Lecture Notes in Computer Science 2847
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Rogério de Lemos
Taisy Silva Weber
João Batista Camargo Jr. (Eds.)

Dependable Computing

First Latin-American Symposium, LADC 2003
São Paulo, Brazil, October 21-24, 2003
Proceedings

Springer

Volume Editors

Rogério de Lemos
University of Kent, Computing Laboratory
Canterbury, Kent CT2 7NF, UK
E-mail: r.delemos@ukc.ac.uk

Taisy Silva Weber
Universidade Federal do Rio Grande do Sul, Instituto de Informática
Caixa Postal 15064, 91501-970 Porto Alegre - RS, Brazil
E-mail: taisy@inf.ufrgs.br

João Batista Camargo Jr.
Universidade de São Paulo, Escola Politécnica
Travessa 3, Número 158, Edifício de Engenharia Eletricidade
05508-900 São Paulo - SP, Brazil
E-mail: joao.camargo@poli.usp.br

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

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

CR Subject Classification (1998): C.3, C.4, B.1.3, B.2.3, B.3.4, B.4.5, D.2.4, D.2.8, D.4.5, E.4, J.7

ISSN 0302-9743
ISBN 3-540-20224-2 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP Berlin GmbH
Printed on acid-free paper SPIN: 10959947 06/3142 5 4 3 2 1 0

Preface

This is the first edition of the Latin American Symposium on Dependable Computing (proceedings of the LADC). LADC is the sole Latin American event dedicated to discussing the many issues related to computer system dependability. This symposium succeeded the well established Brazilian Symposium on Fault Tolerant Computers, which was a biennial event that lasted for 20 years.

Although the symposium was based in Latin America, the intention was to attract researchers from all over the world. There were 46 paper submissions from Europe, and South and North America. The selection process was rigorous, with each manuscript being sent out for review to four Program Committee members. A total of 19 papers were selected to be included in the proceedings, of which two were experience reports and three were short papers. In addition to these papers, we also invited Henrique Madeira and Eliane Martins to contribute with two papers, and we were very grateful that they accepted our invitation.

LADC 2003 was privileged to have a very prestigious and dedicated Program Committee, that embraced an electronic reviewing process that was demanding and time consuming. We would like to thank its members for their dedication and effort in helping to put together the final program. The electronic submission and reviewing process was only possible due to the efforts of Alan Cleber Borim and Lisandro Zambenedetti Granville in installing and maintaining the EDAS conference manager. As part of the technical program, we also included three invited talks and a panel. Again, we are very grateful to Karama Kanoun, Eno Siewerdt, and Aad van Moorsel for agreeing to deliver invited talks at the symposium, and to João Gabriel Silva for organizing and chairing the panel on dependability benchmarking.

We would also like to thank the LADC Steering Committee for their advice and support. Our special thanks goes to João Batista Camargo Jr., our enthusiastic and supportive General Chair who put so much effort into the organization of a successful LADC 2003. We are also thankful to Springer-Verlag for publishing the symposium proceedings in the prestigious series Lecture Notes in Computer Science.

Finally, we hope that these LADC 2003 proceedings will enlighten and deepen the knowledge of the symposium participants and future readers.

August 2003

Rogério de Lemos
Taisy Silva Weber

Organizing Committee

General Chair

João Batista Camargo Jr.
USP, Brazil

Program Co-chairs

Rogério de Lemos
University of Kent, UK

Taisy Silva Weber
UFRGS, Brazil

Local Arrangements Chair

Sérgio Miranda Paz
USP, Brazil

Publicity Chair

Jorge Rady de Almeida Jr.
USP, Brazil

Finance Chair

Paulo Sérgio Cugnasca
USP, Brazil

Publication Chair

Lucia Vilela Leite Filgueiras
USP, Brazil

Registration Chair

Ricardo Alexandre Veiga Gimenes
USP, Brazil

Lúcio Flávio Vismari
USP, Brazil

Tutorials Co-chairs

Marinho P. Barcellos
UNISINOS, Brazil

Francisco Vilar Brasileiro
UFCG, Brazil

LADC Steering Committee

Francisco Vilar Brasileiro, Brazil
Rogério de Lemos, UK
Joni da Silva Fraga, Brazil
Raimundo Macêdo, Brazil
Eliane Martins, Brazil (Chair)
Carlos A. Maziero, Brazil
Taisy Silva Weber, Brazil

Program Committee

Jean Arlat, France
Marinho P. Barcellos, Brazil
Andrea Bondavalli, Italy
Luiz Eduardo Buzato, Brazil
Paulo Lício de Geus, Brazil
Elias P. Duarte Jr., Brazil
Markus Endler, Brazil
Paul Ezhilchelvan, UK
Gerhard Fohler, Sweden
Joni da Silva Fraga, Brazil
Pedro Gil, Spain
Roberto Gómez, Mexico
John Harauz, Canada
Ingrid Jansch-Pôrto, Brazil
Valérie Issarny, France
Kane Kim, USA
Jean-Claude Laprie, France
Raimundo Macêdo, Brazil
José Carlos Maldonado, Brazil
Eliane Martins, Brazil
Carlos A. Maziero, Brazil
Selma S.S. Melnikoff, Brazil
Fabio Panzieri, Italy
Sergio Rajsbaum, Mexico
Michel Raynal, France
Cecília M.F. Rubira, Brazil
William H. Sanders, USA
João Gabriel Silva, Portugal
Paulo Veríssimo, Portugal
Mladen A. Vouk, USA
Raul Weber, Brazil
Avelino Zorzo, Brazil

External Referees

Richard Achmatowicz
Nick Cook
Miguel Correia
Rômulo Silva de Oliveira
Egon Hilgenstieler
Andreá Luiz Pires Guedes
Andreas Kiefer
Graham Morgan
Bogdan Tomoyuki Nassu
Nuno M. Neves
José Rufino
Juan Carlos Ruiz-Garca
Frank Siqueira
Paulo Sousa

Organizer

University of São Paulo, Polytechnic School

Sponsor

Brazilian Computing Society (SBC)

In Co-operation with

Argentine Society for Informatics and Operations Research (SADIO)
Chilean Computer Science Society (SCCC)
IEEE Computer Society TC on Fault-Tolerant Computing
IFIP Working Group 10.4 “Dependable Computing and Fault-Tolerance”
European Workshop on Industrial Computer Systems – Reliability, Safety and Security (EWICS)

Table of Contents

Adaptive Fault Tolerance

Distributed Algorithms

Components and Fault Tolerance

Panel

Workshops

Tutorials

Foreword

The 1st Latin American Symposium on Dependable Computing (LADC 2003) is the sole event in Latin America dedicated to discussing the many issues related to computer system dependability. This symposium succeeded the well established Brazilian Symposium on Fault Tolerant Computers (SCTF), which for almost two decades was the main Brazilian forum for exchanging state-of-the-art research and development in dependable computing. The objective of LADC 2003 was to provide a forum for international and Latin American researchers and practitioners to present their latest results and experiences in this very dynamic field.

The LADC 2003 program presented keynote talks from top international experts in the area, technical sessions, a panel, tutorials, and two workshops: an industrial-oriented workshop for discussing the application of computers in critical systems, and a students' forum for discussing their latest research.

LADC 2003 was sponsored by the Brazilian Computing Society (SBC), and organized in co-operation with the Argentine Society for Informatics and Operations Research (SADIO), Chilean Computer Science Society (SCCC), IEEE TC on Fault-Tolerant Computing, IFIP Working Group 10.4 "Dependable Computing and Fault-Tolerance," and the European Workshop on Industrial Computer Systems – Reliability, Safety and Security (EWICS).

I would like to acknowledge the financial support received from the following LADC 2003 partners: FAPESP, CNPq, CAPES, FINEP, FDTE, Banco Real ABN AMRO, Scopus Tecnologia S.A., and Microsoft Corporation.

I would like also to thank the LADC 2003 Organizing Committee for their support in helping me with the symposium organization, especially Jorge Rady de Almeida Jr., Paulo Sérgio Cugnasca, Lúcio Flávio Vismari, Ricardo Alexandre Veiga Gimenes, Alan Cleber Borim and Lisandro Zambenedetti Granville. Another special thanks goes to Rogério de Lemos and Taisy Silva Weber, the Program Co-chairs, who exhaustively worked for a successful LADC 2003.

LADC 2003 was held in São Paulo, the major financial and business center of Latin America, responsible for half of Brazil's GDP. São Paulo is the capital of fine cuisine, leisure and culture and is also the financial and commercial center of Brazil, offering visitors a great variety of attractions. I am sure that all participants appreciated and enjoyed LADC 2003 in São Paulo.

August 2003 João Batista Camargo Jr.

Dependability Benchmarking: How Far Are We?

