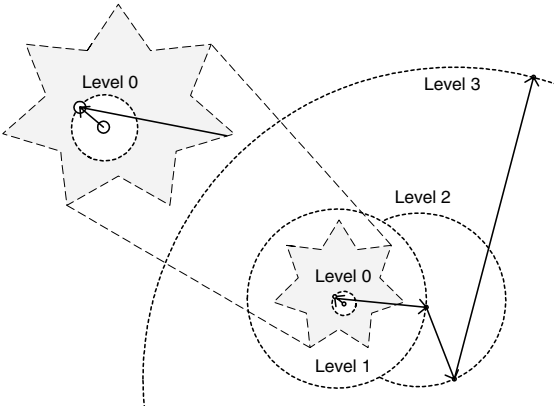


Fig. 3. Sample trajectory of a typical message in the Pastry network, based on experimental data. The message cannot re-enter the circles representing the distance of each of its routing steps away from intermediate nodes. Although the message may partly "turn back" during its initial steps, the exponentially increasing distances traveled in each step cause it to move toward its destination quickly.

among the k numerically closest nodes to a key, a message tends to first reach a node near the client. Of course, this process only approximates routing to the nearest node. Firstly, as discussed above, Pastry makes only local routing decisions, minimizing the distance traveled on the next step with no sense of global direction. Secondly, since Pastry routes primarily based on nodeId prefixes, it may miss nearby nodes with a different prefix than the key. In the worst case, $k/2 - 1$ of the replicas are stored on nodes whose nodeIds differ from the key in their domain at level zero. As a result, Pastry will first route towards the nearest among the $k/2 + 1$ remaining nodes.

Pastry uses a heuristic to overcome the prefix mismatch issue described above. The heuristic is based on estimating the density of nodeIds in the nodeId space using local information. Based on this estimation, the heuristic detects when a message approaches the set of k numerically closest nodes, and then switches to numerically nearest address based routing to locate the nearest replica. Results presented in Section 3.3 show that Pastry is able to locate the nearest node in over 75%, and one of the two nearest nodes in over 91% of all queries.

2.6 Arbitrary Node Failures and Network Partitions

Throughout this paper, it is assumed that Pastry nodes fail silently. Here, we briefly discuss how a Pastry network could deal with arbitrary nodes failures, where a failed node continues to be responsive, but behaves incorrectly or even maliciously. The Pastry routing scheme as described so far is deterministic. Thus, it is vulnerable to malicious or failed nodes along the route that accept messages but do not correctly forward them. Repeated queries could thus fail each time, since they normally take the same route.

In applications where arbitrary node failures must be tolerated, the routing can be randomized. Recall that in order to avoid routing loops, a message must always be

forwarded to a node that shares a longer prefix with the destination, or shares the same prefix length as the current node but is numerically closer in the nodeId space than the current node. However, the choice among multiple nodes that satisfy this criterion can be made randomly. In practice, the probability distribution should be biased towards the best choice to ensure low average route delay. In the event of a malicious or failed node along the path, the query may have to be repeated several times by the client, until a route is chosen that avoids the bad node. Furthermore, the protocols for node join and node failure can be extended to tolerate misbehaving nodes. The details are beyond the scope of this paper.

Another challenge are IP routing anomalies in the Internet that cause IP hosts to be unreachable from certain IP hosts but not others. The Pastry routing is tolerant of such anomalies; Pastry nodes are considered live and remain reachable in the overlay network as long as they are able to communication with their immediate neighbors in the nodeId space. However, Pastry's self-organization protocol may cause the creation of multiple, isolated Pastry overlay networks during periods of IP routing failures. Because Pastry relies almost exclusively on information exchange within the overlay network to self-organize, such isolated overlays may persist after full IP connectivity resumes.

One solution to this problem involves the use of IP multicast. Pastry nodes can periodically perform an expanding ring multicast search for other Pastry nodes in their vicinity. If isolated Pastry overlays exists, they will be discovered eventually, and reintegrated. To minimize the cost, this procedure can be performed randomly and infrequently by Pastry nodes, only within a limited range of IP routing hops from the node, and only if no search was performed by another nearby Pastry node recently. As an added benefit, the results of this search can also be used to improve the quality of the routing tables.

3 Experimental Results

In this section, we present experimental results obtained with a prototype implementation of Pastry. The Pastry node software was implemented in Java. To be able to perform experiments with large networks of Pastry nodes, we also implemented a network emulation environment, permitting experiments with up to 100,000 Pastry nodes.

All experiments were performed on a quad-processor Compaq AlphaServer ES40 (500MHz 21264 Alpha CPUs) with 6GBytes of main memory, running True64 UNIX, version 4.0F. The Pastry node software was implemented in Java and executed using Compaq's Java 2 SDK, version 1.2.2-6 and the Compaq FastVM, version 1.2.2-4.

In all experiments reported in this paper, the Pastry nodes were configured to run in a single Java VM. This is largely transparent to the Pastry implementation—the Java runtime system automatically reduces communication among the Pastry nodes to local object invocations.

The emulated network environment maintains distance information between the Pastry nodes. Each Pastry node is assigned a location in a plane; coordinates in the plane are randomly assigned in the range $[0, 1000]$. Nodes in the Internet are not uniformly distributed in a Euclidean space; instead, there is a strong clustering of nodes and the triangulation inequality doesn't always hold. We are currently performing emulations based on a more realistic network topology model taken from [26]. Early results indicate

that overall, Pastry's locality related routing properties are not significantly affected by this change.

A number of Pastry properties are evaluated experimentally. The first set of results demonstrates the basic performance of Pastry routing. The routing tables created within the Pastry nodes are evaluated in Section 3.2. In Section 3.3 we evaluate Pastry's ability to route to the nearest among the k numerically closest nodes to a key. Finally, in 3.4 the properties of Pastry under node failures are considered.

3.1 Routing Performance

The first experiment shows the number of routing hops as a function of the size of the Pastry network. We vary the number of Pastry nodes from 1,000 to 100,000 in a network where $b = 4$, $|L| = 16$, $|M| = 32$. In each of 200,000 trials, two Pastry nodes are selected at random and a message is routed between the pair using Pastry.

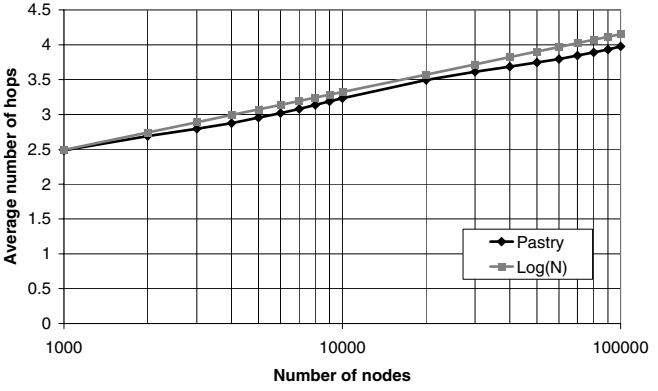


Fig. 4. Average number of routing hops versus number of Pastry nodes, $b = 4$, $|L| = 16$, $|M| = 32$ and 200,000 lookups.

Figure 4 show the average number of routing hops taken, as a function of the network size. "Log N" shows the value $log_{2^b} N$ and is included for comparison. ($\lceil log_{2^b} N \rceil$ is the expected maximum number of hops required to route in a network containing N nodes). The results show that the number of route hops scale with the size of the network as predicted.

Figure 5 shows the distribution of the number of routing hops taken, for a network size of 100,000 nodes, in the same experiment. The results show that the maximum route length is ($\lceil log_{2^b} N \rceil$ ($\lceil log_{2^b} 100,000 \rceil = 5$), as expected.

The second experiment evaluated the locality properties of Pastry routes. It compares the relative distance a message travels using Pastry, according to the proximity metric, with that of a fictitious routing scheme that maintains complete routing tables. The distance traveled is the sum of the distances between consecutive nodes encountered along the route in the emulated network. For the fictitious routing scheme, the distance

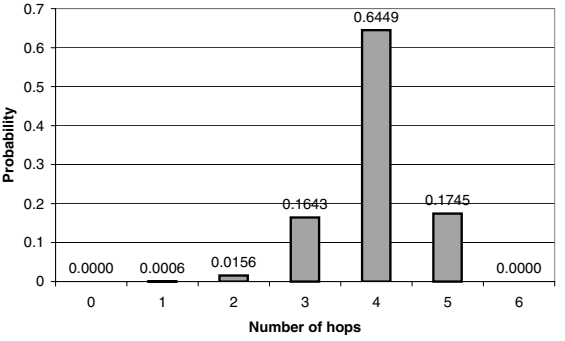


Fig. 5. Probability versus number of routing hops, $b = 4$, $|L| = 16$, $|M| = 32$, $N = 100,000$ and 200,000 lookups.

traveled is simply the distance between the source and the destination node. The results are normalized to the distance traveled in the fictitious routing scheme. The goal of this experiment is to quantify the cost, in terms of distance traveled in the proximity space, of maintaining only small routing tables in Pastry.

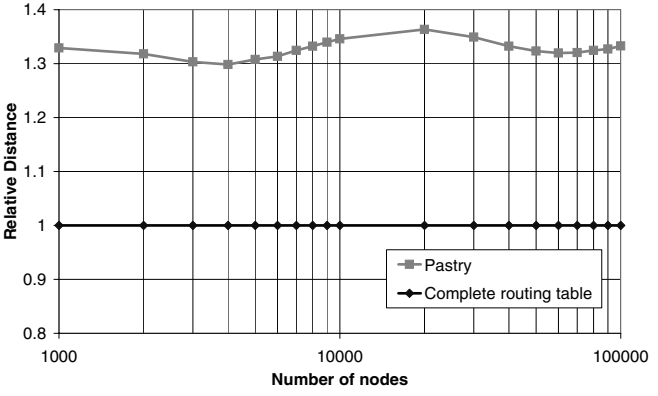


Fig. 6. Route distance versus number of Pastry nodes, $b = 4$, $|L| = 16$, $|M| = 32$, and 200,000 lookups.

The number of nodes varies between 1,000 and 100,000, $b = 4$, $|L| = 16$, $|M| = 32$. 200,000 pairs of Pastry nodes are selected and a message is routed between each pair. Figure 6 shows the results for Pastry and the fictitious scheme (labeled "Complete routing tables"). The results show that the Pastry routes are only approximately 30% to 40% longer. Considering that the routing tables in Pastry contain only approximately $\lceil log_{2^b} N \rceil \times (2^b - 1)$ entries, this result is quite good. For 100,000 nodes the Pastry routing tables contain approximately 75 entries, compared to 99,999 in the case of complete routing tables.

We also determined the routing throughput, in messages per second, of a Pastry node. Our unoptimized Java implementation handled over 3,000 messages per second. This indicates that the routing procedure is very lightweight.

3.2 Maintaining the Network

Figure 7 shows the quality of the routing tables with respect to the locality property, and how the extent of information exchange during a node join operation affects the quality of the resulting routing tables vis-à-vis locality. In this experiment, 5,000 nodes join the Pastry network one by one. After all nodes joined, the routing tables were examined. The parameters are $b = 4, |L| = 16, |M| = 32$.

Three options were used to gather information when a node joins. "SL" is a hypothetical method where the joining node considers only the appropriate row from each node along the route from itself to the node with the closest existing nodeId (see Section 2.4). With "WT", the joining node fetches the entire state of each node along the path, but does not fetch state from the resulting entries. This is equivalent to omitting the second stage. "WTF" is the actual method used in Pastry, where state is fetched from each node that appears in the tables after the first stage.

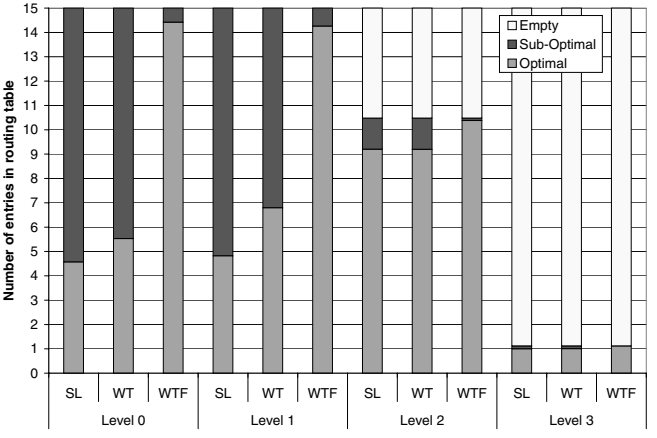


Fig. 7. Quality of routing tables (locality), $b = 4$, $|L| = 16$, $|M| = 32$ and 5,000 nodes.

The results are shown in Figure 7. For levels 0 to 3, we show the quality of the routing table entries with each method. With 5,000 nodes and $b = 4$, levels 2 and 3 are not fully populated, which explains the missing entries shown. "Optimal" means that the best (i.e., closest according to the proximity metric) node appeared in a routing table entry, "sub-optimal" means that an entry was not the closest or was missing.

The results show that Pastry's method of node integration ("WTF") is highly effective in initializing the routing tables with good locality. On average, less than 1 entry per level of the routing able is not the best choice. Moreover, the comparison with "SL" and

"WT" shows that less information exchange during the node join operation comes at a dramatic cost in routing table quality with respect to locality.

3.3 Replica Routing

The next experiment examines Pastry's ability to route to one of the k closest nodes to a key, where $k = 5$. In particular, the experiment explores Pastry's ability to locate one of the k nodes near the client. In a Pastry network of 10,000 nodes with $b = 3$ and $|L| = 8$, 100,000 times a Pastry node and a key are chosen randomly, and a message is routed using Pastry from the node using the key. The first of the k numerically closest nodes to the key that is reached along the route is recorded.

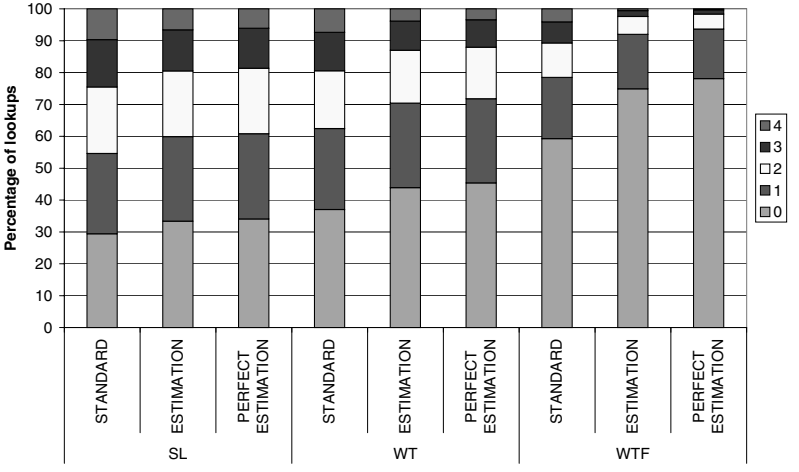


Fig. 8. Number of nodes closer to the client than the node discovered. ($b = 3$, $|L| = 8$, $|M| = 16$, 10,000 nodes and 100,000 message routes).

Figure 8 shows the percentage of lookups that reached the closest node, according to the proximity metric (0 closer nodes), the second closest node (1 closer node), and so forth. Results are shown for the three different protocols for initializing a new node's state, with ("Estimation") and without ("Standard") the heuristic mentioned in Section 2.5, and for an idealized, optimal version of the heuristic ("Perfect estimation"). Recall that the heuristic estimates the nodeId space coverage of other nodes' leaf sets, using an estimate based on its own leaf sets coverage. The "Perfect estimation" ensures that this estimate of a node's leaf set coverage is correct for every node.

Without the heuristic and the standard node joining protocol (WTF), Pastry is able to locate the closest node 68% of the time, and one of the top two nodes 87% of the time. With the heuristic routing option, this figures increase to 76% and 92%, respectively. The lesser routing table quality resulting from the "SL" and "WT" methods for node joining have a strong negative effect on Pastry's ability to locate nearby nodes, as one

would expect. Also, the heuristic approach is only approximately 2% worse than the best possible results using perfect estimation.

The results show that Pastry is effective in locating a node near the client in the vast majority of cases, and that the use of the heuristic is effective.

3.4 Node Failures

The next experiment explores Pastry's behavior in the presence of node failures. A 5,000 node Pastry network is used with $b = 4$, $|L| = 16$, $|M| = 32$, $k = 5$. Then, 10% (500) randomly selected nodes fail silently. After these failures, a key is chosen at random, and two Pastry nodes are randomly selected. A message is routed from these two nodes to the key, and this is repeated 100,000 times (200,000 lookups total). Initially, the node state repair facilities in Pastry were disabled, which allows us to measure the full impact of the failures on Pastry's routing performance. Next, the node state repair facilities were enabled, and another 200,000 lookups were performed from the same Pastry nodes to the same key.

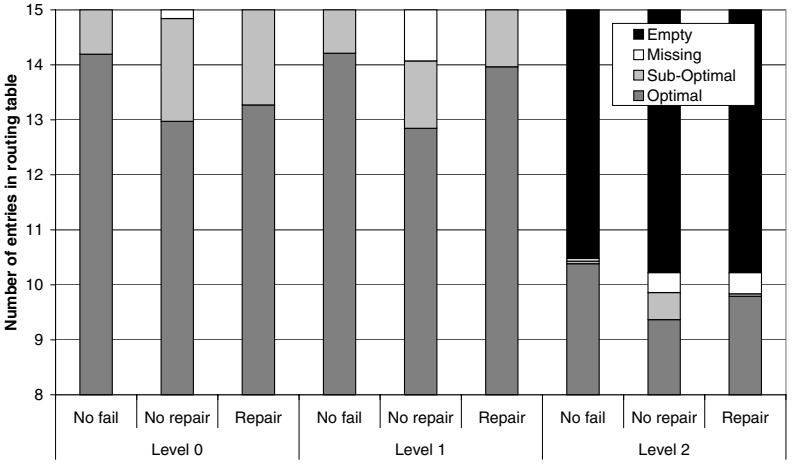


Fig. 9. Quality of routing tables before and after 500 node failures, $b = 4$, $|L| = 16$, $|M| = 32$ and 5,000 starting nodes.

Figure 9 shows the average routing table quality across all nodes for levels 0–2, as measured before the failures, after the failures, and after the repair. Note that in this figure, missing entries are shown separately from sub-optimal entries. Also, recall that Pastry lazily repairs routing tables entries when they are being used. As a result, routing table entries that were not used during the 200,000 lookups are not discovered and therefore not repaired. To isolate the effectiveness of Pastry's repair procedure, we excluded table entries that were never used.

The results show that Pastry recovers all missing table entries, and that the quality of the entries with respect to locality (fraction of optimal entries) approaches that before the

failures. At row zero, the average number of best entries after the repair is approximately one below that prior to the failure. However, although this can't be seen in the figure, our results show that the actual distance between the suboptimal and the optimal entries is very small. This is intuitive, since the average distance of row zero entries is very small. Note that the increase in empty entries at levels 1 and 2 after the failures is due to the reduction in the total number of Pastry nodes, which increases the sparseness of the tables at the upper rows. Thus, this increase does not constitute a reduction in the quality of the tables.

Figure 10 shows the impact of failures and repairs on the route quality. The left bar shows the average number of hops before the failures; the middle bar shows the average number of hops after the failures, and before the tables were repaired. Finally, the right bar shows the average number of hops after the repair. The data shows that without repairs, the stale routing table state causes as significant deterioration of route quality. After the repair, however, the average number of hops is only slightly higher than before the failures.

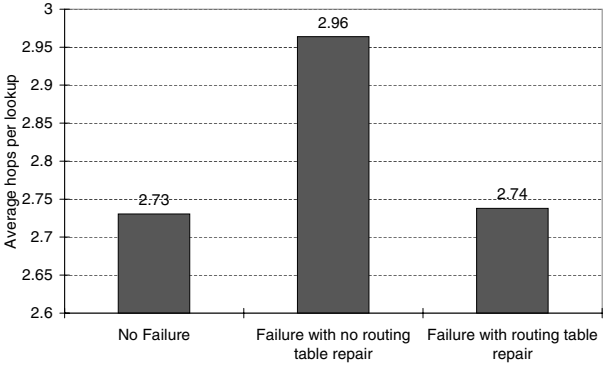


Fig. 10. Number of routing hops versus node failures, $b = 4$, $|L| = 16$, $|M| = 32$, 200,000 lookups and 5,000 nodes with 500 failing.

We also measured the average cost, in messages, for repairing the tables after node failure. In our experiments, a total of 57 remote procedure calls were needed on average per failed node to repair all relevant table entries.

4 Related Work

There are currently several peer-to-peer systems in use, and many more are under development. Among the most prominent are file sharing facilities, such as Gnutella [2] and Freenet [8]. The Napster [1] music exchange service provided much of the original motivation for peer-to-peer systems, but it is not a pure peer-to-peer system because its database is centralized. All three systems are primarily intended for the large-scale sharing of data files; reliable content location is not guaranteed or necessary in this environment. In Gnutella, the use of a broadcast based protocol limits the system's scalability

and incurs a high bandwidth requirement. Both Gnutella and Freenet are not guaranteed to find an existing object.

Pastry, along with Tapestry [27], Chord [24] and CAN [19], represent a second generation of peer-to-peer routing and location schemes that were inspired by the pioneering work of systems like FreeNet and Gnutella. Unlike that earlier work, they guarantee a definite answer to a query in a bounded number of network hops, while retaining the scalability of FreeNet and the self-organizing properties of both FreeNet and Gnutella.

Pastry and Tapestry bear some similarity to the work by Plaxton et al [18] and to routing in the landmark hierarchy [25]. The approach of routing based on address prefixes, which can be viewed as a generalization of hypercube routing, is common to all these schemes. However, neither Plaxton nor the landmark approach are fully self-organizing. Pastry and Tapestry differ in their approach to achieving network locality and to supporting replication, and Pastry appears to be less complex.

The Chord protocol is closely related to both Pastry and Tapestry, but instead of routing towards nodes that share successively longer address prefixes with the destination, Chord forwards messages based on numerical difference with the destination address. Unlike Pastry and Tapestry, Chord makes no explicit effort to achieve good network locality. CAN routes messages in a d-dimensional space, where each node maintains a routing table with $O(d)$ entries and any node can be reached in $O(dN^{1/d})$ routing hops. Unlike Pastry, the routing table does not grow with the network size, but the number of routing hops grows faster than $logN$.

Existing applications built on top of Pastry include PAST [11,21] and SCRIBE [22]. Other peer-to-peer applications that were built on top of a generic routing and location substrate like Pastry are OceanStore [15] (Tapestry) and CFS [9] (Chord). FarSite [5] uses a conventional distributed directory service, but could potentially be built on top of a system like Pastry. Pastry can be seen as an overlay network that provides a self-organizing routing and location service. Another example of an overlay network is the Overcast system [12], which provides reliable single-source multicast streams.

There has been considerable work on routing in general, on hypercube and mesh routing in parallel computers, and more recently on routing in ad hoc networks, for example GRID [17]. In Pastry, we assume an existing infrastructure at the network layer, and the emphasis in on self-organization and the integration of content location and routing. In the interest of scalability, Pastry nodes only use local information, while traditional routing algorithms (like link-state and distance vector methods) globally propagate information about routes to each destination. This global information exchange limits the scalability of these routing algorithms, necessitating a hierarchical routing architecture like the one used in the Internet.

Several prior works consider issues in replicating Web content in the Internet, and selecting the nearest replica relative to a client HTTP query [4,13,14]. Pastry provides a more general infrastructure aimed at a variety of peer-to-peer applications. Another related area is that of naming services, which are largely orthogonal to Pastry's content location and routing. Lampson's Global Naming System (GNS) [16] is an example of a scalable naming system that relies on a hierarchy of name servers that directly corresponds to the structure of the name space. Cheriton and Mann [7] describe another scalable naming service.

Finally, attribute based and intentional naming systems [6,3], as well as directory services [20,23] resolve a set of attributes that describe the properties of an object to the address of an object instance that satisfies the given properties. Thus, these systems support far more powerful queries than Pastry. However, this power comes at the expense of scalability, performance and administrative overhead. Such systems could be potentially built upon Pastry.

5 Conclusion

This paper presents and evaluates Pastry, a generic peer-to-peer content location and routing system based on a self-organizing overlay network of nodes connected via the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliably routes a message to the live node with a nodeId numerically closest to a key. Pastry can be used as a building block in the construction of a variety of peer-to-peer Internet applications like global file sharing, file storage, group communication and naming systems.

Pastry routes to any node in the overlay network in O(log N) steps in the absence of recent node failures, and it maintains routing tables with only O(log N) entries. Moreover, Pastry takes into account locality when routing messages. Results with as many as 100,000 nodes in an emulated network confirm that Pastry is efficient and scales well, that it is self-organizing and can gracefully adapt to node failures, and that it has good locality properties.

Acknowledgments. We thank Miguel Castro and the anonymous reviewers for their useful comments and feedback. Peter Druschel thanks Microsoft Research, Cambridge, UK, and the Massachusetts Institute of Technology for their support during his visits in Fall 2000 and Spring 2001, respectively, and Compaq for donating equipment used in this work.

References

1. Napster. http://www.napster.com/.
2. The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.
3. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. SOSP'99*, Kiawah Island, SC, Dec. 1999.
4. Y. Amir, A. Peterson, and D. Shaw. Seamlessly selecting the best copy from Internet-wide replicated web servers. In *Proc. 12th Symposium on Distributed Computing*, Andros, Greece, Sept. 1998.
5. W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. SIGMETRICS'2000*, Santa Clara, CA, 2000.
6. M. Bowman, L. L. Peterson, and A. Yeatts. Univers: An attribute-based name server. *Software — Practice and Experience*, 20(4):403–424, Apr. 1990.
7. D. R. Cheriton and T. P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Trans. Comput. Syst.*, 7(2):147–183, May 1989.

8. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.
9. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.
10. R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000.
11. P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, Schloss Elmau, Germany, May 2001.
12. J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. OSDI 2000*, San Diego, CA, 2000.
13. J. Kangasharju, J. W. Roberts, and K. W. Ross. Performance evaluation of redirection schemes in content distribution networks. In *Proc. 4th Web Caching Workshop*, San Diego, CA, Mar. 1999.
14. J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proc. IEEE Infocom 2000*, Tel Aviv, Israel, Mar. 2000.
15. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, Cambridge, MA, November 2000.
16. B. Lampson. Designing a global name service. In *Proc. Fifth Symposium on the Principles of Distributed Computing*, pages 1–10, Minaki, Canada, Aug. 1986.
17. J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographical ad hoc routing. In *Proc. of ACM MOBICOM 2000*, Boston, MA, 2000.
18. C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.
19. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.
20. J. Reynolds. RFC 1309: Technical overview of directory services using the X.500 protocol, Mar. 1992.
21. A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP'01*, Banff, Canada, Oct. 2001.
22. A. Rowstron, A.-M. Kermarrec, P. Druschel, and M. Castro. Scribe: The design of a large-scale event notification infrastructure. Submitted for publication. June 2001. http://www.research.microsoft.com/ antr/SCRIBE/.
23. M. A. Sheldon, A. Duda, R. Weiss, and D. K. Gifford. Discover: A resource discovery system based on content routing. In *Proc. 3rd International World Wide Web Conference*, Darmstadt, Germany, 1995.
24. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, San Diego, CA, Aug. 2001.
25. P. F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *SIGCOMM'88*, Stanford, CA, 1988.
26. E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM'96*, San Francisco, CA, 1996.
27. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

Providing QoS Customization in Distributed Object Systems*

Jun He[1], Matti A. Hiltunen[2], Mohan Rajagopalan[1], and Richard D. Schlichting[2]

[1] Department of Computer Science, The University of Arizona, Tucson, AZ 85712
[2] AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932

Abstract. Applications built on networked collections of computers are increasingly using distributed object platforms such as CORBA, Java RMI, and DCOM to standardize object interactions. With this increased use comes the increased need for enhanced Quality of Service (QoS) attributes related to fault tolerance, security, and timeliness. This paper describes an architecture called CQoS (Configurable QoS) for implementing such enhancements in a transparent, highly customizable, and portable manner. CQoS consists of two parts: application- and platform-dependent interceptors and generic QoS components. The generic QoS components are implemented using Cactus, a system for building highly configurable protocols and services in distributed systems. The CQoS architecture and the interfaces between the different components are described, together with implementations of QoS attributes using Cactus and interceptors for CORBA and Java RMI. Experimental results are given for a test application executing on a Linux cluster using Cactus/J, the Java implementation of Cactus. Compared with other approaches, CQoS emphasizes portability across different distributed object platforms, while the use of Cactus allows custom combinations of fault-tolerance, security and timeliness attributes to be realized on a per-object basis in a straightforward way.

1 Introduction

Middleware platforms such as CORBA [23], Java RMI [29], and DCOM [2] provide high-level programming abstractions that facilitate distributed object computing, but lack a unified framework for supporting Quality of Service (QoS) guarantees related to fault tolerance, security, and timeliness. The lack of uniformity is apparent in two separate ways, both equally important. First, most existing standardization and research efforts in this area focus on providing a single QoS attribute such as fault tolerance (e.g., [4,6,15,22,24]) or security (e.g., [1]) rather than combinations of attributes. While useful, applications in areas such as financial services and multimedia often need multiple types of guarantees, as well as the ability to control performance and functionality tradeoffs between the different attributes. Second, most efforts provide point solutions for only a single middleware platform rather than a framework that can be used uniformly across different platforms. While sufficient in some cases, it limits applicability and requires

* This work supported in part by the Defense Advanced Research Projects Agency under grant N66001-97-C-8518, and by the National Science Foundation under grant ANI-9979438.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 351–372, 2001.

unnecessary reimplementation of essentially similar techniques. Indeed, many of the basic techniques for implementing various QoS attributes can be used with minor changes on any platform supporting a request/reply interaction paradigm.

To address these issues, we have developed CQoS, a platform-independent QoS architecture for distributed object computing. CQoS consists of *CQoS interceptors* and configurable *CQoS service components*. The service components can be customized to provide the desired combinations of attributes, while the interceptors are used to insert CQoS transparently between the application and middleware platform on both the client and server hosts. The service component is implemented using Java version of Cactus, a system that supports construction of highly-configurable network protocols and services [10,12]. CQoS is designed to be easily portable, and in particular, to allow QoS attributes to be implemented in a way that can be used across different middleware platforms. To do this, the interceptor is used to abstract away middleware and application-specific details, providing a standard interface for implementing enhanced functionality.

The primary goal of this paper is to present CQoS as a single unified framework for providing multiple types of QoS attributes across multiple middleware platforms. We do this by first describing the software architecture and then giving examples of how customized fault-tolerance, timeliness, and security attributes can be realized in a platform-independent manner. The mapping of the architecture to CORBA and Java RMI is then presented, along with experimental results using Visibroker 4.1 and JDK 1.3 on Linux. Finally, we evaluate our approach in the context of related work. While issues related to providing enhanced QoS in middleware have been explored elsewhere (e.g., [19,33]), no other approach offers the same combination of support for multiple QoS attributes, platform independence, and the ability to make fine-grain object-specific customizations as does the architecture described here.

2 Software Architecture

2.1 Overview

The CQoS architecture allows the QoS attributes of a distributed object system to be customized transparently to client and server applications. Figure 1 provides a high-level overview of the architecture. In our prototype implementation, the middleware platform can be CORBA or Java RMI, but as noted above, the same approach can be used for any platform that supports a request-reply interaction paradigm, including standard remote procedure call (RPC) systems. For example, it would be feasible to intercept HTTP requests and replies, in which case the TCP socket layer would be viewed as the middleware layer. CQoS can be configured separately for each distributed object application in the system, allowing application-specific customization.

QoS customization can be done on different system levels, including below the middleware, as a modification to the middleware, or as a service built on top of the middleware. Each alternative has its relative tradeoffs, including factors such as transparency, effectiveness of the approach, performance overhead, and the ease of implementing and customizing such service enhancements. In the case of CQoS, implementing on top of the middleware layer has a number of advantages, including the ability to use the higher-level primitives provided by the middleware for locating objects and for performing

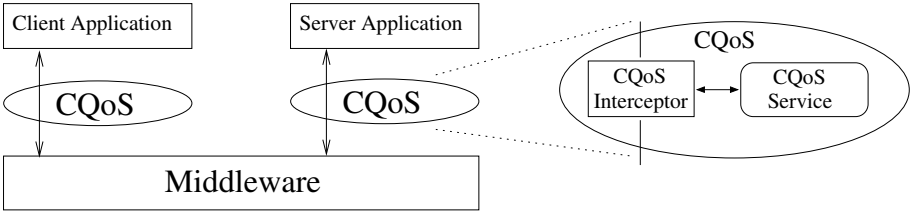


Fig. 1. High-level view of CQoS architecture.

inter-object communication. It also means that CQoS can be inserted transparently to both application objects and middleware, requiring no changes in either. We discuss the relative merits of the different approaches further in section 6.

The key observation enabling this design is that the fundamental techniques for implementing QoS properties such as fault tolerance, security, or timeliness are similar regardless of the specific middleware platform. For example, fault tolerance can be increased by replicating the server and multicasting each method call, independent of whether the middleware is CORBA or Java RMI. The architecture eliminates the need for reimplementing similar techniques for different platforms by allowing QoS properties to be realized in a platform-independent manner.

To achieve portability, CQoS is structured as two components: a middleware- and application-specific *CQoS interceptor* and a generic *CQoS service component* that implements the QoS enhancements. The interceptor provides the service component with the necessary interfaces for manipulating requests and replies to implement the properties. Note that CQoS is needed only to enhance the QoS attributes provided to the application, not to replace or reimplement guarantees that are already provided. For example, if the underlying CORBA ORB provides security services, CQoS can be configured to enhance properties other than security.