Karama Kanoun

LAAS-CNRS
7 avenue du Colonel Roche, 31077 Toulouse Cedex 4 — France
Karama.Kanoun@laas.fr

Extended Abstract

System performance is no longer the only factor that keeps customers satisfied. Dependability is increasingly playing a determinant role, as well. Implementing a well-planned benchmarking program is essential for competition. However, while benchmarking is widely used to measure computer performance in a deterministic and reproducible manner, allowing users to appreciate the capacity of their computer systems properly, dependability benchmarking is hardly emerging. Indeed, despite the broad range of promising work on dependability benchmarking, carried out by different organizations and research groups in the last decade, we are far from having reached the consensus needed to get to the current situation of performance benchmarks.

The definition of an appropriate workload, representing a synthetic or a realistic operational profile, is a common feature to performance and dependability benchmarks. The selection of such a workload constitutes the main difficulty of performance benchmarks as an agreement between system vendors and potential users should be reached to expect recognition and adoption of any performance benchmark.

Dependability benchmarks characterize the system behavior in the presence of "faults", or under "abnormal" conditions or perturbations. Thus, in addition to the definition of an appropriate workload, they require the definition of suitable "perturbations". This is not an easy task since such perturbations have to simulate faults that could be internal or external to the system being benchmarked. Additionally, several benchmarking measures may be of interest. Furthermore, measures may characterize the system in a comprehensive way (with respect to the service delivered) or characterize specific features of the system (e.g., fault tolerance mechanisms). This adds an extra complexity.

As a result, a great variety of benchmarks has been defined, e.g., robustness benchmarks, fault tolerance benchmarks, recovery benchmarks, competitive (or comprehensive) benchmarks, specific (or developer) benchmarks. Most of these dependability benchmarks are *ad hoc* benchmarks that have been defined in a non-coordinated way. Nevertheless, the work achieved so far is necessary to pave the way for standard dependability benchmarks.

In this keynote, an overview of the various approaches for dependability benchmarking will be presented and further research in this emerging area will be discussed, together with the challenges we face and the barriers to cross in order to reach the necessary agreement among the various players for the establishment of well-accepted dependability benchmarks.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, p. 1, 2003.

Safety-Critical Systems in Air Traffic Management

Eno Siewerdt

ATECH - Tecnologias Críticas
Aeroporto de Congonhas – Saguão Central/1° andar
Av. Washington Luís,s/n
04695-900 – São Paulo, SP, Brasil
{eno@atech.br}

Civil aviation, since the early 1950's, has increasingly become a significant contributor to the overall well-being and economic vitality of individual nations as well as to the world in general. One of the requirements of carrying out civil aviation activities is to ensure that a safe, secure, efficient, and environmentally sustainable air navigation system is available, at the global, regional and national levels. This requires the implementation of an air traffic management (ATM) system.

The current ATM system has limitations that result in inefficient aircraft operations, including requirements to fly circuitous departure and arrival procedures; exclusion of civil air traffic from airspace reserved for defense purposes; indirect fixed routes between destinations; excessive system related ground and en-route delays; operation of aircraft at inefficient altitudes, speeds, and in unfavorable winds; and insufficient flexibility to permit optimum management of weather-related disruptions to airline operations. Therefore, the evolution of Air Traffic Management (ATM) is based on relatively new concepts and technologically advanced systems. It is a portion of the Communications, Navigation, Surveillance/Air Traffic Management (CNS/ATM) systems, as developed by the special Committee on Future Air Navigation System (FANS). This Committee, established by the International Civil Aviation Organization, in 1983, to study alternatives to the technical shortcomings then afflicting the provision of air traffic services and compromising the safe growth of aviation, worldwide, initially focused on solutions for the technical CNS elements, but soon realized that these tools had to be put in an operational context. The resulting CNS/ATM Concept developed by FANS was endorsed by States and Aviation related Organizations in 1991, but implementing an ATM System that allows maximum use of enhanced capabilities provided by technical advances and high degrees of automation requires a thorough assessment and analysis of human capabilities and ATM infrastructure, where the attainment of a safe system has the highest priority, whilst achieving efficient and effective outcomes.

The ATM system is a system that provides air traffic management through the collaborative integration of humans, information, technology, facilities and services, supported by air, ground and/or space-based communications, navigation and surveillance. Clearly-defined requirements are the basis for the coordinated implementation of CNS/ATM technologies. Following the recognition that technology is not an end in itself, a comprehensive concept of an integrated and global ATM system has been developed by the Air Traffic Management Operational Concept Panel (ATMCP), established by ICAO. Several interdependent concept

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 2–3, 2003.

components are integrated to form the ATM system. They comprise Airspace Organization and Management, Aerodrome Operations, Demand and Capacity Balancing, Traffic Synchronization, Conflict Management, Airspace User Operations, and ATM Service Delivery Management. Information management is vital to the proper functioning of these components.

Information management is necessary for the provision of accredited quality-assured and timely information used to support ATM operations, will also monitor and control the quality of the shared information and provide information-sharing mechanisms that support the ATM community. Information management shall assemble the best possible integrated picture of the historical, real-time and planned or foreseen future state of the ATM situation, thus providing the basis for improved decision-making by all ATM community members. Key to the concept will be the management of an information-rich environment.

While safety is the highest priority, many other expectations for the global ATM system have been established, including equity for all airspace users that have access to a given airspace or service, greater capacity to meet airspace user demand at peak times and locations while minimizing restrictions on traffic flow, cost-effectiveness, efficiency, protection of the environment, flexibility, global interoperability, predictability, security, and a continuous involvement of the ATM community in the planning, implementation, and operation of the system.

In any current or future ATM system, human operators, including air traffic controllers, play a vital role. The air traffic controller's job consists of complex tasks demanding very special skill and active application of unique cognitive abilities, all of which may greatly benefit from up-to-date automation in form of advanced software, new display capabilities, conflict alert and resolution tools, inter alia. Along the whole process of implementing new, increasingly automated, ATM systems, devoting special attention to the guiding principles applied to the development of dependable computing systems is of paramount importance to assure the continuing safety of air navigation. Uniform safety standards and risk and safety management practices must be applied systematically to the ATM system. In implementing elements of the global aviation system, safety needs to be assessed against appropriate criteria, and in accordance with appropriate and globally standardized safety management processes and practices.

The system safety approach outlined in the ATM Operational Concept is holistic, applying across the spectrum of the ATM system, where the system will be considered to include people, procedures and technologies performing specific tasks in a given environment. This approach stems from the same reasons that ATM shares with many other safety-critical activities, thus providing that each element of the ATM system, wherever implemented (aircraft, ground, space, etc.) shall be subject to specific safety analysis, as an individual element, and as a component of the larger integrated system. The implementation of any element of the system shall be subject to appropriate safety assurance processes. Clear accountabilities for all aspects of safety, in particular those related to software dependable systems, must be defined, and the roles and responsibilities for the management and integration of system elements must be explicitly stated. Therefore, the importance of attracting more and more specialist developers of dependable computing systems to work with Air Traffic Management systems cannot be overestimated.

Managed Utility Computing: The Grid as Management Backplane

Vijay Machiraju, Akhil Sahai, and Aad van Moorsel

Hewlett-Packard Laboratories,
Palo Alto, California, USA
{vijaym,asahai,aad}@hpl.hp.com

Enterprise IT exhibits increasingly complex networked systems and distributed applications, making the task of an IT operator or administrator exceedingly difficult. We argue in the keynote associated with this paper for the necessity of a clean, standardized, service-centric software architecture to automate and facilitate operator tasks throughout the life-cycle of systems and applications. We refer to the proposed solution as 'managed utility computing,' since it enables utility computing while improving manageability. The proposed architecture is a web services based 'grid' architecture, with targeted management extensions. This short note argues that this architecture is also necessary and appropriate as backplane for traditional management software (irrespective of the utility computing context), to support increasingly complex management tasks for increasingly complex systems.

1 Management Software to Date

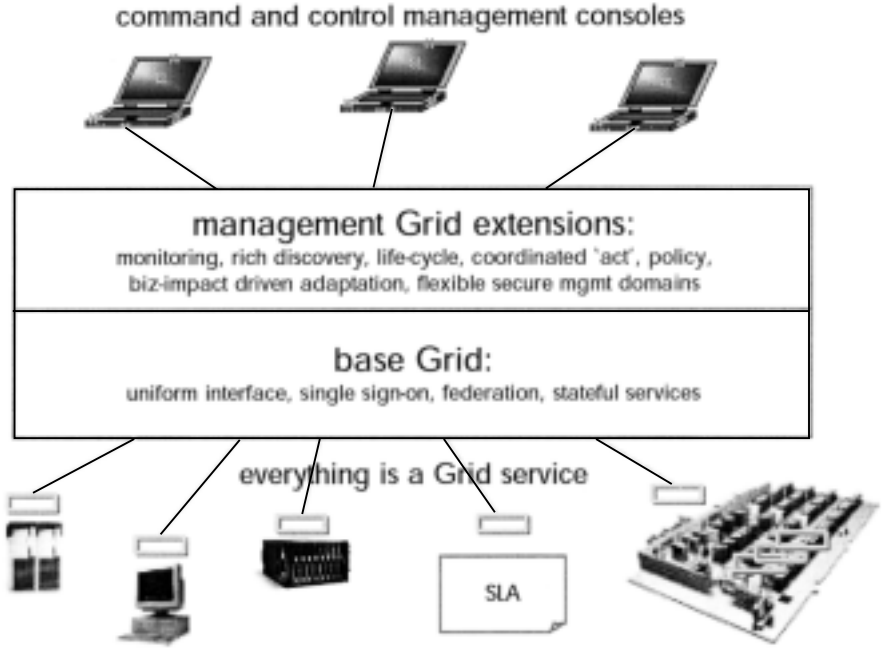
Management software is a distributed software application that assists IT operators and administrators in their respective jobs. Such software has traditionally emphasized instrumentation and monitoring, supported by standardized technologies such as SNMP, CIM, WBEM and product suites such as HP OpenView, CA, IBM Tivoli. Certainly, traditional management software has proven its value, but future Internet-based systems and services will demand richer functionality from management software, resulting in technology challenges that cannot be met by above standards and products.

In particular, IT decisions in the enterprise are no longer based on the necessity to keep up with Moore's Law in CPU and storage—as an example, consider desktop PCs, which for many employees are simply 'good enough' and no longer require continuous replacement. As a consequence, IT decision makers nowadays explore new and different avenues to improve total cost of ownership and return on IT investment. This increases the demand for effective management software, and changes the requirements posed on it. In particular, since operator and support cost is a major element of total cost of ownership, next-generation management software must offer more and more *automation*. Furthermore, IT control (automated or not) must be more and more understood in terms of *business impact*, requiring advanced tools to correlate management actions with their consequences in terms of business metrics.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 4–7, 2003.

To compound these increasingly challenging requirements, the systems for which they are posed are becoming increasingly complex. The *scale* at which enterprise computing operates becomes all encompassing: devices and sensors are getting smaller and smaller, data centers and data sets larger and larger, and everything gets networked together to form an interconnected infrastructure of planetary scale. *Heterogeneity* is a given within each enterprise, and is not yet diminishing, and IT solutions that involve *federation* (that is, hook multiple parties together) become increasingly prevalent. We refer to [1] for a sample scenario outlining the complexity we deal with, and an inventory of components required for its management.

To recap the above challenges in a single sentence, we require *automated and business-driven management, for systems with increasing scale, heterogeneity and federation challenges*. None of the existing management software standards or products is designed to deal with this.



2 Management Software Tomorrow

To respond to the above challenges, we argue that management software must evolve into a richer type of distributed application, with the Grid as its backplane. With Grid, we mean the 'services grid' [2] as advocated through the Global Grid Forum [3], in particular its standards for Open Grid Services Architecture and Infrastructure.

The above figure positions the management backplane relative to existing management software. At the top the legacy management consoles are shown, at the bottom devices are shown that need to be controlled; these devices may have legacy data collection agents (SNMP, CIM). Instead of traditional direct connections from agents

to consoles, we 'wedge' the Grid backplane in between. The backplane maintains a Grid service for anything worth managing, and manipulates the managed elements through this Grid service—in the figure, PCs, storage, SLAs and data centers are controlled in this way. To achieve the desired flexibility, we leverage the base Grid infrastructure services, and to increase functionality we develop Grid service extensions for management purposes (represented in the figure by the two layers of the 'stack')[1].

When evaluating the proposed management backplane, the crucial question is if it helps us fulfill the requirements listed above—that is, how does the backplane support automated and business-driven management, for systems with increasing scale, heterogeneity and federation challenges? We argue that for the following reasons the Grid is the answer to these challenges.

a. **Rich interfaces.** To automate management, one needs control interfaces for devices, and one needs to coordinate control actions across multiple devices. For example, to stop an application when server performance degrades and to restart the application on a different server, one needs start and stop commands and need to coordinate between servers and applications, as well as across the start and stop actions. SNMP and other management protocols are simply too much focused on reporting, and have too few extensibility features to be suitable for such tasks. The Grid, being web services based, is easily and naturally extensible with richer interfaces for adaptation and control.
b. **Uniform, abstraction and virtualization interfaces.** There are several strong software engineering arguments behind introducing the layers of indirection of the management backplane. First, the *uniform representation* of devices by Grid services hides device (and data collection agent) heterogeneity from client software. It also allows distinct management tasks (such as provisioning and runtime SLA assurance) to be unified through the same Grid services. In addition, Grid management extensions can be introduced to provide interfaces at any desired *higher level of abstraction*, such that the interfaces are most natural for the developer of management software. Finally, the Grid management extensions provide a natural place to introduce *virtualization*, that is, a place where the decision is made which resources implement a client request and where the bindings between resources and requests are maintained, without exposing this to the client software (think of virtual LANs or virtual machines).
c. **Leverage web services.** It can not be stressed enough that the Grid is nothing more (or less) than a web services platform (hence our desire to talk about the services grid, to distinguish it from traditional resource and data grids [2]). Since service grid interfaces are web service extensions, one can make use of any other desired web service technology, such as WS Inspection, WS Security, WS Coordination, WS Policy or WS Transaction. Note that these web service standards target federated environments, which can be exploited by management applica-

[1] We previously discussed the integration of management and Grid functionalities in [4], while web (instead of Grid) services based architectures, similar to the management backplane, have been proposed in [5] and [6].

tions in two ways: one can conveniently manage across federated environments, and one can create management applications that are federated themselves. Also note that web services are a natural way to deal with business impact of management actions, since business impact is often determined by the functioning of web services that drive an enterprise. See our discussion of web services management network, which deals with management of business services, providing an architecture that is fully compatible with the Grid-based management backplane [5].

d. **Leverage Grid.** The Grid adds features to web services that make it especially useful for management. In particular, it introduces an elegant way to deal with state, through the service data concept. Since management is nothing else than reporting and manipulating state of devices, it forms an obvious application for the Grid. The Grid also adds features such as single sign-on and the creation of virtual organizations, which may be exploited by management applications to manage dynamically changing environments or dynamically adapt management domains.
e. **Standards.** Scale, heterogeneity and federation make it impossible for single vendors or technologies to provide the complete solution. No proprietary solution can be expected to offer close to all the required features, nor can it be expected to deal with all heterogeneity and federation issues. Standardization is therefore a must, and judged by its working groups (which deal with topics such as resource reservation and management models), the Global Grid Forum is the natural place to standardize service-oriented management technologies.

For all above reasons, the Grid-based management backplane is necessary and appropriate for building future management applications. It helps to automate operator tasks in business-driven fashion, and is able to deal with increasingly challenging issues around scale, heterogeneity and federation of enterprise IT.

References

1. V. Machiraju, J. Rolia, A. van Moorsel, Quality of Business Driven Service Composition and Utility Computing, *HP Labs Technical Report HPL-2002-66*, March 2002.
2. A. Reinefeld, F. Schintke, Concepts and Technologies for a Worldwide Grid Infrastructure. *Euro-Par 2002 Parallel Processing*, Springer LNCS 2400, pp 62–71.
3. *Global Grid Forum*, http://www.ggf.org.
4. S. Graupner, V. Machiraju, A. Sahai, A. van Moorsel, Management += Grid, *HP Labs Technical Report HPL-2003-114*, March 2003, accepted for publication in 2003 workshop on Distributed Systems: Operations and Management.
5. V. Machiraju, A. Sahai, A van Moorsel, Web Service Management Network: An Overlay Network for Federated Service Management, *Proceedings of IM 2003*, Colorado, USA, March 24–28, 2003.
6. *Web Services Management Framework technical documents*, available from http://devresource.hp.com, 2003.

Plug and Play Fault Injector for Dependability Benchmarking

Pedro Costa[1], Marco Vieira[2], Henrique Madeira[3], and João Gabriel Silva[3]

[1] ISCAC/CISUC, Polytechnic Institute of Coimbra, 3040-316 Coimbra - Portugal
pncosta@dei.uc.pt
[2] ISEC/CISUC, Polytechnic Institute of Coimbra, 3031 Coimbra – Portugal
mvieira@isec.pt
[3] DEI/CISUC, University of Coimbra, 3030 Coimbra – Portugal
{henrique,jgabriel}@dei.uc.pt[†]

Abstract. Dependability benchmarks must include fault injectors that are very simple to install and use. They should be downloadable from the web together with the other components of the benchmark and be able to target all system components. Since the existing injectors did not fulfill this requirement, we have developed DBench-FI. Presently this SWIFI tool targets the Linux OS on Intel processors, and uses a flexible runtime kernel upgrading algorithm to allow access to either user or system spaces. It does not require the availability of the source code of any system component or user process. It is not based on any debug mode either, being able to inject faults even in tasks that were already running when it is installed, irrespective of their complexity. It currently only injects memory faults, but there are plans to include other fault models in the future. Results of complex fault injection campaigns are presented.