The CQoS interceptors for the client and server sides, called the *CQoS stub* and *CQoS skeleton*, respectively, are automatically generated from the server IDL description (e.g., CORBA IDL) using our Cactus IDL compiler. The generic CQoS service components on each side are implemented as Cactus components that are referred to as the *Cactus client* and *Cactus server*, respectively. The rest of this section describes these components in more detail.

2.2 CQoS Interceptors

Client-side interception is based on replacing the conventional stub used by middleware platforms such as CORBA or Java RMI by the CQoS stub (figure 2). When the client invokes a method on this stub, it creates an abstract request object and notifies the Cactus client. The stub then stores the pending request until the call has been completed. Similarly, server-side interception is based on using the CQoS skeleton as a proxy server for the actual server object. This skeleton overwrites the server object binding with the underlying middleware layer. This causes the incoming requests to be forwarded

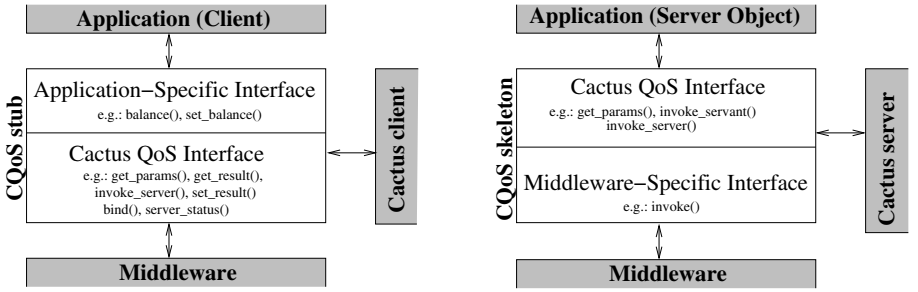


Fig. 2. Structure of CQoS interceptors.

automatically to the CQoS skeleton, which also creates an abstract request object and notifies the Cactus server.

To implement this functionality, CQoS stubs and skeletons provide multiple interfaces for interaction with the application, middleware, and QoS service component. The application interface on the CQoS stub is identical to the original stub and provides operations for each of the server object methods. The middleware interface on the CQoS skeleton provides operations that allow the middleware to pass the request to the CQoS skeleton. The details of CORBA and Java RMI implementations of this interface are discussed in section 4. Finally, the interface for the QoS service component, called the *Cactus QoS* interface, provides methods for the Cactus client and Cactus server to manipulate the requests and connections to servers.

To support request manipulation, the Cactus QoS interface provides an abstract representation of the client request together with appropriate operations. Specifically, the request is represented as a Java class, where the request parameters are represented as a vector of Java objects (`java.lang.Objects`). This interface provides a set of accessor methods to get and set parameters and return values. The implementation of the interface, which is platform specific, takes care of converting the abstract request into a form required by the specific platform. For example, a CORBA implementation that uses DII/DSI converts the abstract request structure into a CORBA request (`org.omg.CORBA.Request`). The request object also provides a field for piggybacking additional parameters onto the request. For example, the Cactus client may include a priority parameter that is used by the Cactus server to determine the order in which requests are to be processed.

The Cactus QoS interface also provides abstract representation of the server objects. Since the implementation of certain attributes such as fault tolerance requires communication with multiple servers, the interface provides operations for creating connections with specific servers (*bind()*), testing the status of a server (*server_status()*), and sending requests to specific servers (*invoke_server()*). The *bind()* operation can also be used to rebind to a failed server after it has recovered. Details related to server names and addressing are hidden by the interface so that the CQoS service component can be independent of the application as well as the middleware platform. In particular, the interface allows the server replicas to be referred to by numbers (1..N) rather than by application

or middleware-specific identifiers. Currently, the *server_status()* operation only indicates if the server is running or failed, but it could be extended to provide information such as the load conditions on the server for load balancing purposes. On the server side, the interface provides operation *invoke_servant()* that the Cactus server can use to actually invoke the method in the server object and the *invoke_server()* operation that the Cactus server can use to communicate with Cactus servers on other replicas. The latter operation is used, for example, to send ordering messages between replicas to implement a consistent total ordering of client invocations.

2.3 CQoS Service Component

QoS attributes are implemented by the Cactus client and Cactus server components. These components are *composite protocols* that are implemented using Cactus, a design and implementation framework for constructing configurable protocols and services [10]. In the following, we briefly introduce Cactus, describe the Cactus client and server components, and outline how these components can be configured to provide custom QoS properties.

Cactus overview. A service or protocol in Cactus is implemented as a composite protocol, with each service property or other functional component implemented as a software module called a *micro-protocol*. A micro-protocol is structured as a collection of event handlers, which are procedure-like segments of code that are executed when a specified event occurs. A customized version of a protocol or service is constructed by selecting the micro-protocols that implement the desired features. The service can also be changed during execution by dynamically altering the configuration of micro-protocols within the composite protocol.

Cactus provides a variety of operations for managing events and event handlers. For example, an operation is provided for binding a handler to an event, with optional static arguments that are passed to the handler on every activation. Events are raised either implicitly by the runtime system or explicitly by a micro-protocol executing an appropriate operation. When an event is raised, all handlers bound to that event are executed. The raise operation also supports a delay parameter, which can be used to implement time-driven execution, and dynamic arguments that are passed to the handlers upon invocation. An event raise may be blocking (synchronous), where the caller is blocked until all the handlers have been executed, or non-blocking (asynchronous), where the caller continues execution concurrently with the handler execution. Other operations are available for unbinding handlers, creating and deleting events, halting event execution, and canceling a delayed event. Cactus also supports data structures shared by micro-protocols in a composite protocol and a message abstraction designed to facilitate development of configurable services.

A number of prototype implementations of Cactus have been completed. These include Cactus/C, a C version that runs on Mach MK 7.3 and Linux; Cactus/C++, a C++ version that runs on Linux and Solaris; and Cactus/J, a Java version that runs on most platforms. The prototypes described here use Cactus/J on Linux.

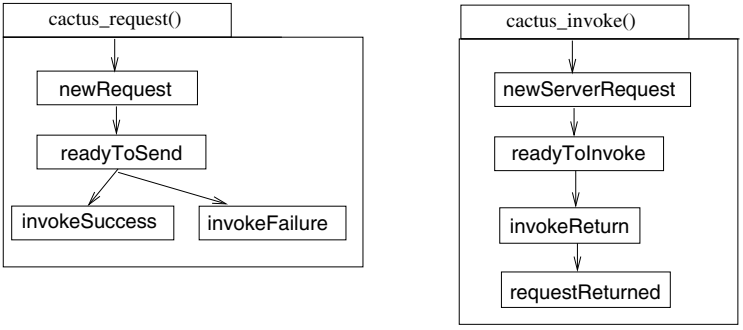


Fig. 3. Cactus events used in CQoS.

Cactus client and server. The Cactus client and server provide a very simple interface for CQoS interceptors. In particular, the client provides a *cactus_request(requestID)* operation that the stub can use to notify it of request arrival, and the server provides an analogous operation *cactus_invoke(requestID)* for the skeleton. Both operations block until the request has been completed. For example, when *cactus_request()* returns, the stub can return the return value from the request structure to the client. The implementations of these operations simply raise the appropriate events newRequest and newServerRequest, respectively, with the actual processing done by various micro-protocols. The above design assumes all client invocations are synchronous, but the implementation could easily be extended to support asynchronous invocations.

Figure 3 illustrates the events used in the Cactus client and server. The arrows between events indicate causal relations between events, that is, an arrow from ev1 to ev2 indicates that some micro-protocol that processes ev1 (i.e., has a handler bound to ev1) raises ev2. Event readyToSend indicates that a request is ready to be sent to the server(s), and invokeSuccess and invokeFailure indicate that an invocation completed successfully or failed. Event readyToInvoke indicates that an invocation is ready to be passed to the server object, invokeReturn that the invocation has returned from the object, and requestReturned that the reply to the request has been sent back to the client side.

Customization. Since CQoS decouples the application from the specification and implementation of the QoS attributes, QoS customization can be done in a variety of ways by end users, system administrators, or application designers. While customization must currently be done using a programming interface, a graphical tool similar to the CactusBuilder [8] could be developed to facilitate the process. Here, we focus on the underlying mechanisms designed to support this customization.

Cactus supports customization both statically at configuration time and dynamically at execution time. With static customization, the desired set of micro-protocols is specified either by modifying the constructor of the composite protocol to start the appropriate micro-protocols or by using a configuration file that is read by the constructor of the composite protocol. With dynamic customization, the set of micro-protocols is deter-

mined after the Cactus protocols start execution and the micro-protocols are loaded using Java's dynamic code loading features.

Dynamic customization uses two generic micro-protocols developed for Cactus/J: rBoot and rControl. rBoot provides the minimal functionality required to load rControl. In particular, it connects to the object from which the code is to be loaded and accepts a message that contains rControl as a Java archive. rControl loads the actual micro-protocols required in the configuration. It remains active for the duration of the composite protocol, and thus, allows new micro-protocols to be loaded during execution. Using this technique, the object constructor in the composite protocol needs only to start the generic rBoot micro-protocol to support full dynamic customization. The current versions of RBoot and rControl use a separate TCP connection to load the code, but other alternatives are being explored such as using the underlying middleware platform or Jini technology.

While static customization is conceptually simple and easy to use, dynamic customization offers more flexibility. For example, the configurations in statically customized client and server protocols must match for the system to operate correctly, while dynamic customization allows a matching configuration to be loaded at execution time. This ability can be utilized in a number of ways. For example, the client can download the necessary micro-protocols from the server, the server can download micro-protocols from the client, or both the server and client can download micro-protocols from some external configuration service. Each of these options has useful application areas. For example, a client can download a multicast or load balancing micro-protocol that is used by a specific collection of replicated servers, while a server can download the secure communication micro-protocol required by a given client. An external configuration service allows the properties—and thus the configurations—to be defined for all [user,service] pairs without requiring direct manual configuration of protocols. All of these alternatives make it easier to deploy new or updated micro-protocols since the updates only need to be made at the clients or the servers, or the configuration service.

The ability to alter configurations dynamically introduces the need to coordinate these changes on different hosts to maintain consistency. For example, if a micro-protocol implementing consistent total ordering of client invocations is changed at runtime, all the server replicas must perform the change at the same time with respect to the flow of invocations. In our current prototype, dynamic customization is limited to the client side and only at the time when the client binds to a server. Since we do not consider configuration changes at the servers or more general runtime changes, the only coordination required is to ensure that the Cactus client loads the necessary micro-protocols before it passes requests to the Cactus server. We have explored coordination issues in the context of group communication services [3] and intend to apply these techniques in future versions of CQoS.

3 Micro-protocols for QoS Enhancement

Cactus micro-protocols can be used to implement any service property or function, but this paper focuses on QoS attributes related to fault tolerance, security, and timeliness. Other properties and functions such as caching, prefetching, and load balancing could

be implemented in similar ways. Note that none of the specific techniques used here are novel; indeed, all have been used in other CORBA and Java RMI systems, and prior to that, in RPC systems and other systems that use the request/reply paradigm. The novelty of our approach is the way in which they can be configured to realize different combinations of attributes.

3.1 Base Micro-protocols

A Cactus composite protocol typically includes micro-protocols that provide basic service functionality. In CQoS, these micro-protocols are ClientBase at the client and ServerBase at the server. ClientBase consists of three handlers:

- *assigner*. Bound to event newRequest, it assigns a server to the request and raises readyToSend.
- *syncInvoker*. Bound to readyToSend, it uses the server determined by *assigner* to issue the request. It checks the server status (*server_status()*), connects to the server if necessary (*bind()*), calls the server (*invoke_server()*), and raises invokeSuccess or invokeFailure depending on the result of the call.
- *resultReturner*. Bound to both invokeSuccess and invokeFailure, it provides the default processing of these events by releasing the waiting client thread when an invocation is completed.

ServerBase consists of two handlers:

- *getParameters*. Bound to newServerRequest, it extracts Cactus parameters from the request structure and raises readyToInvoke.
- *invokeServant*. Bound to readyToInvoke, it invokes the server object by calling the *invoke_servant()* method of the CQoS skeleton and raises invokeReturn.

Note that the basic behavior is broken into multiple handlers with events used to pass control from one handler to another. This allows the actual QoS micro-protocols to insert their processing at the appropriate points of the control flow. All the handlers in the base micro-protocols have been ordered to be the last ones executed when its respective event is raised. This makes it possible for other micro-protocols to do additional processing before these handlers are executed or to override them by stopping the event execution before their execution.

3.2 Fault Tolerance

Many fault-tolerance techniques are relatively easy to implement transparently. For example, method calls can be sent to replicated servers to tolerate host failures, and messages can be retransmitted to tolerate transient network failures. To hide the impact of such replication, the mechanisms used also need to eliminate duplicate messages and combine multiple replies. It may also be necessary to ensure that all server replicas receive method calls in the same order and have a consistent view of the server group membership. The current prototype does not support state transfer for either recovering or newly joining replicas. Such a facility could easily be added using standard techniques,

although it would lessen transparency since but it would require that server objects provide operations for getting and setting their state. Also, the current implementation only considers host crash failures, although a similar customizable approach could be used for less benign failures [9]. Currently, we assume the underlying platform handles network failures, but it would be easy to add retransmission micro-protocols if necessary.

Our current prototype has two replication micro-protocols: ACTIVEREP and PASSIVEREP. ACTIVEREP implements active replication where the request is sent to all server replicas and all non-crashed replicas reply. ACTIVEREP consists of one handler *actAssigner* that is similar to the base *assigner* except that it raises readyToSend asynchronously. The constructor of ACTIVEREP binds *actAssigner* to the event newRequest multiple times, once for each server. When the event is raised, an instance of *actAssigner* is executed for each replica. From this point, execution proceeds as outlined above—each instance of *actAssigner* raises readyToSend, which starts a separate instance of *syncInvoker*. The fact that readyToSend is raised asynchronously means that each instance of *syncInvoker* is executed concurrently by a separate thread and thus, the blocking server invocations (*invoke_server()*) are executed in parallel. The *actAssigner* handlers override the base *assigner* by executing before it and halting further execution associated with the event.

PASSIVEREP supports passive replication, where a designated primary server replies after forwarding the request to other replicas to keep them consistent. The client side consists of two handlers:

- *pasAssigner*. This handler overrides base *assigner*, and assigns the first non-failed server to serve the request.
- *primarySelector*. This handler overrides base *resultReturner* for event invokeFailure, marks the current primary as failed, and raises newRequest to re-execute the request.

The net result is that the client thread is not released until a proper result has been received or all replicas have failed. The primary server uses techniques similar to those used in ACTIVEREP to forward the request concurrently to the backup replicas. It also keeps track of requests already received, so that receiving a request again does not corrupt the server state.

The current prototype supports three different *acceptance semantics*, which determine when a request is considered completed and a reply can be returned to the client. CLIENTBASE by default implements a policy useful for the non-replicated case where the first reply (success or failure) to arrive is returned to the client. A second micro-protocol returns the result from the first successful execution and a third returns the majority value from non-failed replicas. Both of these micro-protocols consist of one handler that is executed before the base *resultReturner*.

The TOTALORDER micro-protocol ensures that all replicas receive requests from multiple clients in a consistent total order. Our prototype uses a sequencer-based total ordering algorithm, where a coordinator determines the ordering for each request, and multicasts it to the other replicas. Although failure of the coordinator is not currently tolerated, it would be simple to add this using standard techniques. TOTALORDER consists of three handlers in the Cactus server:

- *assignOrder*. Bound to readyToInvoke at the coordinator, it determines an ordering for each new request and sends it to other replicas in parallel using technique similar to ACTIVEREP.
- *checkOrder*. Bound to the same event on all replicas, it processes both requests and ordering information and releases any request that becomes eligible for execution.
- *checkNext*. Bound to invokeReturn, it determines if a waiting request can be executed.

3.3 Security

Many security features such as secure communication, authentication, and access control can be implemented transparently. Our current prototype includes provisions for message confidentiality, message integrity, and access control. As an example, DESPRIVACY encrypts and decrypts the request parameters and reply using DES. The client side uses a handler bound to readyToSend to encrypt the request parameters and a handler bound to invokeSuccess to decrypt the reply value. Both handlers are executed as the first handler for these events. The server-side decryption of request parameters is implemented by a handler that overrides the base *getParameters* handler. The server-side encryption of the reply value is implemented by a handler bound to invokeReturn. Note that since the micro-protocol encrypts only the request parameters and replies, the security level provided is slightly less than CORBA Security Level 1, which encrypts the entire request message. Integrity is provided by a signature-based scheme implemented by micro-protocols at the client and server, while access control is implemented by a micro-protocol at the server.