1 Introduction

With the spread of computing systems and the growing number of its applications in critical tasks, high dependability is required. Systems failures are increasingly more inconvenient and cause more damages then ever before. Although the more serious consequences arise from failures in safety critical applications, such as medical and aircraft systems, there are other areas where such system failures cause important damage, like financial losses. There are many recent examples of system failures with consequent high costs in a wide range of areas. The evaluation of the dependability of computer based systems is hard [3]. One of the approaches relies on benchmarks that require faults or errors to be injected, and then analyze the ensuing system behavior. Developing and validating dependability benchmarks is the objective of the European Union sponsored project

[†] Funding for this paper was provided, in part, by Portuguese Government/European Union through R&D Unit 326/94 (CISUC) and by DBench Project, IST 2000 - 25425 DBENCH, funded by the European Union

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 8–22, 2003.

DBench (`www.laas.fr/dbench`), within which the fault injector described in this paper was developed.

The DBench-FI injector had a very clear set of requirements - benchmarks including the injector had to be downloadable from a web site, execute without any special installation procedure, while at the same time being able to inject faults/errors in the whole system, not just in the user space as some simple tools based on the ptrace mechanism do, nor requiring a special launch procedure like the one required by many debugging mechanisms.

Software implemented fault injection, also known as SWIFI, was a clear choice, since using special hardware was excluded. Still, most existing SWIFI tools like for instance Xception [2] require complex installation procedures, including changes to the kernel that have to be done offline. With DBench-FI every thing is done on-the-fly. On the other hand, the general SWIFI drawbacks apply,like:

- Not all locations are accessible when compared to hardware fault injectors (some processor and system resources cannot be reached) [2];
- Permanent faults are very difficult to inject, except for very particular circumstances;
- The execution and consequently the performance of the system under test is disturbed.

The last problem, known as intrusiveness, is a consequence of the instrumentation necessary to inject faults and monitor their effects in the target system. Special care must be taken to minimize its effects. With DBench-FI, the intrusiveness is essentially undetectable in almost all applications. Despite the negative aspects, SWIFI is very popular because tools are easy to develop, and the injected faults or errors are representative of many real upsets, and the injectors are very portable.

The current version of the DBench-FI injector uses a very simple error model - it just changes the value of memory locations. This simple error model was deemed sufficient for the first versions of the benchmarks, particularly if the target areas for injection are carefully chosen, as was done in the experiments reported in this paper. Still, it is clear that as more sophisticated versions of the benchmarks are developed, the set of supported fault models will have to be enlarged. The techniques used in other SWIFI tools like Xception [2] and Mafalda [11] can be added, as no architectural characteristic of DBench-FI prevents them from being used. A deeper justification of the relevance of this simple fault-model for benchmarking purposes is outside the scope of this paper, as it has to do with the requirements for a dependability benchmark, not with an intrinsic limitation of the methods used to build the fault injector.

This paper is organized as follows. Related research is discussed in section 2. Section 3 presents the architecture of the fault injector as well as a detailed description of the proposed methodology. This section also contains a short survey of the Linux kernel architecture. Section 4 describes the test-bed used to demonstrate the fault injector capabilities and presents the results obtained in experiments. Section 5 concludes the paper and presents some future work.

2 Related Work

Many injection tools have been developed. Here only those belonging to the SWIFI family are considered, as for the type of dependability benchmarking that is considered in this paper no special hardware can be used.

One of the early approaches is FIAT (Fault Injection-Based Automated Testing Environment) [12]. It adds fault injection and monitoring capabilities to application code and operating system, by changing the code and data that is copied in memory at load time. Faults are triggered when target system execution reaches the locations where special instructions have been inserted in the code. Although this tool could inject memory faults at runtime, it was not able to inject most transient faults. A similar pre-runtime approach of changing the file image generated by the compiler was taken by a tool called DOCTOR, developed for the HARTS real-time distributed system [5].

The FINE tool [8] used a software monitor to trace the control flow and inject faults. It had a large overhead, and required the source code of the target application, but was significantly more powerful than its predecessors, particularly in the type of faults that could be injected. DEFINE [7], an extension of FINE, was developed to include distributed capabilities, introducing a modified hardware clock interrupt handler to inject CPU and bus faults with time triggers and inject some kind of software faults.

Kanawati *et al.* [6] developed the FERRARI tool for injecting faults using the UNIX ptrace function. The fault process initiates and executes the target process in a special trace mode, enabling the injection of transient and permanent faults. It was able to inject a very wide set of fault types, but was restricted to injection in user space.

FTAPE (Fault Tolerant and Performance Evaluator) [14] is part of a fault tolerant benchmark which measures system failures and the system performance degradation during fault conditions. It also includes a synthetic program for generating CPU, memory and I/O activity. The FTAPE SWIFI tool is able to inject fault in CPU registers, memory and the disk subsystem. It is capable of selecting the time and location of faults either based on the workload activity or randomly.

The Xception tool, which uses the debugging and monitoring capabilities of modern processors, is presented in [2]. It provides a set of spatial, temporal and data manipulation fault triggers like FERRARI or FTAPE, but with a minimal intrusion on the target system, besides being able to target also system space. Xception was originally implemented on a PowerPC based machine, and has since been ported to other processors, having originated the single commercial fault injector offering we know of (www.xception.org).

MAFALDA, presented in [11], uses principles very similar to Xception, adding mechanisms to intersect and inject system calls in microkernels. The ChorusOS microkernel was the first target.

During the development of DBench-FI we became aware of an improved ptrace-based SWIFI tool, presented in [15]. HiPerFI (High-Performance Fault Injector), very significantly reduces the intrusiveness and overhead caused by the

context switch between the injector process and the target application. It also integrates a method, similar to the approach used by Xception, which enables the fault injection mechanism to intersect the kernel exception handlers and thus extends significantly the tools triggering and injection capabilities. Like DBench-FI, it does not require kernel recompilation and rebooting in order to statically link those new functionalities into the kernel image, but still requires the target applications to be launched in debug mode.

None of these systems satisfied the requirements of web distributable dependability benchmarking, either because the overhead caused would be too high, or because only user space could be targeted, or because source code of the target applications was required, or because a special debug mode imposing a particularly launch mode was required. None of these limitations could be accommodated, thus leading to the development of DBench-FI. Its central feature is perhaps the runtime kernel schedule upgrading algorithm, independent of any debug mode.

3 Fault Injector Architecture

The DBench-FI consists of two modules, shown in Fig. 1: a fault injector core module and a fault injector controller module. The core module, dynamically linked with the kernel, is responsible for implementing the runtime kernel upgrading algorithm. The new kernel, incorporating this module, provides to the user the capability of injecting faults into any process running on the target system, including those that are part of the operating system itself. The user interface is given by the fault injector controller module. Information about the target process pid, fault type (stuck-at-0, stuck-at-1 and bit-flip) and target address range are sent to the core module by the fault injector controller module.

The fault injector has been implemented on an Intel Pentium IV system running the Linux RedHat 7.3 (Linux kernel version 2.4.18-3). The algorithm is implemented using Linux loadable kernel modules.

3.1 Fault Injection Design and Implementation

The fault injector presented is based on common characteristics and concepts of operating systems like memory management, where any process running on the system is regarded as having its own memory address space, and scheduling, where multiple processes appear to be simultaneous executed even in systems with just one processor.