3.4 Timeliness

Providing rigorous timeliness guarantees is difficult and requires control of client admission and request scheduling. However, service differentiation properties that provide more timely service to high priority requests can be implemented as relatively simple micro-protocols. Our current prototype includes three such micro-protocols. The first, PRIORITYSCHED, manipulates thread priorities. It consists of one handler *setPriority* bound to readyToInvoke that sets the priority of the current thread based on the request priority. It is set to execute as the first handler for this event so that it can change the priority as early as possible. The second, QUEUEDSCHED, schedules request execution by queuing low priority requests if high priority requests are executing. This behavior is implemented by three handlers:

- *checkPriority*. Bound to readyToInvoke, it allows a request to either continue or queues it.
- *notifyWaiting*. Bound last to invokeReturn, it raises event requestReturned asynchronously with a low thread priority if no high priority requests are being executed.
- *wakeupNext*. Bound to requestReturned, it releases waiting low priority requests.

Note that the second handler uses a modified raise operation that allows the thread priority to be specified. This ensures that execution of *wakeupNext* does not interfere

with the thread that is returning the high priority request. Finally, the third micro-protocol, TimedSched, uses a similar strategy, except that it keeps track of how many high priority requests have arrived in a time period and only releases low priority requests one at a time when the number of high priority requests in the previous period is smaller than a threshold. Currently, the request priority is simply determined based on client identity, but other techniques could easily be added.

Note that both TotalOrder and the last two service differentiation micro-protocols order request execution, making it possible for the orders to conflict. That is, it is possible for the next request according to TotalOrder to be a low priority request that is queued and thus blocked by the service differentiation micro-protocols. This problem can be solved by including the service differentiation micro-protocols only at the coordinator of the total ordering algorithm. This ensures that the total order assignment respects request priorities.

Two changes were made in the Cactus runtime system to allow these micro-protocols to manipulate thread priorities. The first is the variant of the raise operation mentioned above that specifies the priority of the thread used to execute the handlers. The second preserves thread priorities in event operations. In particular, the handlers associated with a given event are guaranteed to be executed by a thread with the same priority as the thread that raised the event, unless specified otherwise.

3.5 Combining QoS Properties

The Cactus framework allows micro-protocols to be designed so that the composite protocol can be customized to provide different combinations of QoS micro-protocols. In our case, the fault-tolerance, security, and timeliness micro-protocols have been designed to work together in any combination. The fault-tolerance micro-protocols can be used in five different combinations: passive replication (1) or active replication with any combinations of total order and acceptance (4). Overall, a service can be configured with no fault tolerance or any of these five fault-tolerance combinations with any combination of the three security micro-protocols and any of the three timeliness micro-protocols. As a result, even this small set of micro-protocols can be configured in over 100 different ways.

While using Cactus does not automatically guarantee that micro-protocols are composable, it provides flexible mechanisms that allow the micro-protocol designer to maximize composability. One example is the event mechanism that allows handler execution to be ordered as desired, including provisions for overriding existing handlers. These facilities are used, for instance, to ensure that the decryption handler is executed transparently prior to all other handlers. Note, however, that the composability is the result of careful design of the event set and ordering of handler execution.

The current set of micro-protocols could be easily extended with different security, timeliness, and fault-tolerance techniques, and to handle more complicated invocation scenarios, such as an invocation from one replicated server to another. Additional security micro-protocols could also be added using the approach presented in [11], which includes numerous micro-protocols for confidentiality, as well as for other security attributes such as authenticity, non-repudiation, key distribution, and auditing. Each security attribute can also be enforced using combinations of two or more micro-protocols

if desired. Additional timeliness micro-protocols could include admission control and traffic enforcement, while additional fault-tolerance micro-protocols could include request logging, server recovery, and more rigorous failure detection and membership management.

4 Implementing CQoS on CORBA and Java RMI

The CQoS stub and skeleton form the middleware and application-dependent components of this framework, providing a mechanism to abstract system specific details and implement a standard interface by which Cactus protocols can access data in an implementation neutral manner. This section describes highlights of how these interceptors have been mapped to CORBA and Java RMI in the prototype implementation. We emphasize again that the actual implementation of the QoS attributes in the Cactus client and server is independent of the specific platform used.

4.1 CORBA

CORBA is a vendor-independent software architecture designed to facilitate object-based distributed computing. The architecture consists of three major components: an interface definition language (IDL), a communication infrastructure (ORB) supporting remote method invocation, and a variety of supporting services. Examples of services include a naming service, a security service, and an event notification service. The IDL definition is used to generate stubs and skeletons. Stubs marshal the invocation into a request and pass it to the ORB, which is responsible for transmitting the request from the client to the server host. Skeletons unmarshall the request and perform the invocation on the corresponding servant for the object. The result of the invocation is returned in an analogous manner.

Our approach to adding customizable QoS is to insert CQoS stubs and skeletons between the client and ORB and the ORB and the servant. These replace the conventional stubs and skeletons and are responsible for intercepting method invocation on both the client and server sides. The interception is completely transparent and no modifications are required to the client code, to the server code, or to the IDL description of the server interface. Among other things, this allows enhanced QoS to be added to legacy CORBA applications.

Figure 4 illustrates the different system components and their interactions. `Client` and `Servant` are the user-provided CORBA client and servant, where the servant is a Java object that implements the request processing for one or more CORBA objects. `stub`, `helper`, and `skeleton` are standard components generated by the IDL compiler. The standard `stub` and `skeleton` are replaced in our approach by the CQoS stub and skeleton and the `helper` is modified slightly to enable interception. `startup` on the server side is a standard object-specific initialization file modified slightly to enable interception. Dotted lines represent interactions at servant initialization and client bind time, while solid lines represent interactions at invocation time.

A careful naming convention for the POAs (Portable Object Adaptors) and the CQoS skeletons is used to enable the client side to locate the potentially replicated objects.

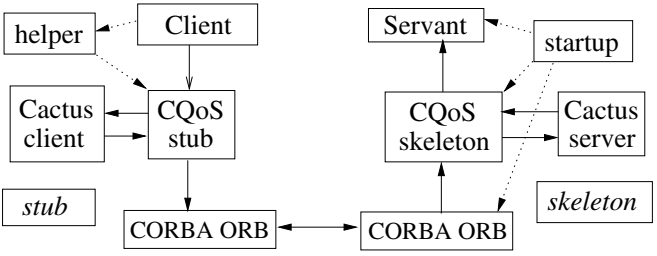


Fig. 4. System components and interactions in the CORBA implementation.

Generally, a CORBA object is represented by an IOR (Interoperable Object Reference) that is created by a POA when the servant that implements the object registers with the POA using the appropriate object identifier. We used a set of POAs in our system, one for each object replica, to create the IORs. The POA for the i^{th} instance of object with identifier "OID" is named "OID_agent_poa_i". Given that the POAs have different names, we can use the same object identifier "OID_CQoS_Skeleton" for the CQoS skeletons on all replicas. Using this convention, the CQoS stub can get the IORs of the object replicas by binding with the correct POA and using the object identifier "OID_CQoS_Skeleton".

Client- and server-side interception is implemented by modifying the `helper` and `startup` files, respectively. On the server side, the `startup` file is modified to start and register the CQoS skeleton rather than the application servant. Specifically, `startup` uses the above naming convention to create a POA for the object and register the CQoS skeleton with the POA. A pointer to the original servant is passed as an argument to the CQoS skeleton. This pointer is used within the skeleton to implement the *invoke_servant()* operation by making a native Java call to the servant object.

On the client side, a line in the standard `helper` file is modified to specify the CQoS stub as the stub to be used by this client. As described above, this stub has a method corresponding to each server object method, which converts the method call into an abstract request structure that is passed to the Cactus client using *cactus_request()*. The CQoS stub also intercepts a client's *bind()* operations. In our current implementation, the *bind()* operation simply creates a CQoS stub and returns it to the client, with the actual binding done at the time the client issues its first request. The CQoS stub may create multiple bindings (e.g., for replication), and the Cactus client can access the status of bindings and request rebinding through the Cactus QoS interface described above in section 2.2. Note that the CQoS skeleton uses identical techniques to establish connections between server object replicas when necessary.

Finally, the implementation of the *invoke_server()* operation provides communication between the CQoS stub and skeleton, and between CQoS skeletons, using CORBA facilities. We have completed two implementations based on two alternative CORBA interfaces defined for the IDL to Java mapping [25]. The first uses the Dynamic Invocation Interface (DII) at the client and the Dynamic Skeleton Interface (DSI) at the server, while the second uses CORBA streaming APIs. In the DII/DSI approach, the CQoS stub constructs a CORBA DII request object using a delegate for the CQoS skeleton and

sends the request using the *invoke()* operation provided by the request object. The CQoS skeleton provides an analogous *invoke()* operation that is called by the POA when the request object arrives. This operation uses DSI facilities to extract the method name and parameters, including any extra Cactus parameters added to the request.

In the streaming approach, the CQoS stub uses the streaming APIs to construct a CORBA portable output stream, write the request parameters to the output stream, invoke the server delegate with the output stream as a parameter, and read the result from the resulting input stream. The CQoS skeleton provides an *invoke()* operation in a manner similar to the DII/DSI approach, but in this case it uses stream API operations to extract the method name and parameters from the input stream. Regardless of the method used to extract the parameters, the CQoS skeleton creates an abstract request object that is passed to the Cactus server by calling *cactus_invoke()*. The experimental results in section 5 demonstrate the performance benefits of this approach relative to the use of DII/DSI.

4.2 Java RMI

Java Remote Method Invocation (RMI) is a communication architecture aimed at integrating the distributed object model into the Java programming language while maintaining its original semantics [29]. Overall, the RMI architecture is relatively similar in structure to the CORBA architecture in figure 4, with automatically generated stubs hiding the lower-level communication details from the applications. However, the RMI architecture since JDK 1.2 does not use server-side skeletons and does not require a separate helper file. The RMI specification supports multiple underlying protocols—the default JRMP (Java Remote Method Protocol) and CORBA IIOP—and custom stubs may be generated for each of these using the `rmic` stub compiler. The architecture also provides the RMI registry, which is a naming service used to bind remote objects with generic names. Clients use the registry to locate remote objects through methods provided in the `java.rmi.Naming` class.

Server-side interception for RMI is modeled on our approach on CORBA. However, RMI is simpler than CORBA and does not have concepts such as POA and DSI, which affects implementation details. Since Java no longer supports server side skeletons, we introduce the CQoS skeleton as a proxy object that is modeled on a typical RMI (JDK 1.1) skeleton. We use a naming convention for the skeletons at the object replicas. Specifically, the skeleton for the i^{th} replica of object with identifier "OID" registers with the Java naming service using name "OID_CQoS_Skeleton_i". The client side CQoS stubs then bind to these skeletons, rather than the original objects, and thus, all client requests are automatically delivered to the skeletons. A mechanism similar to DSI is simulated in RMI by having the skeleton export only a generic *invoke()* method (`java.lang.Object invoke(java.lang.Object[])`). The Java request structure containing the actual method to be called is passed as a parameter to this method. The actual invocation of the server method is done through a native Java call similar to the CORBA implementation.

Client-side interception is based on simply replacing the standard stub with a CQoS stub with the same name. The stub provides one method for each of the server methods and when a method is called, the stub creates the abstract request structure and notifies

the Cactus client. When the Cactus client wants the request to be sent to the server, it calls the *invoke_server()* method of the stub with the request as a parameter. Similar to the CORBA implementation, the stub binds to the server when the client issues its first request. We are working on extending the Cactus IDL compiler to automatically generate the CQoS stubs and skeletons for Java RMI.

Note that while Java RMI currently supports both JRMP and IIOP, the default is currently in the process of being changed to IIOP, which allows RMI objects to access CORBA objects if they conform to a small set of restrictions. These RMI-IIOP systems can be customized using the CQoS on CORBA interception mechanisms described above. To achieve this, RMI-IIOP stubs are simply replaced with customized CQoS stubs for CORBA. This interception approach can be extended to any RPC-like client/server communication model.

5 Experimental Results

The performance of the approach was tested using a simple BankAccount object that provides operations for setting and retrieving the balance of a bank account. Tests were conducted on a cluster of 600 MHz Pentium III PCs running Linux 2.2.14 connected by a 1 Gbit Ethernet. The CORBA tests were conducted using Visibroker 4.1, while the Java RMI tests use Java 2 SDK version 1.3 for Linux.

The first set of experiments focused on measuring the overhead imposed by our approach. We first measured the response time using the standard middleware platform (CORBA or Java RMI) and then ran the same experiments for different combinations of CQoS components. Each test run measured the time to execute 10000 pairs of *set_balance()* and *get_balance()* operations, and was run multiple times. The client and server objects were on different machines. The results are given in table 1, where each line adds one more CQoS component into the configuration. Note that in the CORBA case, adding the CQoS stub and CQoS skeleton does not simply add them to the baseline test, but replaces the original stub and skeleton generated by the standard IDL compiler. The CORBA tests in the table were performed using the stream API implementation. The Cactus client and server were configured with only the base micro-protocols. The column labeled “ohead” indicates the overhead of the added component compared to the previous configuration, while the subsequent column gives the cumulative overhead compared to the baseline.

Overall Java RMI performance, both the baseline and the CQoS enhanced, appears to be better than CORBA. We speculate that this is because Java RMI is a lighter weight middleware without the need, for example, to support multiple programming languages. The larger CQoS overhead with CORBA is due partially to the fact that the stub must first convert a request into the abstract form and then transmit it on the output stream one parameter at a time, whereas the Java RMI version simply transmits the abstract request object. A number of optimizations have been used in both implementations to improve performance, such as reuse of the request data structures to avoid object creation. The cost of the Cactus client and server have also been reduced by optimizing the Cactus runtime system. For example, use of a thread pool for event handling reduced overhead considerably.

Table 1. Average response times (in ms)

Configuration	set + get	one call	ohead	cum ohead
Original CORBA	2.74	1.37	0	0
+ CQoS stub	3.01	1.51	0.14	0.14
+ CQoS skeleton	3.06	1.53	0.02	0.16
+ Cactus client and server	3.44	1.72	0.19	0.35

Configuration	set + get	one call	ohead	cum ohead
Original Java RMI	2.19	1.10	0	0
+ CQoS stub	2.21	1.11	0.01	0.01
+ CQoS skeleton	2.27	1.14	0.03	0.04
+ Cactus client and server	2.61	1.31	0.17	0.21

We also tested the DII/DSI-based implementation and performed other tests to identify the impact of Java garbage collection and compiler optimization on the CORBA implementation. In the first case, we determined that the DII/DSI numbers were consistently larger. For example, the time for one call with just the CQoS stub was 1.64 ms, while the time for one call with all components was 2.16 ms. Thus, the overhead was approximately 10%-20% despite our attempts to optimize performance such as reuse of the name-value list objects required by DII/DSI. Java garbage collection also impacted performance results, especially the variance between runs. To measure this impact, we increased the heap size and set JVM parameters so that garbage collection was unlikely during a test run. This resulted in considerable performance improvement in all cases and a reduction in the variance. For example, the average time for one standard CORBA call was reduced to 1.29 ms, while the time for one call including all CQoS components was reduced to 1.48 ms. Finally, the impact of the optimizations performed by Sun's standard "HotSpot" compiler was determined to be significant. For example, the average response time for one call with all components was 3.04 ms when optimization was disabled.

The second set of experiments (table 2) illustrates the response times for different QoS configurations. Each micro-protocol increases the response time by using more CPU time (e.g., encryption), sending more messages (e.g., replication and total ordering), or both. In tests with multiple object replicas, the client and each replica are all on separate machines. The CORBA numbers are again based on the stream API implementation. As expected, the numbers indicate that adding functionality that introduces additional messages or that is CPU-intensive is relatively expensive, but simple functionality costs relatively little. Note that the increase in response time when the DesPrivacy micro-protocol is used is greater for CORBA than for Java RMI. We attribute this to the fact that the entire request object is encrypted and transmitted as a single CORBA parameter of type `Any`, whereas the arguments of calls in configurations without encryption can be transmitted as simple data types. In contrast, the request object is transmitted as a single entity in every case with the Jave RMI implementation, so that the only change is the actual CPU overhead associated with execution of the cryptographic algorithms.

Table 2. Response times for different configurations (in ms)

Configuration	num servers	CORBA		Java RMI	
		set + get	one call	set + get	one call
DesPrivacy	1	45.92	22.96	8.57	4.29
PassiveRep	3	6.51	3.26	5.56	2.78
ActiveRep	3	6.50	3.25	4.40	2.20
+ MajorityVote	3	7.98	3.99	4.77	2.39
+ TotalOrder	3	11.23	5.62	8.14	4.07
ActiveRep+TotalOrder	3	8.50	4.25	7.40	3.70
+ DesPrivacy	3	76.70	38.25	13.63	6.84

The last set of experiments in table 3 illustrates the behavior of the TimedSched service differentiation micro-protocol alone, and when combined with other micro-protocols. For these tests, we statically designated some clients as high priority and others as low priority. In these particular tests, we used one high priority client and varying numbers of low priority client. The results indicate that the TimedSched micro-protocol provides relatively good service differentiation and protects a high priority client well from the impact of low priority clients.