In Linux, like in all monolithic architectures, the operating system functionality is concentrated within the kernel. The function responsible for deciding the next executable task that will be dispatched to the CPU is the scheduler. The scheduler is called when a task yields the processor, blocks in I/O, uses up its quantum or is preempted by another task (with higher priority). Fig. 2 gives a common view of the Linux kernel architecture, with special attention on the

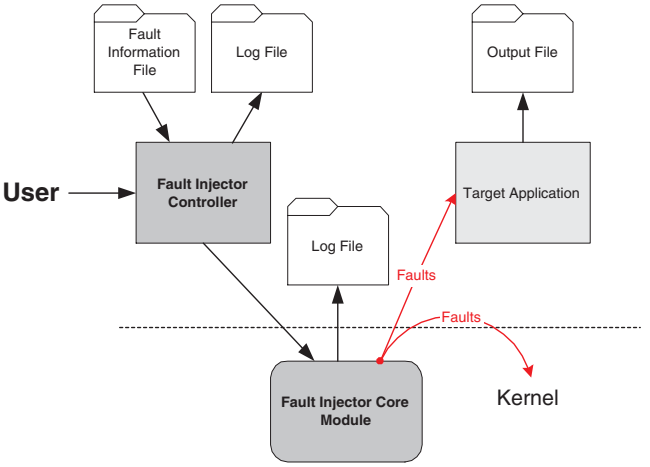


Fig. 1. The fault injector architecture

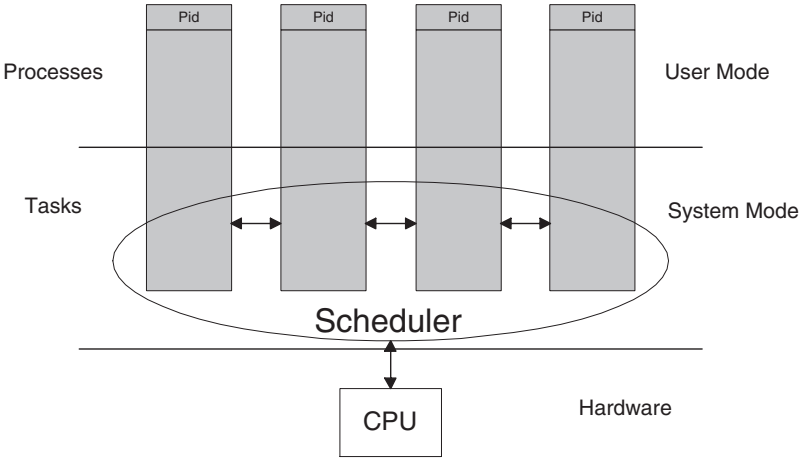


Fig. 2. The Linux operating system memory architecture

interaction between applications, scheduler and hardware. Please note that the kernel is mapped in the address space of every process.

Any process running on the target system is regarded as having a virtual memory space, which includes its code and data segments. A Linux process can allocate three types of memory: stack, heap and mmaped memory, as shown in Fig. 3. A detailed description of the components and mechanisms of the Linux kernel are described in [1], [4] and [9].

The current version of DBench-FI injects faults in the memory space of a particular process. Since the virtual address space of a process is only accessible when that process is scheduled, DBench-FI dynamically changes the Linux kernel

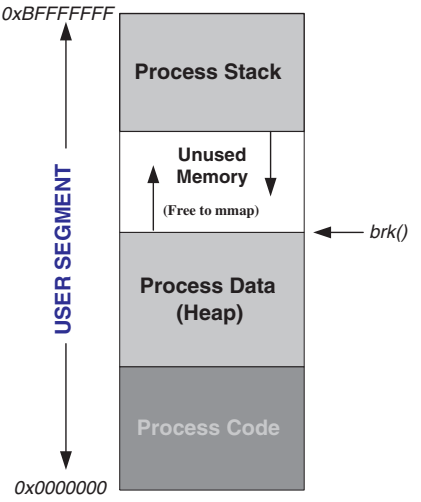


Fig. 3. The user segment (code, heap, stack and mmaped memory)

to intersect the scheduler, in order to detect when a given process gains control over the processor. The required fault is then injected. The kernel schedule has thus to be redirected to a new function called new_schedule, responsible for both the process detection and the fault injection. The procedure is illustrated in Fig. 4 and consists of the following steps:

(1) Determine the runtime address of schedule kernel function using a memory pattern search algorithm. This step requires that DBench-FI knows that pattern, which is always the same, except if in some new version of the Linux kernel changes that part of the code, which is highly unlikely, but will be detected if that's the case;
(2) Copy the first nine bytes of the kernel schedule function to a new function called saved_instructions;
(3) Generate a jump instruction with the runtime address of new_schedule function;
(4) Overwrite the first bytes of schedule code with the generated jump instruction;
(5) Create jump instructions in order to execute the original scheduler function code (to saved_instructions and to the 10th byte of kernel schedule).

It should be noticed that when the fault injector kernel module is loaded, the policy and main algorithm of the operating system scheduler remains the same. Additionally, when it is unloaded or removed, the redirections that were made are undone and the scheduler becomes exactly the original. Concerning the intrusiveness, it is important to stress that when the fault injector is loaded and no faults are injected, the performance penalty corresponds to executing

only ten additional machine assembly instructions that were added in order to intercept the scheduler. This fact guarantees very low intrusiveness.

Information identifying the target process and the fault type are given by the experiment controller module, which allows the user interaction with the new upgraded kernel. In order to activate the injection of the fault, temporal fault triggers are used. In this way, the injected faults are not related with any specific activity of the target application. Related to temporal fault triggers, two options are available in this implementation: a fault is injected once after a given time is elapsed since the application starts, or a fault is repeatedly injected with a certain frequency. Temporal triggers that are randomly chosen are particularly adequate to benchmarking, as they enable statistically significant results to be obtained. The fault types implemented in this first version are: stuck at 0, stuck at 1 and bit flip.

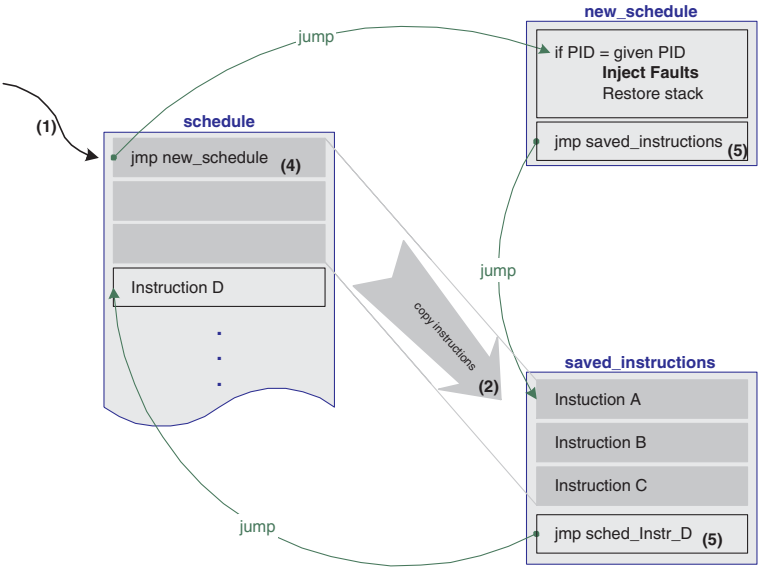


Fig. 4. The fault injection method

Since the presented method is based on the kernel scheduler interception and redirection, it is appropriate to inject faults into any system memory address, including the kernel memory segment. This capability made this fault injector suitable to analyze the kernel robustness under faults, and represents a huge advantage comparatively to the ptrace-based SWIFI tools. Another important benefit relatively to the ptrace mechanism is that DBench-FI can inject faults into any running target application without having to load it explicitly. This is an essential requirement to analyze the dependability of complex systems

like DBMS's. An important issue with SWIFI tools is their portability to other processors. The proposed method can be, with some minor changes, adapted to almost any operating system and processor. A further advantage is its simplicity and ease of use, since there is no need to recompile the kernel or the target application.

3.2 Limitations

The main limitation of DBench-FI, besides the general SWIFI limitations described in the introduction, is the limited set of fault models supported. This is not a limitation of the technique itself, but just of the current implementation, as ease of use was considered more important for dependability benchmarking. For the next versions, it is foreseen that most of the fault models of Xception will also be included, like spatial triggers and injection in the processor resources. A limitation of the technique is the fact that supervisor level privileges are required to install it, as operating system security rules understandably prevent user level processes from modifying the kernel.

4 Experimental Results

In order to demonstrate the efficiency and the capabilities of the presented fault injector, several experiments were conducted using an Oracle 9i Database Management System (DBMS) [10]. For each experiment, the same profile defined by a set of combinations of the target process pid, the target address range, the fault type and their duration, was used. The following sections describe the experiment details.

4.1 Fault Injection Experiments

To exercise the Oracle 9i transactional engine we used the Transaction Processing Performance Council (TPC) Benchmark^{TM} C (TPC-C) [13], which represents a typical database installation. The business represented by TPC-C is a wholesale supplier having a number of warehouses and their associated sales districts, and where the users submit transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. This workload includes a mixture of read-only and update intensive transactions that simulate the activities found in many complex On-Line Transaction Processing (OLTP) application environments. The performance metric provided by the TPC-C benchmark is expressed in transactions-per-minute-C (tpmC).