6 Related Work

As already noted, the specific fault-tolerance, security, and timeliness techniques used in CQoS are standard approaches that have been used in many other systems. Therefore, we focus here more directly on research related to adding QoS guarantees to CORBA, Java RMI, and other distributed object platforms. We will also shortly discuss how the Jini technology relates to our approach.

The QoS work in distributed object systems can generally be classified into one of several approaches depending on how the QoS functionality is added to the system. These include the service approach, the integrated approach, the interception approach, and the gateway based approach.

The service approach implements QoS enhancements as separate services transparently to the middleware platform, while the integration approach directly modifies the platform to provide the enhancements. Both approaches can be made transparent to the application. The integration approach typically provides better performance since low level optimizations are possible, but interoperability becomes an issue. While the service approach is better for portability and interoperability, it is difficult to make any guarantees for communication between the client and service, which makes it impossible to guarantee end-to-end timeliness or security. A good discussion of the tradeoffs between this approach and the previous one is presented in [7]. The service approach has been used for fault tolerance [6,22], and standard CORBA services such as the security service can be viewed as examples of this approach. The integration approach has been used to enhance fault tolerance [15,17] and timeliness properties [28], and could naturally be used to enhance any other QoS attribute.

Table 3. Average response times (in ms)

Configuration	num servers	num low pri. clients	CORBA high pri.	CORBA low pri.	Java RMI high pri	Java RMI low pri
TimedSched	1	1	1.74	3.61	1.36	3.31
	1	2	1.85	4.45	1.39	3.53
	1	3	1.92	5.19	1.43	3.60
	1	4	1.97	5.33	1.48	4.27
+ ActiveRep	3	1	3.45	6.97	2.33	4.83
	3	2	3.45	8.39	2.33	5.30
	3	3	3.46	10.22	2.34	6.20
	3	4	3.47	12.10	2.36	7.27
+ MajorityVote	3	1	4.20	8.62	2.51	5.28
	3	2	4.22	9.98	2.62	7.30
	3	3	4.23	12.05	2.71	8.23
	3	4	4.24	13.95	2.77	9.37
+ TotalOrder	3	1	5.65	11.18	4.10	8.49
	3	2	5.60	13.41	4.14	10.09
	3	3	5.55	15.14	4.17	12.00
	3	4	5.56	18.18	4.17	14.17
ActiveRep+TotalOrder	3	1	4.39	8.70	3.68	7.38
	3	2	4.38	10.01	3.86	8.35
	3	3	4.35	12.18	3.85	9.64
	3	4	4.39	13.60	3.87	11.56

The interception approach, as the name suggests, works by intercepting middleware messages or requests. These systems are classified based on whether the interception takes place above or below the middleware. With CORBA, approaches that operate above the ORB have used mechanisms such as smart stubs and interceptors [31], delegates [16], and reflection [13]. The CORBA 2.2 standard also provides a limited interception mechanism, but it has a number of limitations [26,32]. Approaches that operate below the ORB intercept the IIOP messages sent by the ORB before they are passed to the operating system [18,19,21]. Similar approaches have also been used with Java RMI [20] and other distributed object-based systems [5,14,26]. The interception approach has been used for fault tolerance (e.g., [5,19,21]), security (e.g., [5,21,31]), and timeliness (e.g., [14,16, 18,21]).

Interception approaches that operate above the middleware provide a higher level of abstraction for implementing the QoS enhancements. For example, existing middleware services can be used to locate, communicate with, and detect the failure of servers. Other services such as CORBA security services can also be used if desired, while approaches that do not build on top of the platform must use lower-level facilities to accomplish these tasks. However, interception below the middleware can provide very good performance since it can utilize efficient custom transport protocols. Our approach can be classified as interception above the middleware.

In the gateway approach, a gateway component inserted at the transport layer between the clients and servers is responsible for implementing the QoS enhancements. The gateway may be a single component [1] or may have separate client and server sides that interact [27]. The gateway must be able to intercept IIOP messages. This is done in [1] by using a firewall to forward all IIOP messages to the gateway. The QuO distributed gateway [27] uses a client-side proxy to direct requests to the client-side gateway, which can then interact with the server-side gateway using any mechanism, e.g., a group communication service. This approach does not require any modifications to the ORB and could be used to implement any type of enhancement. However, since the gateway may reside on a different host than the client and server, it may not be possible to provide strict end-to-end QoS guarantees. Moreover, the extra communication steps introduce a performance penalty.

The main features that separate CQoS from other work in this area are its design for portability across different distributed object platforms and its support for fine-grain application-specific customization. The novel architecture in CQoS with two components, one middleware specific and the other generic, provides a beneficial separation of concerns. With this architecture, researchers can focus on developing generic improved algorithms for QoS enhancements, while the work on developing better interception mechanisms proceeds independently. A similar goal of middleware neutrality is presented in [26], but no implementation on CORBA or Java RMI is described, nor any performance results.

The fine-grain configurability supported by Cactus provides the flexibility needed to customize guarantees independently for each application across a broad spectrum of QoS attributes. Although previous work on CORBA often encompasses different QoS attributes (e.g., [18,21,27,31]), to our knowledge, no other approach provides a comparable ability to implement custom combinations of different QoS attributes. Customization of multiple QoS attributes has been addressed in other types of object-based systems, however. The metaobject-based FRIENDS [5] architecture is the closest analogue to CQoS. Metaobjects are roughly equivalent to Cactus micro-protocols as used here, and both approaches emphasize a clear separation of concerns between mechanisms and the application of QoS enhancements. A major concern for any reflection-based approach, however, is the need for a language that supports reflection, the level of reflection supported, and performance overhead. In contrast, CQoS depends on middleware-based interface definitions to introduce QoS enhancements using interceptors working in conjunction with generic QoS components. Another related approach presents an architecture in which multiple interceptors can be stacked to provide combinations of attributes [26]. However, as noted above, the architecture has apparently not yet been implemented.

Finally, Sun's Jini technology [30] is largely complementary to CQoS. Jini is a set of specifications that enables services to discover one another and cooperate in a distributed system. One of its novel features is that a client does not need to know *a priori* where the services are located or how to access them. Rather, a client uses the Jini lookup service to locate services and then dynamically loads a proxy object (i.e., a stub) that implements communication with the service. It would be possible to use these Jini facilities to locate and load micro-protocols for CQoS services. On the other hand, the fine-grain customization and ability to change behavior at runtime provided by CQoS

could be applied to construct configurable and adaptive Jini proxies. CQoS could also be used to enhance transparently the QoS properties of existing Jini services.

7 Conclusions

This paper has presented the CQoS architecture as a single unified framework for providing customizable combinations of multiple QoS attributes across multiple middleware platforms. This novel architecture provides separation of concerns between the algorithms that implement QoS attributes and the details of the underlying middleware platform. This is accomplished by dividing CQoS into two components, each of which addresses one of these concerns and provides the appropriate interfaces for interaction between the components. We presented the architecture and interfaces, described the implementation of this architecture on both the CORBA and Java RMI platforms, and showed preliminary performance results.

Future work will concentrate in several areas. In the near term, these include adding to the collection of available micro-protocols, investigating further performance optimizations, and experimenting with more realistic applications. In the longer term, our goal is to incorporate results from other Cactus-related research involving adaptive behavior and mobile systems to provide similar support for distributed object applications.

Acknowledgments. G. Townsend implemented the Cactus/J system used as the basis for this work. Also, R. Gruber and A. Puder provided helpful comments and suggestions.

References

1. T. Benzel, P. Pasturel, D. Shands, D. Sterne, G. Tally, and E. Sebes. Sigma: Security and interoperability for heterogeneous distributed systems. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 308–319, Jan 2000.
2. N. Brown and C. Kindel. *Distributed Component Object Model Protocol—DCOM/1.0*. Microsoft Corp., Redmond, WA, Jan 1998. Network Working Group Internet Draft.
3. W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 635–643, Mesa, AZ, Apr 2001.
4. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, IN, Oct 1998.
5. J.-C. Fabre and T. Perennou. A metaobject architecture for fault tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers, Special Issue on Dependability of Computing Systems*, 47(1):78–95, Jan 1998.
6. P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 150–159, Niagara-on-the-Lake, Canada, Oct 1996.
7. P. Felber, B. Garbinato, and R. Guerraoui. Towards reliable CORBA: Integration vs. service approach. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming*, pages 199–205. Verlag, 1997.

8. M. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, Oct 1998.
9. M. Hiltunen, V. Immanuel, and R. Schlichting. Supporting customized failure models for distributed software. *Distributed Systems Engineering*, 6:103–111, 1999.
10. M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, Jun 1999.
11. M. Hiltunen, R. Schlichting, and C. Ugarte. Enhancing survivability of security services using redundancy. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001)*, Gothenburg, Sweden, Jul 2001.
12. M. Hiltunen, R. Schlichting, and G. Wong. Cactus system software release. http://www.cs.-arizona.edu/cactus/software.html, Dec 2000.
13. M.-O. Killijian and J. Fabre. Implementing a reflective fault-tolerant CORBA system. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 154–163, Nuremberg, Germany, Oct 2000.
14. R. Koster and T. Kramp. Structuring QoS-supporting services with smart proxies. In *Middleware 2000*, pages 273–288, 2000.
15. S. Landis and S. Maffeis. Building reliable distributed systems with CORBA. In *Theory and Practice of Object Systems*. John Wiley, NY, 1997.
16. J. Loyall, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, J. Gossett, and C. Gill. Comparing and contrasting adaptive middleware support in wide-area and embedded distributed object applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 625–635, Mesa, AZ, Apr 2001.
17. S. Maffeis. Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies,*, Monterey, CA, June 1995.
18. P. Melliar-Smith, L. Moser, V. Kalogeraki, and P. Narasimhan. Realize: Resource management for soft real-time distributed systems. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 281–293, Hilton Head, SC, Jan 2000.
19. L. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki. Eternal: Fault tolerant and live upgrades for distributed object systems. In *Proceedings of the DARPA Information Survivability Conference and Exposition (Discex'2000)*, pages 184–196, Hilton Head, SC, Jan 2000.
20. N. Narasimhan, L. Moser, and P. Melliar-Smith. Interception in the Aroma system. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 107–115, Jun 2000.
21. P. Narasimhan, L. Moser, and P. Melliar-Smith. Using interceptors to enhance CORBA. *Computer*, 32(7):62–68, Jul 1999.
22. B. Natarajan, A. Gokhale, S. Yajnik, and D. Schmidt. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications*, Antwerp, Belgium, Sep 2000.
23. Object Management Group. *The Common Object Request Broker: Architecture and Specification (Revision 2.4.1)*, Nov 2000.
24. Object Management Group. Fault tolerant CORBA. OMG Technical Committee Ducument orbos/2000-04-04, Mar 2000.
25. Object Management Group. *IDL to Java Language Mapping, version 1.1*, Jul 2001.
26. J. Pruyne. Enabling QoS via interception in middleware. Technical Report HPL-2000-29, HP Labs, 2000.
27. R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquire, and J. Loyall. An object-level gateway supporting integrated-property quality of service. In *Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint-Malo, France, May 1999.

28. D. Schmidt, A. Gokhale, T. Harrison, and G. Parulkar. A high-performance end system architecture for real-time CORBA. *IEEE Communications Magazine*, pages 72–77, Feb 1997.
29. Sun Microsystems. *Java Remote Method Invocation Specification, Rev. 1.50*. Sun Microsystems, Inc., Mountain View, CA, Oct 1998.
30. Sun Microsystems. *Jini Technology Core Platform Specification, Version 1.1*. Sun Microsystems, Inc., Mountain View, CA, Oct 2000.
31. V. Thomas, S. Kareti, W. Heimerdinger, and S. Ghosh. Mediators and obligations: An architecture for building dependable systems containing COTS software components. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nuremberg, Germany, Oct 2000.
32. M. Wegdam and A. v. Halteren. Experience with CORBA interceptors. In *Proceedings of Workshop on Reflective Middleware (RM 2000)*, Apr 2000.
33. J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

$2K^{Q+}$: An Integrated Approach of QoS Compilation and Reconfigurable, Component-Based Run-Time Middleware for the Unified QoS Management Framework*

Duangdao Wichadakul, Klara Nahrstedt, Xiaohui Gu, and Dongyan Xu

Department of Computer Science
University of Illinois at Urbana-Champaign
{wichadak, klara, xgu, d-xu}@cs.uiuc.edu

Abstract. Different distributed component-based applications (e.g., distributed multimedia, library information retrieval, secure stock trading applications), running in heterogeneous execution environments, need different quality of service (QoS). The semantics of QoS requirements and their provisions are application-specific, and they vary among different application domains. Furthermore, QoS provisions vary per applications in heterogeneous execution environments due to the varying distributed resource availability. Making these applications QoS-aware during the development phase, and ensuring their QoS guarantees during the execution phase is complex and hard.

In this paper, we present a unified QoS management framework, called $2K^{Q+}$. This framework extends our existing run-time $2K^{Q}$ middleware system [1] by including our uniform QoS programming environment and our automated QoS compilation system (*Q-Compiler*). The uniform QoS programming and its corresponding QoS compilation allow and assist the application developer to build different component-based domain applications in QoS-aware fashion. Furthermore, this novel programming and compilation environment enables the applications to be instantiated, managed, and controlled by the same reconfigurable, component-based run-time middleware, such as $2K^{Q}$, in heterogeneous environments.

Our experimental results show that different QoS-aware applications, using the $2K^{Q+}$ framework, get configured and setup fast and efficiently.

1 Introduction

The rapid growth of distributed component-based environments and coexistence of different application domains, such as multimedia and electronic commerce, present significant challenges to the provision of different applications' Quality

* This work was supported by the National Science Foundation under contract numbers NSF EIA 9870736, NSF CCR-9988199, the Air Force Grant under contract number F30602-97-2-0121, NSF CISE Infrastructure grant under contract number NSF EIA 99-72884, and NASA grant under contract number NASA NAG 2-1250.

R. Guerraoui (Ed.): Middleware 2001, LNCS 2218, pp. 373–394, 2001.

of Service (QoS). First, different domain applications have specific semantics for QoS requirements and provisions. For example, multimedia and library information retrieval applications are concerned about the service qualities of audio, video streaming (e.g., frame rate, frame size, sampling rate, end-to-end delay) and messaging (e.g., priority, response time, reliability), respectively. Developing an application to be QoS-aware during the application development cycle, and ensuring its QoS provisions during the application's run-time are non-trivial tasks for a specific application and even harder for different applications. Second, with the pervasive computing, these applications are expected to be executed in heterogeneous computing and communication environments with different capacities of processing power (e.g., high performance PC, PDAs), battery power, and network bandwidth (e.g., wired, wireless networks). The big challenging problem, as shown in Fig. 1, is:

"How should a unified QoS management framework look like to allow different distributed component-based applications to use it and achieve required guarantees in heterogeneous, dynamic computing and communication environments ?"

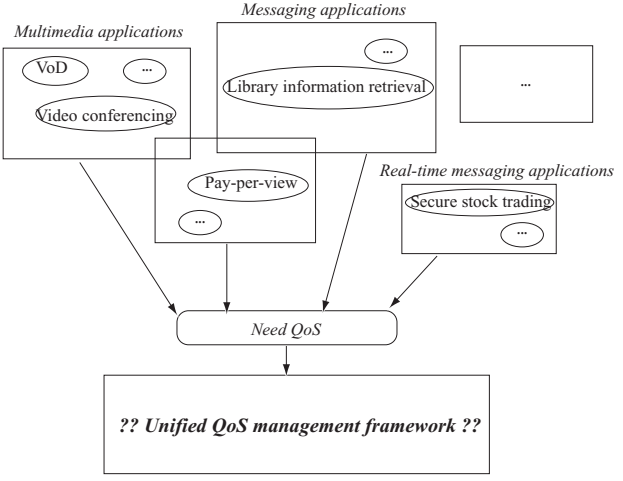


Fig. 1. Challenging Problem

Several QoS architectures and approaches [2,3,4,5,6,7] already exist. However, all of them are designed only for one domain of applications, particularly for multimedia applications, or designed to handle particular aspect of QoS provisions such as the QoS-aware resource management [8]. If one wants to implement and run a different domain of applications, the existing tailored architectures won't be applicable or reusable. In the distributed object computing (DOC) middleware such as CORBA, some QoS support [9,10,11,12] has been proposed. In the CORBA case, an application will be QoS-aware if the application developer

deploys QoS-related interfaces of requested QoS services. Hence, an application developer has to learn (a) different IDL interfaces for different types of QoS, (b) the semantics of these interfaces, and (c) how to translate his/her application QoS requirements into these interfaces and their parameters appropriately. TAO project [13] developed the optimized CORBA ORB which supports the real-time messaging. QuO project [14] allows an application developer to develop distributed applications that can adapt to the changing quality of service in CORBA environment. The QuO considers the quality of service in the broader domain including (1) quality of object interaction such as reliability, (2) real-time method invocation, and (3) security. QoS in distributed object computing middleware is also tailored towards specific domain of applications or particular aspect of QoS provisions. Hence, no work fully solves the above challenging problem.