Fig. 5 shows the key components of the experimental setup. The TPC-C specification includes an external driver system and a System Under Test (SUT). The SUT consists of one or more processing units used to run the workload (set of transactions submitted), and whose performance is to be evaluated. The driver system emulates all the client applications and respective users during the

benchmark run. In the present experimental setup, the memory injector is part of the SUT and an experiment controller module has been added to the driver system in order to control the experiments and to collect the base data needed to get the measures.

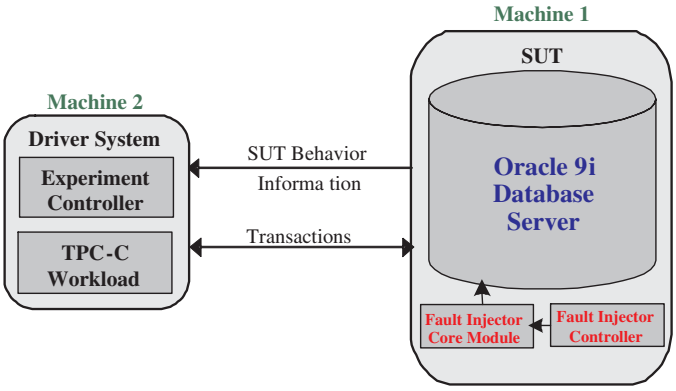


Fig. 5. Experimental setup layout

The basic platform used in the experiments consists of two machines connected through a dedicated fast-Ethernet network. Table 1 summarizes the main characteristics of each machine.

Table 1. Hardware platform used in experiments

Machine	OS	Hardware Platform
SUT (Machine 1)	RedHat Linux 7.3 Kernel 2.4.18 -3	*Processor* : Intel Pentium IV 2 GHz *Memory* : 512 MB *Hard Disks* : One 60GB / 7200 rpm
Driver System (Machine 2)	Windows 2000 Professional SP3	*Processor* : Intel Pentium III 800 MHz *Memory* : 256 MB *Hard Disks* : One 20GB / 7200 rpm

In each experiment, several faults are injected following the profile presented in Fig. 6. The test in each experiment is conducted in a steady state condition that represents the state in which the system is able to maintain its maximum transaction processing throughput (the system achieves a steady state condition after a given time (steady state time) executing transactions). The duration of each individual experiment is 60 minutes plus the time needed for the system to achieve the steady state condition (60 minutes + steady state time). Exceptionally, the experiment is finished before the 60 minutes, if the operating system or

the DBMS hangs due to the effects of the faults injected. It is important to note that the execution of all the experiments is fully automatic.

In order to make it easy to reproduce and compare the results of the experiments we have decided to inject the faults in specific moments (instead of using a random distribution for the fault triggers). This way, the first fault is injected 1 minute after the steady state condition has been achieved and the following faults are injected in intervals of ten minutes after the previous one. The used duration revealed itself efficient in previous test experiments.

It is important to note that, the SUT state is explicitly restored in the beginning of each experiment and the effects of the faults do not accumulate across different experiments. Concerning the fault model, each fault injection consisted of stuck-at-0 affecting 8 KB of the Oracle 9i System Global Area (SGA) [10] (this is the area of memory used by Oracle to store the database information). The target address bytes are contiguous and randomly selected within the SGA.

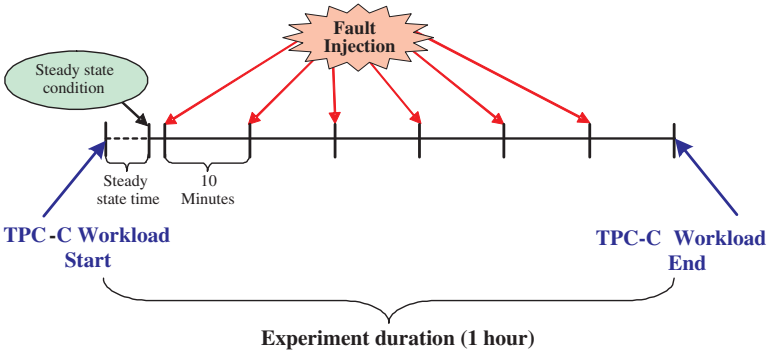


Fig. 6. Fault injection profile

As mentioned before, the experiment controller module collects the base data needed to get the measures. The measures consist of the following:

- Failure mode - represents the type of failure from the SUT point-of-view. Several failure modes have been considered, namely: OS Hang (corresponds to the cases where the operating system is frozen), DBMS Crash (represents the cases where is observed an abrupt shutdown of the Oracle database server), Transactions Aborted (corresponds to the situations where there is no OS or DBMS hang but the Oracle database server aborts some of the submitted transactions due to internal error conditions), and Correct Results (represents the cases where the faults injected did not cause any kind of incorrect behavior in the SUT).
- Transactions-per-minute-C (tpmC) - the number of transactions executed per minute is the standard measure provided by the TPC-C benchmark and

represents the total number of completed New-Order transactions (one of the 5 types of TPC-C transactions) divided by the duration of the measurement interval.
- Percentage of Aborted transactions - percentage of TPC-C transactions submitted (considering all the types of TPC-C transactions) that were aborted by the database server due to an internal error condition.

4.2 Results and Discussion

During the experimental evaluation, 100 experiments have been executed. Fig. 7 shows the failure modes observed from the SUT point-of-view. As we can see, results show that, in most of the experiments, the faults injected in the Oracle SGA caused the server abrupt shutdown (DBMS Crash) and only in a few cases the operating system hanged. There is also a considerable set of cases where the DBMS, although still running, aborted some of the transactions submitted due to errors caused by the faults injected. Finally, only in 2% of the experiments the faults injected have not caused the SUT to produce any erroneous behavior. The results observed show the DBMS behavior under the presence of transient memory faults and the effectiveness of the proposed fault injection methodology.

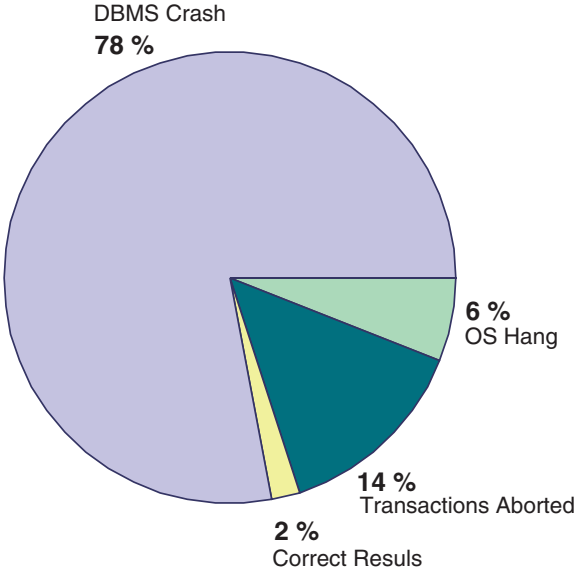


Fig. 7. SUT failure modes observed

The DBMS Crash failure mode represents the cases where an abrupt shutdown of the Oracle database server is observed. This failure mode represents the

correct and expected behavior of a DBMS in the presence of memory faults (it is better for a DBMS to indicate to the system administrator the occurrence of system errors through a shutdown than to continue executing erroneous transactions without any warning). An important aspect to analyze is the number of injected faults before the DBMS crash. This is shown is Fig. 8. As we can see, the majority number of the DBMS crashes occurs between the first and the third injected faults. Knowing that the baseline performance (without any fault

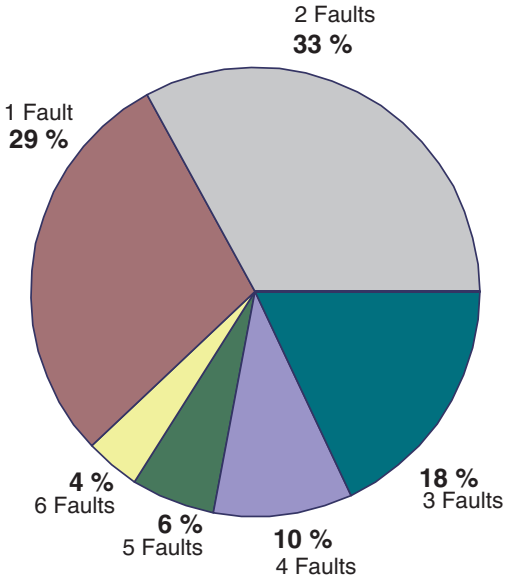


Fig. 8. Number of injected faults before DBMS crash

injection) is 2270 tpmC, we can notice a performance penalty in the experiments that generated this failure mode (see Fig. 9).

The OS Hang failure mode corresponds to the situation where the SUT operating system freezes. As we can state in Fig. 7, this case occurs in only 6% of the experiments. This behavior is due to Oracle DBMS complexity and the used randomness of the fault injection target addresses.

The Transactions Aborted failure mode corresponds to the situation where neither the OS nor the DBMS crashes, but some of the workload transactions are aborted. Fig. 7 shows that this situation occurs in 14% of the total experiments. Concerning this failure mode, it is important to analyze the percentage of aborted transactions (see Fig. 10). It must be noticed that the minimum percentage verified corresponds to a number in the order of the hundreds of transactions. Another relevant characteristic is related with the tpmC verified in each experiment that reports this failure mode. Comparing the values obtained in this failure mode (Fig. 11) with the values related to DBMS crash failure

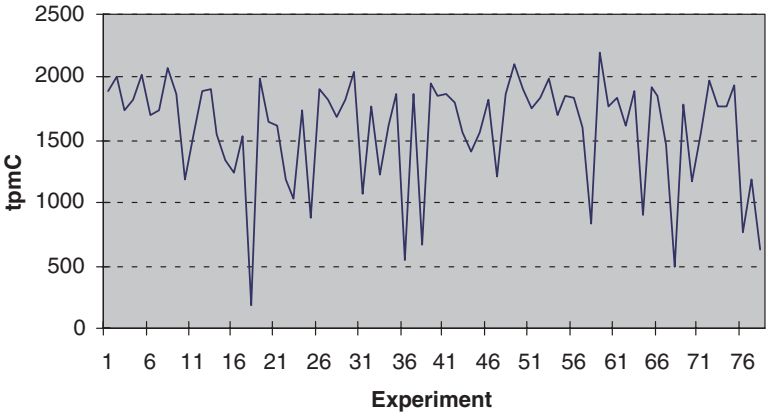


Fig. 9. The tpmC observed in DBMS Crash failure mode

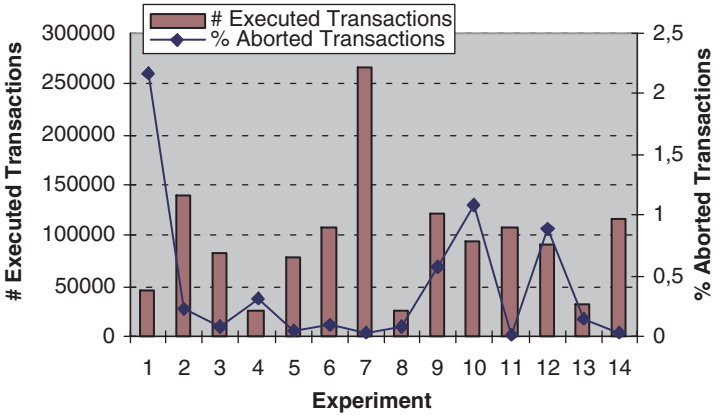


Fig. 10. Percentage of aborted transactions

mode (Fig. 9) we can state a huge decrease of the mean values. This performance degradation is due to the activation of the recovery mechanisms of the Oracle DBMS.

5 Conclusions and Future Work

This paper presented a new software fault injector, DBench-FI, meant to be included in web distributable dependability benchmarks. It has a unique set of features, required by that type of application: very low intrusiveness, capable of injecting both in user and system space, does not require application source code to be available, can be dynamically loaded into a system, and can inject even on applications that are already running when it is installed. The central

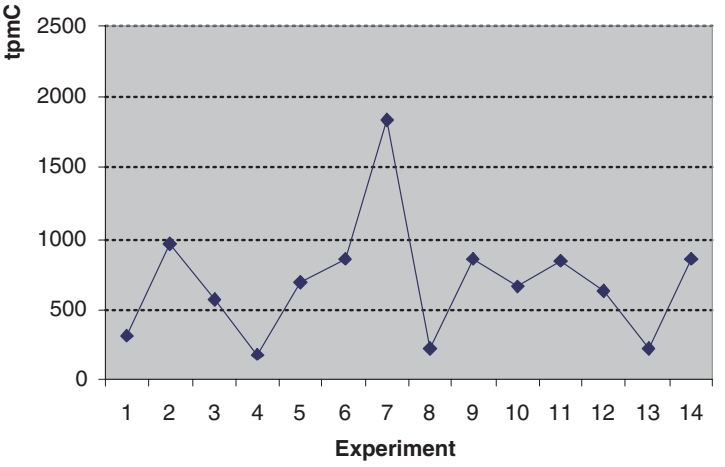


Fig. 11. The tpmC observed in Transactions Aborted failure mode

feature is perhaps the runtime kernel schedule upgrading algorithm, together with a carefully crafted integration with the scheduler and memory management functions. The main current limitation of DBench-FI is the fact that it only injects time triggered memory faults.

While the set of experiments described in this paper fully shows that the injector is capable of handling complex tasks, a more detailed study is under way to determine the exact needs of dependability benchmarks. The fault models supported by the Dbench-FI fault injector will then be extended as required, but no changes are expected to its basic structure.

References

1. Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., Schröter, C., Verworner, I.: Linux Kernel Programming. 3rd edn. Addison Wesley (2002)
2. Carreira, J., Madeira, H., Silva, J.G.: Xception: A Technique for the Experimental Evaluation Dependability in Modern Computers. IEEE Trans. Software Eng., Vol. 24, No. 2 (Feb. 1998) 125–136
3. Carreira, J., Madeira, H., Silva, J.G.: Xception: Software Fault Injection and Monitoring in Processor Functional Units. 5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5) (Sep. 1995) 245–265
4. Ezolt, P.: A Study in Malloc: A Case of Excessive Minor Faults. Proc. 5th Annual Linux Showcase & Conference (Nov. 2001)
5. Han, S., Rosenberg, H., Shin, K.: DOCTOR: An Integrated Software Fault Injection Environment. Tech. Report, Univ. of Michigan (1993)
6. kanawati, G., Kanawati, N., Abraham, J.: FERRARI: A Flexible Software-Based Fault and Error Injection System. IEEE Trans. Computers., Vol. 44, No. 2 (Feb. 1995)

7. Kao, W., Iyer, R.: DEFINE: A Distributed Fault Injection and Monitoring Environment. Proc. Workshop Fault-Tolerant Parallel and Distributed Systems (June 1994)
8. Kao, W., Iyer, R., Tang, D.: FINE: A Fault Injection and Monitoring Environment for Tracing UNIXSystem Behavior Under Faults. IEEE Trans. Software Eng., Vol. 19, No. 11 (Nov. 1993) 125–136
9. Maxwell, S.: Linux Core Kernel Commentary. 2nd edn. Coriolis (2002)
10. Oracle Corp.: Oracle 9i Server Concepts Manual (2001)
11. Rodríguez, M., Salles, F., Fabre, J.-C., Arlat, J.: MAFALDA: Microkernel Assessment by fault injection and design aid. 3rd European Dependable Computing Conference (EDCC-3) (Sep. 1999) 143–160
12. Segall, Z., Vrsalovic, D., Siewiorek, D., Yaskin, D., Kownacki, J., Barton, J., Dancey, R., Robinson, A., Lin, T.:FIAT – Fault Injection Based Automated Testing Environment. Proc. 18th Int. Symp. on Fault-Tolerant Computing (FCTS-18) (June 1988) 102–107
13. Transaction Processing Performance Consortium: TPC Benchmark C, Standard Specification, Version 5.0. `http://www.tpc.org/tpcc/ (2001)`
14. Tsai, T., Iyer, R.: An approach towards Benchmarking of Fault-Tolerant Commercial Systems. Proc. 26th Int. Symp. on Fault-Tolerant Computing (FCTS-26) (June 1996) 314–323
15. Xu, J., Kalbarczyk, Z., Iyer, R.: HiPerFI: A High-Performance Fault Injector. Fast Abstract in Proc. IEEE Int. Conference on Dependable Systems and Networks (June 2002)

Non-intrusive Software-Implemented Fault Injection in Embedded Systems [1]

Pedro Yuste, Juan Carlos Ruiz, Lenin Lemus, and Pedro Gil

Fault Tolerant Systems Group (GSTF)
Technical University of Valencia
46022 Valencia, Spain
{pyuste, jcruizg, lemus, pgil}@disca.upv.es

Abstract. Critical embedded systems, like those used in avionics or automotive, have strong dependability requirements and most of them must face with fault tolerance. One of the methods typically used to validate fault tolerance mechanisms is fault injection. The idea is to study the behavior of the system in presence of faults in order to determine whether the system behaves properly or not. Software-implemented fault injection (SWIFI) techniques enable fault injection to be performed by software. Although interesting, major drawbacks of existing SWIFI techniques are the temporal and the spatial overheads they induced in the systems under study. The reduction of these overheads is thus crucial, in order to be confident on the results and conclusions of a SWIFI experiment. This paper focuses on this problem. It proposes a new non-intrusive SWIFI technique for injecting faults in embedded (system-on-chip) applications. The technique exploits the features of a standard debugging interface for embedded systems, called Nexus, in order to inject faults without temporal overhead. Then, Nexus features are also exploited in order to observe, without spatial intrusion, the behavior of the target system in presence of the injected faults. In other words, the embedded system under study can be controlled (for injecting faults) and observed (for tracing its behavior) without customizing its original structure or altering its normal execution. Since based on Nexus, the technique has also the benefit of being applicable to any Nexus-compliant system. In order to illustrate the potentials of the approach, we use an automotive embedded control unit application as a case study. Some preliminary results obtained from the experiments performed are also discussed.