In this paper, we present the $2K^{Q+}$ framework, shown in Fig. 2), a unified QoS management framework, which allows and assists the application developer to develop different component-based domain applications in QoS-aware fashions. There different domain applications are able to be developed in the *same* programming and compilation environment, and to be instantiated, managed and controlled uniformly by the *same* reconfigurable, run-time middleware. This means that the unification of the QoS management framework happens at two levels: (1) the application developer can develop different domain applications via the *same* set of QoS specifications, and (2) different domain applications can run via the *same* run-time middleware in heterogeneous environments.

The key components in the $2K^{Q+}$ framework are (1) the *uniform QoS programming environment*; (2) the *Q-Compiler*; and (3) a reconfigurable, component-based run-time middleware such as the *run-time* $2K^{Q}$ *middleware*.

The *uniform QoS programming environment* provides a *uniform set* of customizable QoS specifications that can be used to develop different domains of QoS-aware applications. These QoS specifications are considered as the input *"source code"* of the Q-Compiler.

The *Q-Compiler*, the automated QoS compilation system, is the core of the unified QoS management. It compiles the corresponding QoS specifications into a well-defined set of QoS-enabled meta information, called QoS-aware Component-based Application Specification (QoSCASpec), considered as the Q-Compiler's *"object code"*, and used uniformly during the application execution phase.

QoSCASpec includes (1) possible delivery forms (configurations) of the QoS-aware application for running in heterogeneous execution environment with different resource availability; and (2) the association of reusable middleware service components and system resources for each configuration with specific semantics of QoS requirements.

The *run-time* $2K^{Q}$ middleware is the component-based, and dynamically reconfigurable run-time system. It is running in the distributed machines and it is the core of application execution. It assists the Q-Compiler to perform the distributed system resource translation during the application development phase. Furthermore, it instantiates, manages and controls a QoS-aware applica-

tion during the application execution phase, based on the (1) Q-Compiler result, the application's QoSCASpec, (2) incoming user QoS request, and (3) dynamic run-time constraints such as resource availability, execution environment, and mobility.

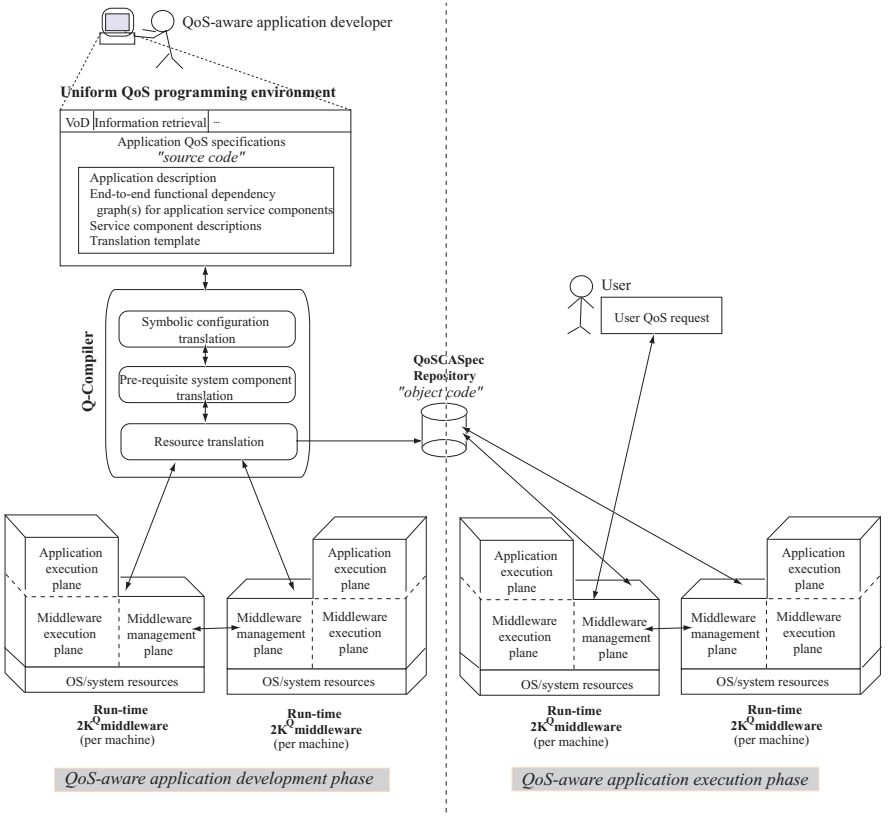


Fig. 2. $2K^{Q+}$: A Unified QoS Management Framework

The rest of the paper is organized as follows. In Sect. 2, we describe the uniform QoS programming environment, and how to develop a QoS-aware application via the uniform set of QoS specifications. In Sect. 3, we give the overview of the Q-Compiler architecture, and its core, the multi-aspect QoS translations. In Sect. 4, we describe the detailed architecture of the run-time $2K^Q$ middleware, how it assists the Q-Compiler to perform the distributed system resource translation, and how it instantiates, manages and controls a QoS-aware application during the application execution phase. In Sect. 5, we describe the implementations and our experimental results. In Sect. 6, we discuss the related work. Finally, we draw conclusions in Sect. 7.

2 Uniform QoS Programming Environment: Application Developer's View

The goal of our uniform QoS programming environment is to allow the application developer develop QoS-aware applications in different domains easily and uniformly via the same set of QoS specifications. To achieve this goal, it provides the following set of QoS specifications: (a) the application description, (b) the end-to-end functional dependency graph(s) of application service components, (c) the service component descriptions, and (d) the user-to-application-specific translation template.

The application description allows an application developer to specify the general information about the application, such as name, category, and accessibility. The Q-Compiler associates this information with the compiled results of the application. The run-time $2K^Q$ middleware uses this information as an index to the appropriate application's configuration(s). Figure 3 shows the application description for two applications: (a) the library information retrieval application, and (b) the video on demand application.

Attributes	Values
Application name	QLibAccess
Application category	Information retrieval
Accessibility	Public

(a) Library Information Retrieval Application

Attributes	Values
Application name	MonetVOD
Application category	Video on Demand
Accessibility	Public

(b) Video on Demand Application

Fig. 3. Application Description for Two Different Applications (Example)

The end-to-end functional dependency graph of application service components allows the application developer to implement an application via the composition of service components flexibly[1]. In a functional graph, we differentiate among service components as follows: each node represents a service component which can be a *specific service component*[2] (e.g., QLibServer, MPEG-II renderer, MPEG-II decoder), a *composite service component* (e.g., VoD Client consisting of MPEG-II decoder, and MPEG-II renderer), or a *generic service component*[3] (e.g., Information Retrieval Service (IRS), Encoder Service (ES), Decoder Service (DS)). The dotted-rounded rectangle in Fig. 4 represents a

[1] We assume that the interfaces between two connected components are clearly defined, and the programming environment can check the compatibility between two connected components in the functional dependency graph.

[2] A specific service component represents a specific implementation of a functional unit. We assume that specific service components have been already built, and there exists service component repository storing these components implemented for different devices, resource demands, and with different quality levels in mind.

[3] A generic service component represents the requirement of specific service without specific implementation. The application developer can develop an application via the functional dependency graph even though he/she doesn't have all built specific

machine, and a solid line from service component A to service component B represents a data flow from component A to component B. A functional graph is *fully-defined* if all service components are *substituted* with specific service components or sets of specific service components (in case of composite service component), and it is *partially-defined* if a service component is specified with a generic service component or a composite service component with some generic service components. Figure 4 shows the application functional graph for two applications: (a) the library information retrieval application, and (b) the video on demand application.

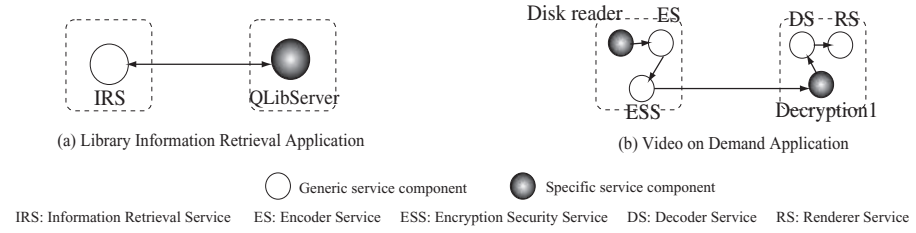


Fig. 4. Application Functional Graph for Two Different Applications (Example)

The service component description allows an application developer to specify individual service component's information which is required by the Q-Compiler. Figure 5 shows the service component descriptions for two applications: (a) the library information retrieval application, and (b) the video on demand application. The "*" represents undefined value which will be determined by the Q-Compiler. The "**" represents the undefined value which will be determined by the run-time $2K^Q$ during the application execution time.

The user-to-application-specific translation template (UtoA template) (as shown in Fig. 6) allows an application developer to define the mapping between different user QoS levels (e.g., UserIrQoS:High, UserPVQoS:High) and corresponding application-specific QoS categories (e.g., Information retrieval, Performance:Video) and QoS dimensions (e.g., request priority, format, resolution)[4]. User QoS levels are determined by the application developer as available application-category-specific QoS levels for the user during the application request time. Application-specific QoS categories and dimensions are determined by the application developer as the input QoS parameters to the Q-Compiler,

service components. A generic service component will be substituted with possible specific service component(s) during the QoS compilation time.

[4] *QoS category* and *QoS dimension* are part of *QoS specifications* to characterize non-functional properties of a specific application [15,16]. A *QoS dimension* defines a qualitative or quantitative attribute for a *QoS category*, and it is used to characterize a particular category [16]. For example, QoS dimension "resolution" is an attribute of QoS category "Performance:Video". QoS dimension can be interchangably used with QoS parameter.

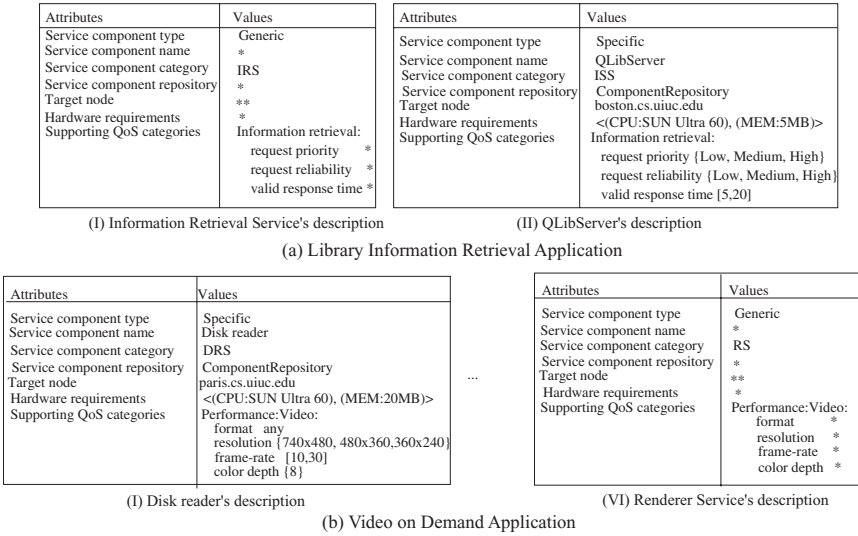


Fig. 5. Service Component Descriptions for Two Different Applications (Example)

corresponding to different user QoS levels. *QoS levels* can be generated by the Q-Compiler, based on all combinations of user QoS levels in different application-specific QoS categories. However, without loss of generality, we assume in this paper that the Q-Compiler will generate *QoS levels* for an application, based on combinations of only the same user QoS levels in all different application-specific QoS categories. Figure 6 shows the UtoA template for two different applications: (a) the library information retrieval application, and (b) the video on demand application.

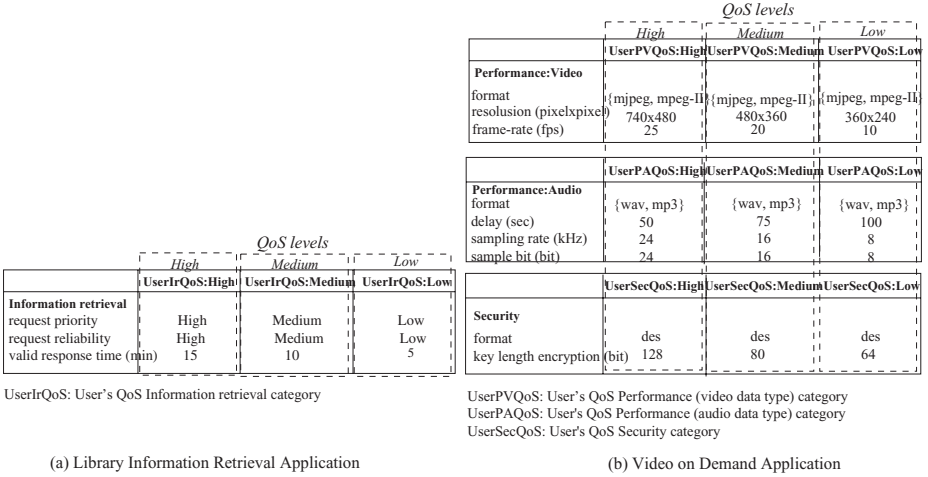
The uniform QoS programming environment enables the first level of unification, mentioned in the Introduction. Once the above QoS specifications are in place, the Q-Compiler starts to compile the QoS specifications into the QoS-enabled meta information as described in the following section.

3 Q-Compiler: Automated QoS Compilation System

The primary goal of the Q-Compiler[5], which is the key part of the $2K^{Q+}$ unified QoS management, is to compile QoS specifications of different domain applications into QoS-enabled meta information that can be used uniformly by the distributed run-time middleware during the application execution in heterogeneous environments.

In the Q-Compiler, we consider the following resulting output information as QoS-relevant to ensure the QoS provisions for different domain applications during their execution:

[5] The details of the Q-Compiler framework can be found in [17].



(a) Library Information Retrieval Application

(b) Video on Demand Application

Fig. 6. User to Application Specific Translation Template for Two Different Applications (Example)

- Possible delivery forms of an application, associated with different quality levels, for heterogeneous computing and communication environments, and for adaptations;
- Association of each delivery form with suitable set of middleware service components and their appropriate QoS parameters;
- System resource requirements of each delivery form with its associated middleware service components to be used (1) for resource reservation in case it exists, (2) for selection of the most suitable delivery form, and (3) for global resource optimization.

To achieve the primary goal, the Q-Compiler (as shown in Fig. 7) consists of three aspects of translations: (1) *symbolic configuration translation*, (2) *prerequisite system component translation*, and (3) *resource translation*.

The **Symbolic Configuration Translation** compiles the input QoS specifications of an application into possible delivery forms (configurations). This aspect of translation is based on the *generic service component's substitution* and *application functional graph transformation* with the *consistency check* of QoS requirements and QoS provisions between two consecutive specific service components in the graph. The pre-defined possible configurations for different domain applications are provided as an internal information for the translation. The symbolic configuration translation can be considered as the *horizontal end-to-end configuration translation* in the application layer. The results of the symbolic configuration translation are the *symbolic QoS configurations*, represented by fully-defined functional graphs with associated *QoS levels*. Note that the number of possible configurations for an application is limited by some factors such as the failure of consistency check, the number of generated QoS levels which

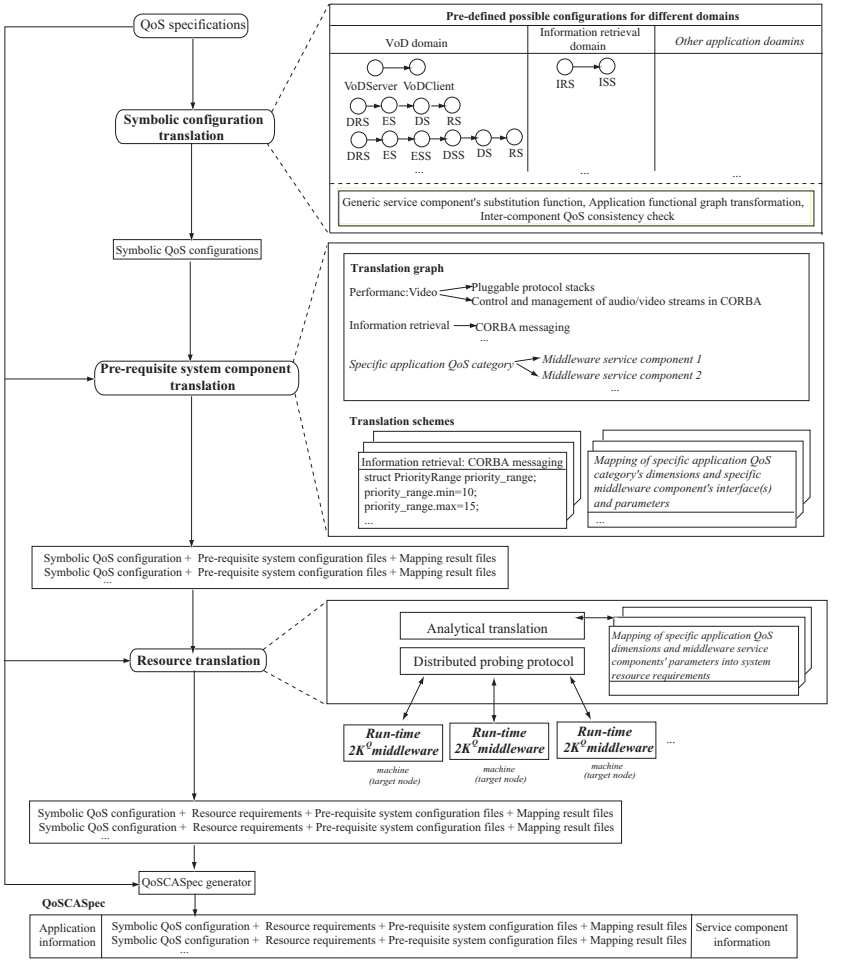


Fig. 7. Multi-aspect QoS Translations

is usually small, and the limited number of application's possible configurations pre-defined as a part of the service component substitution and graph transformation. Hence, the considered number of feasible configurations is bounded.

The **Pre-requisite System Component Translation** determines the appropriate set of *middleware service components*, including their interfaces and QoS parameters. For each symbolic QoS configuration, the included middleware service components can be classified into two groups: (1) QoS-specific components, such as CORBA messaging, control and management of audio/video streams in CORBA, which provide service quality for specific QoS requirements, and (2) QoS-generic components, such as observer and resource adaptor in Agi-

los [18], DSRT [19], which perform generic functionality as adaptations, system resource management and monitoring.

This aspect of translation is based on an extensible *translation graph* which represents the association between specific application QoS categories and their suitable middleware service components, and *translation schemes* which represent the mappings between specific QoS categories' dimensions and middleware service components' interfaces and parameters. The input application QoS categories and their dimensions are retrieved from the specific service components in each symbolic QoS configuration. The pre-requisite system component translation can be considered as the refinement of each symbolic QoS configuration with the *vertical end-system configuration translation* from specific service components' QoS requirements in application layer into service components in middleware layer, and the *horizontal end-to-end configuration translation* among the determined middleware service components. The results of the pre-requisite system component translation are (1) the *pre-requisite system configuration files*, containing the associated middleware service component(s) for the symbolic QoS configuration per target node, and (2) the *mapping result files*, associated with individual pre-requisite system configuration files, containing the mapping result between specific service components' QoS category's dimensions and middleware service components' interfaces and parameters.

Note that how these two results will be used by whom depends on the involving middleware service components. In this paper, we consider two types of the middleware service components: (type I) components (e.g., CORBA messaging, DSRT) which need the instrumentation of application specific service components' source codes to include their pre-defined APIs, and (type II) components (e.g., pluggable protocol) which do not need any modification of the application specific service components' source codes. For type I, the compiled results will be interactively returned as hints to the application developer. There hints will indicate modification places in the specific service components' source codes to include the middleware service components' APIs, corresponding to the mapping information in the mapping result files. For type II, the compiled results will be used directly by the run-time middleware to dynamically link the required middleware service component(s) with their suitable parameters into the application execution environment.

This phase of compilation enables that different QoS-aware applications are executed uniformly in the *same* reconfigurable, component-based run-time middleware. It means that this phase represents the second level of QoS management unification as discussed in the Introduction.

The **Resource Translation** translates each symbolic QoS configuration and its associated middleware service component(s) into distributed system resource requirements. It is dealing with coordination among distributed resource brokers, resource negotiation, and resource translation. This aspect of translation is based on the *analytical translations*, and the *distributed probing and profiling techniques*. The analytical translation consists of the mapping or translation functions from specific QoS dimensions in the application layer and/or middle-

ware layer into system resource requirements. The analytical translation will be used if there exists a suitable function. The distributed probing protocol, assisted by the run-time middleware, (1) instantiates the specific application service components and their associated middleware service components into the distributed target nodes, (2) coordinates with distributed resource brokers to perform the system resource probing, (3) collects the probing results, and (4) associates them with the previously compiled results. The resource translation can be considered as the *vertical resource translation* from QoS requirements in the associated upper layers into the system resource requirements at the end-system, and the *horizontal end-to-end resource translation* among the distributed resource brokers. The results of the resource translation are the minimum end-to-end system multi-resource requirements for each symbolic QoS configuration, associated with specific middleware service components, and specific QoS requirements.

The application's compiled results are represented as the *QoS-aware Component-based Application Specification (QoSCASpec)*, which is the Q-Compiler's *"object code"*. QoSCASpec includes (1) application description; (2) set of *QoS configurations*; and (3) service component description for all specific service components in the symbolic QoS configurations. QoSCASpec is installed in the QoSCASpec repository as a "ready-to-use" configuration information for end-to-end QoS setup and adaptation of an application.

Figure 8 demonstrates the Q-Compiler's multi-aspect QoS translations and the QoSCASpecs for two different applications: (a) the library information retrieval application, and (b) the video on demand application. Note that in our implementation, the QoSCASpecs are represented by the XML-based description. When the applications' QoSCASpecs are available, the applications are ready for execution by the distributed run-time middleware.

4 Run-Time $2K^Q$ Middleware

In this section, we present the component-based and reconfigurable run-time $2K^Q$ middleware, which plays major role during application execution.

4.1 Architecture

The run-time $2K^Q$ middleware (shown in Fig. 9) consists of three planes: the *application execution plane*, the *middleware execution plane*, and the *middleware management plane*.

In the **application execution plane**, the *application run-time container* is the place where the application service components, compiled with required middleware service components' APIs and their necessary libraries, are instantiated and running. There exists one application run-time container per application.

In the **middleware execution plane**, the *middleware service component container* is the place where the application-neutral middleware service components (e.g., resource brokers [19], observer and resource adaptor [18]), are instantiated and running. There exists one middleware service component container

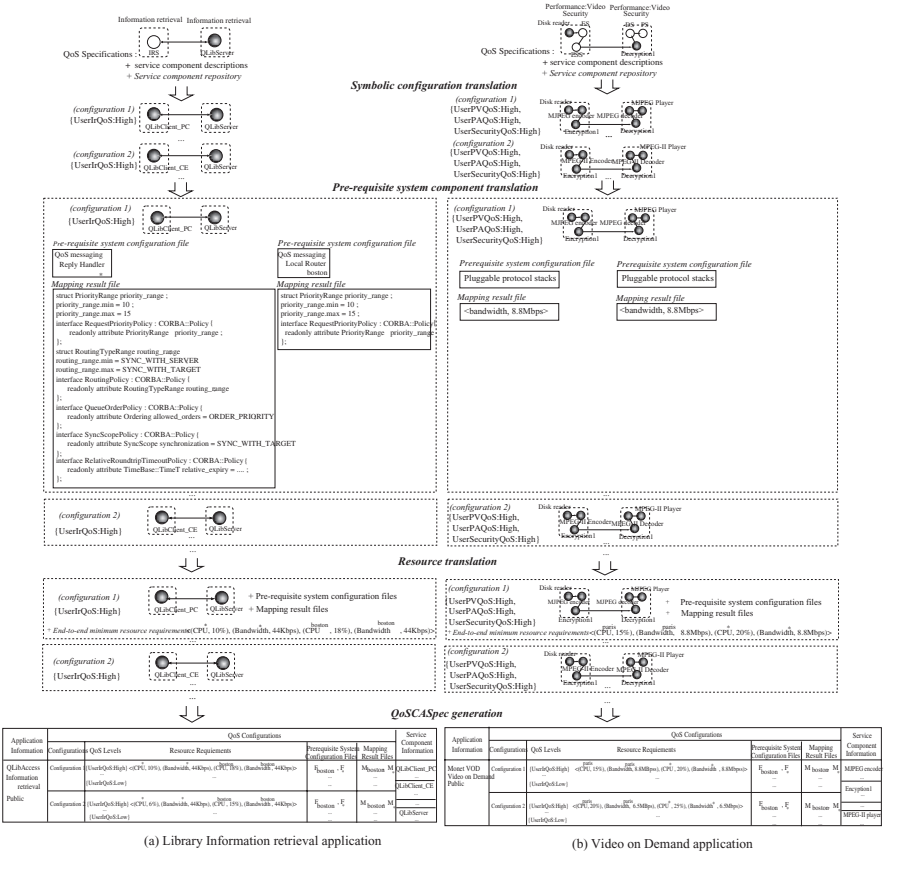


Fig. 8. Multi-aspect QoS Translations of Two Different Applications (Example)

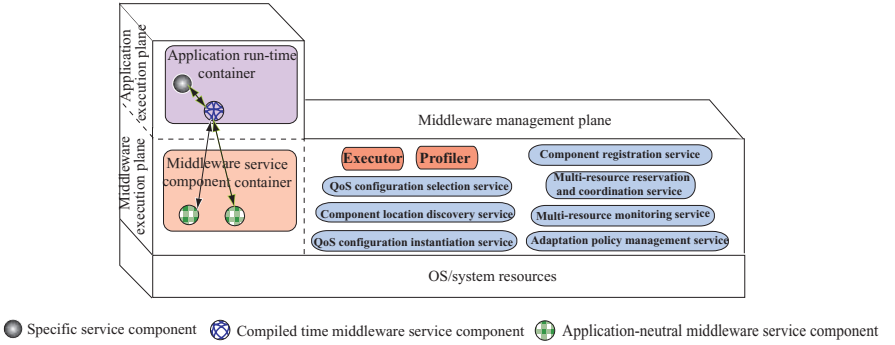


Fig. 9. Run-time $2K^Q$ Middleware Architecture

per machine and is shared among different applications. Both containers are dynamically created, managed and controlled by the middleware management plane.

The **middleware management plane** provides a set of middleware management services which help to instantiate, manage, and control an application with the quality of service provision. Specifically, it needs to provide support for (1) *QoS setup* of the application corresponding to user QoS request[6], and (2) *QoS adaptations* of the application in case of changes in user preference, and dynamic constraints such as resource availability, execution environment and mobility. The goal is to maintain the QoS provision or gracefully degrade QoS during the application run-time if changes occur.

The *profiler* and *executor* are the front-end agents of the run-time $2K^{Q}$ middleware waiting for probing service requests from the Q-Compiler's resource translation, and for user QoS requests from end-users for the application execution, respectively. They handle the requests using the provided services in the middleware management plane.

The middleware management plane composes of the following services: (1) *QoS configuration selection service*; (2) *component location discovery service*; (3) *QoS configuration instantiation service*; (4) *component registration service* ;(5) *multi-resource reservation service*, (6) *multi-resource monitoring service*, and (7) *adaptation policy management service.*

The *QoS configuration selection service* consults the *QoSCASpec repository* to get all possible QoS configurations corresponding to the user QoS request for an application. The selection service chooses the best configuration among the returned QoS configurations based on the current available resources, and execution environment (e.g., it performs the match between the current available resources and the configurations' system resource requirements). If the chosen configuration consists of a component with undefined location (target node), the *component location discovery service* will be activated to discover (i.e., contacting the public domain of running components) the component's best location.

The *QoS configuration instantiation service* is responsible to instantiate specific service components and their associated middleware service components defined in the pre-requisite system configuration files[7] into the distributed locations (target nodes). The instantiation services in the distributed locations coordinate among themselves to (1) dynamically create the application run-time containers and the middleware service component containers in the distributed locations,(2) dynamically download the service component(s) from a service component repository if the required specific or middleware service component(s) are

[6] For example, an end-user provides a user request as follows: [<application name = "MONETVoD">, <application category = "Video on Demand">, <QoS level = [High:UserPVQoSHigh, UserPAQoS:High, UserSecQoS:High]>, <accessibility = "Public">].

[7] The configuration files are accompanied with corresponding mapping result files (sec Sect. 3) and the mapping result files will be used by the run-time middleware only if the associated middleware service components do not need any instrumentation of the specific service components' source codes.

not located in the specified locations, and (3) instantiate the service components in these containers. These steps will be performed only if no instance of the required service component is running on a particular target node. Note that when a service component is instantiated, it may advertise itself to a public domain of running components via the *component registration service*.

The *multi-resource reservation and coordination service*, based on the end-to-end run-time multi-resource negotiation, is activated after the QoS configuration instantiation service if the involving distributed locations support the reservation of resources. The end-to-end resource negotiation is based on the minimum resource requirement information compiled during the application development phase.

The *multi-resource monitoring service* is responsible to measure and gather, via the resource brokers, current available resources and service components' resource requirements, at the end-system and in the distributed locations. The returned result from the resource monitoring service can be used as hint for selecting the most suitable configuration during the QoS configuration selection service, and the most suitable QoS adaptation during the application run-time. During the application development phase, the Q-Compiler's resource translation uses this service via the profiler to measure the minimum end-to-end multi-resource requirements for each specific configuration.

The *adaptation policy management service* performs the setup of adaptation policies which include the possible QoS configurations and their transitions, corresponding to user preferences encoded in the user QoS request, and application developer's choices specified during application development phase. The setup of the adaptation policies can be considered as part of the QoS setup. It helps the run-time middleware to manage the adaptations of an application appropriately during the application run-time.

While we present the middleware management plane as the composition of seven necessary middleware management services to manage and control the application execution plane, the middleware management plane is configurable and extensible by additional services. Moreover, the middleware management plane on different machines with varied capacities (e.g., high performance PCs, handheld PDAs) and environments (e.g., reservation-enabled, or best-effort) can be dynamically customized. For example, the middleware management plane on the handheld PDAs can be configured to include only partial services or even no services, and to rely on a gateway with fully-support middleware management plane.

In the following subsections, we describe how the run-time $2K^Q$ middleware assists the Q-Compiler's resource translation during the application development phase, and how it manages and controls the execution of an application during the application execution phase.

4.2 Run-Time $2K^Q$ Middleware and Q-Compiler's Resource Translation

The run-time $2K^Q$ middleware performs the following steps (see Fig. 10) to assist the Q-Compiler's resource translation measuring the distributed multi-resource requirements for a specific configuration[8]. Step 1: the profiler gets the resource translation's distributed probing request. Step 2: the profiler activates the QoS configuration instantiation service to collaboratively create the application run-time containers and the middleware service component containers in the distributed locations, corresponding to the specific configuration associated with the distributed probing request. Step 3: the instantiation services in the distributed locations, then, dynamically download the required service components from the service component repository, and instantiate them into the created containers. Step 4: the profiler activates the resource monitoring service to collaboratively gather the distributed multi-resource requirements for the configuration. Step 5: the profiler returns the minimum end-to-end multi-resource requirements of the configuration to the Q-Compiler's resource translation.

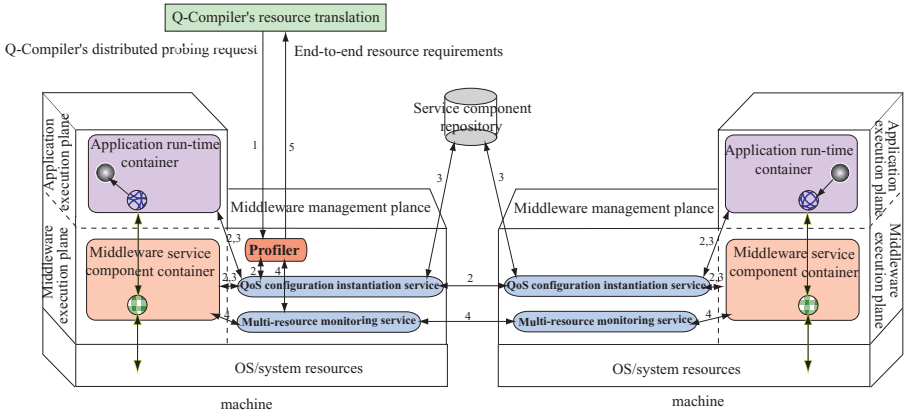


Fig. 10. Run-time $2K^Q$ Middleware Assisting Q-Compiler's Resource Translation

4.3 Run-Time $2K^Q$ Middleware and QoS-Aware Application Execution

In this section, we describe how the run-time $2K^Q$ middleware instantiates, manages, and controls an application execution corresponding to a user QoS request, the application's QoSCASpec, and the dynamic run-time constraints such as current resource availability, and execution environment.

[8] Note that the individual steps in this paragraph correspond to the steps in Fig. 10.

The run-time $2K^Q$ middleware performs the following steps[9] (see Fig. 11). Step 1: the executor gets a user QoS request. Step 2: the executor activates the QoS configuration selection service to find the most suitable configuration, which will be determined based on (1) the application's compiled result getting from the QoSCASpec repository (see Step 2.1 in Fig. 11), and (2) the current available resources getting from the multi-resource monitoring service (see Step 2.2 in Fig. 11). The adaptation policy management service performs the setup of the adaptation policies among different possible configurations getting from the QoSCASpec repository (see Step 2.3 in Fig. 11). The component location discovery service will be activated in Step 3 if there exists an undefined location in the selected configuration from Step 2. Step 4: the QoS configuration selection service returns the most suitable configuration to the executor. Step 5: the executor activates the QoS configuration instantiation service to collaboratively create the application run-time containers in the distributed locations, according to the specified information in the returned configuration. Step 6: the instantiation services in the distributed locations dynamically download the required service components from the service component repository, and instantiate them into the created containers. Note that in Step 6, when a service component is instantiated, it may also register itself to a public domain of running components via the component registration service. In Fig. 11, we assume that the middleware service component containers are already available in the distributed locations. Step 7: the executor activates the multi-resource reservation and coordination service to coordinate the reservation of resources in the distributed locations for the end-to-end configuration. The resource reservation service will be activated only if the involving distributed locations support the reservation model. Step 8: the executor returns the execution result to the end-user.

Because the run-time $2K^Q$ middleware is component-based and reconfigurable, and allows for dynamic downloading and linking of service components into distributed target nodes securely, it can execute and ensure the QoS provisions for different types of QoS-aware applications uniformly, based on the compiled meta information in applications' *QoSCASpec*s.

5 Implementation and Experiments

The implementation is divided into two parts: (1) the *uniform QoS programming environment* and the *Q-Compiler's multi-aspects QoS translations* are implemented in Java, and integrated with the visual programming environment [20]; (2) the distributed *run-time* $2K^Q$ *middleware* is implemented as a set of CORBA components, based on the dynamic reconfigurable middleware "dynamicTAO" [21], and the resource reservation model in the QualMan system [22].

The run-time $2K^Q$ nodes are connected via a 100 Mbps Ethernet[10]. The nodes are: (1) two Sun Ultra-60 workstations, each with two 360 MHz processors and 768 MB memory, (2) one Sun Ultra-60 workstation with two 450 MHz

[9] Note that the individual steps in this paragraph correspond to the steps in Fig. 11.
[10] We are working towards heterogeneous networks with different devices' capabilities.

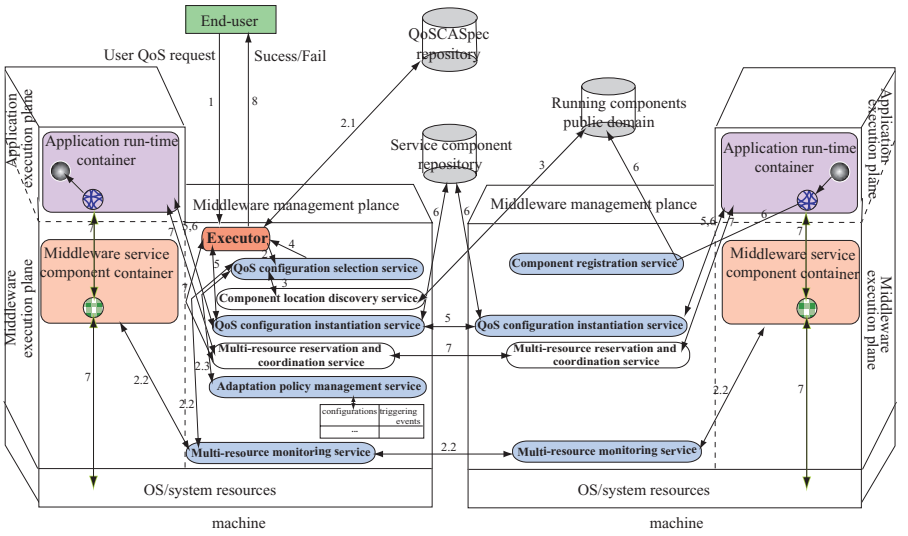


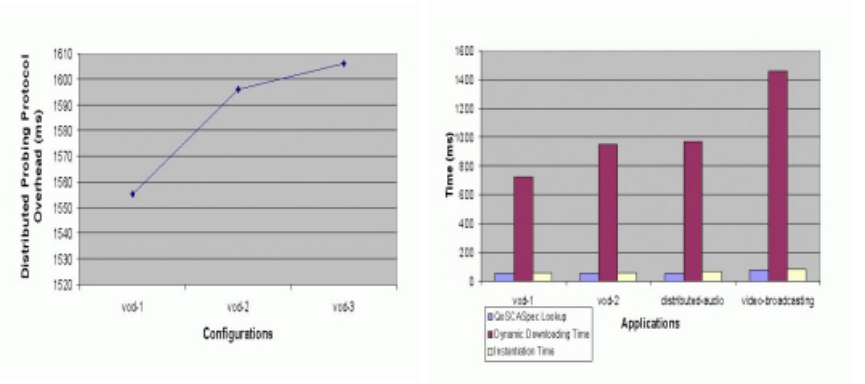
Fig. 11. Run-time $2K^{Q}$ Middleware Managing QoS-aware Application Execution

processors and 1024 MB memory. All machines are running SunOS 5.7. The file system is NFS. The Q-Compiler runs on Pentium-II PC with one 366 MHz processor and 128 MB memory, and connects to the run-time $2K^{Q}$ nodes via the 100 Mbps Ethernet.

We show our first results of the unified QoS management framework's performance via the *overhead of the resource translation's* distributed probing protocol (see Fig. 12(a)), and the *overhead of the QoS setup* (see Fig. 12(b)) for different applications.

In Fig. 12(a), vod-1, vod-2 and vod-3 are three different configurations of a video on demand application: (1) a single server with a single client, (2) a single server with two clients, and (3) a single server with three clients. The graph is an average of fifteen runs for the three configurations. The results show that the average resource translation times, based on the *distributed probing protocol*, are 1555.1, 1596.1, and 1606.4 ms, respectively. Times vary corresponding to the number of associated service components needed to be instantiated in the configurations, service components' sizes and target locations. Note that the measurements do not include the duration of individual service components' local resource probing.

In Fig. 12(b), we measure the *QoS Setup* time of four applications: vod-1, vod-2, distributed audio (distributed-audio) with three audio players, and video broadcasting (video-broadcasting) with a server, a gateway, and two receivers. The QoS setup time includes QoSCASpec lookup, dynamic downloading of service components and component instantiation. The graph is an average of fifteen runs for the four applications. The results show that the QoS setup times for the four applications vary mainly according to their dynamic downloading times,



(a) (b)

Fig. 12. (a) Resource Translation Times for Different Configurations During the Application Development Phase, (b) QoS Setup Times for Different Applications During the Application Execution Phase

which are 723.0, 945.5, 964.5, and 1458.7 ms, respectively. Dynamic downloading times vary corresponding to the number of associated service components in the configurations, the service components' sizes and the target locations. The average QoSCASpec lookup times for the four applications are 53.3, 54.7, 48.9, and 72.3 ms, respectively. The QoSCASpec lookup is stable and very small because the current QoSCASpec repository is running on the same machine as the executor getting the user QoS request. Also, in these experiments, the QoSCASpec lookup performs only the configuration matching without considering additional constraints. The average component instantiation times for the four applications are 55.3, 59.3, 66.1, and 82.1 ms respectively. The component instantiation time is the time used in CORBA method invocations to start the distributed service components forming the application. If we assume that service components (some or all) in a configuration are already in places, the QoS setup time for the configuration will be very small.

6 Related Work

Extensive related work exists in different areas of the QoS:

– **QoS specifications and QoS translations**
QoS specifications, proposed as part of QoS architectures or QoS middleware architectures, depend on the design and objectives of the architectures. For example, in QuO project [14], QoS is specified via a suite of description languages based on aspect-oriented programming [23]. In QoSME [7], QoS is described via a Quality of Service Assurance Language (QuAL). In CORBA, QoS is specified via pre-defined interfaces of different QoS extensions such as control and management of audio/video streams in CORBA [9], CORBA

messaging [10], fault tolerant CORBA [11], and real-time CORBA [12]. In Agilos middleware [18], QoS is defined via rules and membership functions. In Q-RAM project [6], QoS is represented via utility functions.
In the area of *QoS Translation*, QoS translations, based on analytical functions, are proposed. For example, in [2] Nahrstedt et al. propose a translation from a multimedia application's QoS parameters into transport subsystem's QoS parameters, and in [24] Kim et al. propose a translation from MPEG video parameters into CPU requirements. The proposed analytical translations are application-specific and applied to multimedia domain. Moreover, they deal only with the translation between the application QoS parameters and the system resource requirements.

– **QoS Architectures**
Several reservation-based QoS architectures and approaches already exist. For example, in [2,3,4,5] researchers propose end-to-end QoS management frameworks for multimedia applications. In the best-effort environment, the adaptation-based QoS architectures exist. For example, Agilos [18] proposes a framework assisting in QoS enforcement for a distributed visual tracking application. Q-RAM project [6] proposes a QoS management framework based on the multi-resource allocation model mainly focusing on the global resource optimization. QoSME [7] with the Quality of Service Assurance Language (QuAL) provides the abstractions for QoS management to the underlying network management. These QoS architectures and approaches are designed only for one type of applications, or to handle particular aspect of QoS provisions such as the resource management. If one wants to run a different type of applications, the existing QoS architectures do not scale, i.e., might not be applicable, or reusable.
– **QoS in Distributed Object Computing (DOC) Middleware**
In *distributed object computing (DOC) middleware* such as CORBA, control and management of audio/video streams in CORBA, CORBA messaging, fault tolerant CORBA and real-time CORBA [9,10,11,12] are proposed to provide quality of service for different types of applications. In this case, an application will be QoS-aware if the application developer deploys these QoS-related interfaces. Hence, an application developer has to learn different IDL interfaces for different types of provisions. In addition, the application developer has to know the semantics of these interfaces, and how to translate his/her application QoS requirements into these interfaces and their parameters appropriately. For example, in real-time CORBA, an interface allows the application developer to specify the protocol configuration such as protocol type, ORB protocol property, and transport protocol property. What is not clear is how an application's QoS specifications should be translated appropriately into these IDL interfaces and parameters. Other QoS efforts in DOC middleware are the optimizations of ORBs, such as TAO project [13] at Washington university to support the real-time messaging. BBN's Quality Object (QuO) project [14] allows an application developer to develop distributed applications that can adapt to the changing quality of service in CORBA environment. Lancaster's multimedia component architecture [25]

is extended beyond CORBA or DCOM and takes application QoS parameters into account. Adapt project [26] allows explicit bindings in CORBA via open bindings. Qualities of services in DOC middleware are also tailored toward specific type of applications or particular aspect of QoS provisions.

7 Conclusions

Different domains of distributed component-based applications, running in heterogeneous execution environments, need different quality of service semantics. It is hard to provide quality of service for individual applications, and even harder to handle them uniformly in a QoS management framework. The difficulty is in both the development phase, and the execution phase to integrate the QoS as part of the application.

In this paper, we describe the architecture and the philosophy of a unified QoS management framework, $2K^{Q+}$, based on the integrated approach of the QoS compilation and the component-based and reconfigurable run-time middleware. The framework provides uniform and systematic mechanisms for developing a QoS-aware application during the application development phase, and for ensuring QoS provisions based on the compiled information and dynamic run-time constraints, during the application execution time.

While the component-based middleware concept by itself is not novel, (1) the introduction of an automated QoS compilation concept, which helps the application developer to decide the possible configurations and the appropriate set of middleware components running in heterogeneous execution environments, and (2) the integration of these two concepts, forming the unified QoS management framework, are novel.

We believe that $2K^{Q+}$ is a practical solution for a unified QoS management framework, which includes not only QoS setup and QoS provision during the application run-time, but also QoS programming and compilation for different applications during the application development.

References

[1] K. Nahrstedt, D. Wichadakul, and D. Xu. Distributed qos compilation and runtime instantiation. *In Proceedings of the Eighth IEEE/IFIP International Workshop on Quality of Service*, pages 198–207, June 2000.

[2] K. Nahrstedt and J. Smith. Design, implementation and experiences with the omega end-point architecture. *IEEE Journal on Selected Areas in Communication*, 14(7):1263–1279, September 1996.

[3] A. Campbell, G. Coulson, and D. Hutchison. A quality of service architecture. *Computer Communication Review*, 24(2):6–27, April 1994.

[4] L. C. Wolf. *Resource Management for Distributed Multimedia Systems.* Kluwer, Boston, Dordrecht, London, 1996.

[5] A. Hafid and G. Bochmann. An approach to qos management in distributed multimedia applications: Design and an implementation. *Multimedia Tools and Applications*, 9(2), 1999.

[6] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. *In Proceedings of the IEEE Real-Time Systems Symposium*, pages 298–307, December 1997.

[7] P. G. S. Florissi. *QoSME: QoS Management Environment.* PhD thesis, Columbia University, Department of Computer Science, 1996.

[8] K. Nahrstedt, H. Chu, and S. Narayan. Qos-aware resource management for distributed multimedia applications. *Journal on High-Speed Networking, Special Issue on Multimedia Networking, IOS Press*, 8(3-4):227–255, 1998.

[9] IONA Technologies Plc., Lucent Technologies Inc., and AG Siemens-Nixdorf. Control and management of audio/video streams omg rfp submission. *online documentation at http://www.omg.org/docs/telecom/98-10-5.doc*, May 1998.

[10] BEA Systems Inc., Expersoft Corporation, Imprise Corporation, International Business Machine Corporation, International Computers Ltd., IONA Technologies Plc., Northern Telecom Corpoaration, Novell Inc., Oracle Corporation, Peerlogic Inc., and TIBCO Inc. Corba messaging. *online documentation at http://www.omg.org/cgi-bin/doc?orbos/98-05-05.*, May 1998.

[11] Ericsson, Eternal Systems Inc., HighComm, Inprise Corporation, IONA Technologies Plc., Lockheed Martin Corporation, Lucent Technologies, Objective Interface Systems Inc., Oracle Corporation, and Sun Microsystems Inc. Fault tolerant corba, joint revised submission. *online documentation at http://www.omg.org/techprocess/meetings/schedule/Fault_Tolerance_RFP.html*, December 1999.

[12] Alcatel, Hewlett-Packard Company, Highlander Communications L.C., Inprise Corporation, IONA Technologies, Lockheed Martin Federal systems Inc., Lucent Technologies Inc., Nortel Networks, Objective Interface Systems Inc., Object-Oriented Concepts Inc., Sun Microsystems Inc., and Tri-Pacific Software Inc. Real-time corba, joint revised submission. *online documentation at http://www.omg.org/cgi-bin/doc?orbos/99-02-12*, March 1999.

[13] D. Schmidt, D.Levine, and C. Cleeland. *Advances in Computers, Marvin Zelkowitz (editor)*, chapter Architectures and Patterns for High-performance, Real-time ORB Endsystems. Academic Press, 1999.

[14] J. Zinky, D. Bakken, and R. Schantz. Architecture support for quality of service for corba objects. *Theory and Practice of Object Systems*, 3(1):55–73, January 1997.

[15] G. Bochmann, B. Kerherve, and M. Mohamed-Salem. Quality of service management issues in electronic commerce applications. *to be published as a chapter in a book.*

[16] S. Frolund and J. Koistinen. Quality of service specification in distributed object systems design. *In Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, pages 1–18, 1998.

[17] D. Wichadakul and K. Nahrstedt. Distributed qos compiler. *Technical Report UIUCDCS-R-2001-2201 UILU-ENG-2001-1705, Department of Computer Science, University of Illinois at Urbana-Champaign, (submitted for journal publication)*, February 2001.

[18] B. Li and K. Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, 17(9):1632–1650, September 1999.

[19] H. Chu and K. Nahrstedt. Cpu service classes for multimedia applications. *In Proceedings IEEE International Conference on Multimedia Computing and Systems*, pages 296–301, June 1999.

[20] X. Gu, D. Wichadakul, and K. Nahrstedt. Visual qos programming environment for ubiquitous multimedia services. *to appear in Proceedings of IEEE International Conference on Multimedia and Expo*, August 2001.

[21] M. Roman, F. Kon, and R.H. Campbell. Design and implementation of runtime reflection in communication middleware: the dynamictao case. *In Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*, pages 122–127, June 1999.

[22] K. Nahrstedt, H. Chu, and S. Narayan. Qos-aware resource management for distributed multimedia applications. *Journal of High-Speed Networks, Special Issue on Multimedia Networking*, 7(3-4):229–257, 1998.

[23] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. Qos aspect languages and their runtime integration. *In Lecture Notes in Computer Science, Springer-Verlag of the Fourth International Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 1511:303–318, May 1998.

[24] K. Kim and K. Nahrstedt. *Building QoS into Distributed Systems, Andrew Campbell, Klara Nahrstedt (editors)*, chapter QoS Translation and Admission Control for MPEG Video, pages 359–362. Chapman and Hall, 1997.

[25] D. G. Waddington and G. Coulson. A distributed multimedia component architecture. *In Proceedings of the First International Enterprise Distributed Object Computing Workshop*, pages 337–345, October 1997.

[26] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. *In Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 128–135, May 1998.

Author Index