1 Introduction

Nowadays, embedded systems are widely used in a large spectrum of application domains, varying from automotive systems to avionics. In these domains, a failure of

[1] This work has been supported by the European IST project DBench (Dependability Benchmarking, IST-2000-25425).

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 23–38, 2003.

the system could have disastrous consequences from both economical and human viewpoints. For this reason, these applications need a high degree of dependability.

Fault injection (FI) is nowadays a well-known technique for characterizing the behavior of (critical) systems in presence of faults [1]. Basically, the goal is to study whether or not computing systems are able to tolerate (or to handle in a safety manner) the occurrence of a fault while providing their service. As stated in [2], these faults can be (i) injected using specialized hardware tools (hardware-implemented fault injection or HWIFI); (ii) emulated from programs (software-implemented fault Injection or SWIFI) or (iii) simulated using models (simulation-based fault injection). In this paper, our interest focuses on SWIFI techniques.

From a high level viewpoint, the term SWIFI groups a set of techniques used to inject faults on computer systems using software. One of the goals of these techniques is to emulate, at a logical level, the consequences of hardware faults in a system [3]. As stated in [4], SWIFI techniques also enables the emulation of software (design) faults. However, this second issue goes beyond the scope of this paper.

Most of the existing SWIFI techniques suffer from two major problems. The first regards their portability, as they usually require deep modifications in the system in which they are applied [5]. The second is the introduction of a temporal overhead that slows down the execution of the application [6]. This overhead may prevent certain of these SWIFI techniques from being applicable on systems with real-time constraints. Observability is an additional problem to consider when applying SWIFI techniques on embedded (System-on-Chip or SoC) systems. In this case, complexity and scale of integration makes difficult the task of observing and tracing the behavior of the system in presence of faults.

This paper focuses on the definition of a new SWIFI technique that handles the above problems. Exploiting the standard features provided by the embedded processor debugging interface Nexus [7], our technique is able to inject faults in SoC applications on-the-fly, while avoiding any temporal intrusion in their normal execution. Then, the approach also provides solutions for observe and study the behavior of the target application in presence of the injected faults. This observation is performed without any spatial intrusion, i.e., without customizing the considered application.

Section 2 provides a high-level description of the main features of Nexus that are useful for injecting faults in embedded (SoC) applications. Taking into account these features, Section 3 presents the SWIFI technique that we have defined. Section 4 illustrates this technique using an embedded diesel engine control unit (ECU) as a case study. Section 5 comments the relation of this work with others of the domain. Section 6 provides the conclusions and depicts the future of this research.

2 Nexus-Based Fault Injection

Nexus is a standard for embedded system debugging [7]. It has been defined by the Nexus 5001 Forum ™, a consortium including important processor manufacturers (such as Hitachi, Infineon, Motorola, National Semiconductor and ST). The Nexus

standard defines a set of features that are classified in four classes. Each nexus class is a superset of its previous class. This implies that Nexus class 2 includes the features of Nexus class 1 and it provides some new ones. Although this paper focus on Nexus class 3 features, the detailed description of all the features associated to this Nexus class is out of scope. Thus, we will only focus on those features of Nexus class 3 that are basic for the understanding of our fault injection technique.

2.1 Portability

One of the aims of the Nexus consortium is the specification of a universal debugging interface. This interface enables the development of debugging tools that runs on any system providing a Nexus-compliant port. As a result, any tool or technique that follows the Nexus specification will benefit from this portability.

Our SWIFI technique enables the injection of faults and the observation of the target system activity using the sole management capabilities offered by Nexus. As a result, this technique does not depend on any particular feature supplied by the considered target system. On the other hand, its specification only relies on features defined in the core of the Nexus standard. Thus, existing proposals for extending the standard are not taken into account.

The above arguments justifies to what extent the SWIFI technique described in this paper is portable (i.e. able to characterize the behavior of any Nexus-compliant embedded system in presence of faults).

2.2 Temporal Overhead

As stated before, most SWIFI techniques introduce an overhead that may alter the temporal behavior of the system under study. This overhead typically results from (i) the selection of the moment in which faults must be injected (fault injection trigger); and (ii) the corruption of the selected system memory locations (fault injection). This computation results in a certain overhead that may be non-negligible from the viewpoint of the system application, especially when this application needs to meet with some real-time execution constraints.

In order to solve the above issues, we propose the use of Nexus as follows:

- Watchpoints solve the problem of triggering the fault injection. These watchpoints are part of the Nexus debug circuitry and they are used in order to signal application events without halting the execution of the application. Basically, Nexus enables the definition of two types of application events: (i) the execution of a given code instruction, and (ii) the access to a predetermined data memory word. In our approach, fault injection is triggered according to these two types of events. Further details on that issue will be supplied in Section 3.2.
- On-the-fly run-time memory access capabilities solve the problem of injecting the faults in memory without interfering with the system normal activity. What we propose is to exploit the mechanisms of the Nexus circuitry in order to

intercept code and data memory accesses. This enables injecting faults in memory before its contents being used by the application.

According to the Nexus standard, the use of these features must not induce inducing any temporal overhead in the considered target system. Thus, every embedded system providing a Nexus-compliant debugging port should ensure this property.

2.3 Observability

As Nexus has been initially specified for debugging purposes, its specification proposes a number of observation capabilities that are very useful for observing the effects produced by fault injection in a target system. The idea is to build execution traces of the system that are rich enough in order to deduce both event- and time-oriented measures from their analysis.

Our SWIFI technique exploits the observation capabilities of Nexus in order to build traces of the execution of the target application:

- Branch trace messaging is useful in order to produce (at run-time) a graph of the application control flow. In each branch trace message, Nexus provides information about the timing and the set of instructions execution since the last branch trace message. This information is the one stored in the resulting application control flow graph.
- Data traced messaging provides the possibility of monitoring read and write memory accesses to the application data.

The combined use of the above tracing features with watchpoints enables to know when a (dormant) fault in memory becomes an error. The idea consists in using watchpoints in order to detect when the execution flow of the target application access a (code or data) memory word containing a fault. This event signals the activation of the fault (i.e. an error appears in the system). Other events that can be traced using this approach are the detection of an error through the activation of an exception and the end of the error recovery process.

The resulting execution traces represent the knowledge that can be obtained through Nexus from the internal activity of an embedded SoC application. Basically, these traces are the image that can be later used in order to study the activity of the system in presence of faults. It must be however worth noting that these execution traces also maintain timing information for the occurrence of each monitored event or message. This timing information is essential for latency computation.

Taking the all above features in mind, next section explains how to use them in order to inject faults in an embedded SoC application.

3 Fault Injection Technique

As their name evoke, SoC applications are embedded applications running inside a chip. This is a major impairment for the control and observation of their behavior in

absence and in prsence of faults. In this second case, injecting faults in a SoC application constitutes a big deal due to the scale of integration and the complexity of current chips. To the best of our knowledge, this is a problem that has not been tackled before.

The goal of this section is to explain how fault injection can be performed on SoC applications. We structure this presentation in three successive paragraphs that provide answers to the following questions:

i. Which are the set of faults that the technique is able to inject?
ii. How is determined the moment in which each fault must be injected?
iii. Which are the successive steps to be followed for injecting the fault?

3.1 Fault Model

As stated in the introduction, we focus on hardware fault models. The faults represented by these models can be permanents or transients. The stuck-at model refers to the first type of faults and the bit-flip model to the second. As stated in [8], it is worth noting that the bit-flip model is representative of a high percentage of the HW faults that an embedded system may suffer. For the sake of completion, stuck-at faults also need to be taken into account.

In our context, injecting a bit-flip, requires a procedure that has three steps. The first thing to do is to read the content of the memory position (or register) where the fault will be injected. Then, a XOR function is applied to this content using a mask that provides the bit (or the set of bits) that must be flipped. If the applied mask is, for example, 0x1C (^{bit7}00011.1000^{bit0}) then bits 2, 3 and 4 of a memory byte will be flipped during the fault injection process. Finally, the modified content is written back to its initial location (either a memory byte or a memory-mapped register). Following this approach, the technique is able to inject both single and multiple bit-flips in a selected byte boundary location.

Injecting a stack-at fault requires the continuous monitoring of the location where the fault must be introduced. When the contents of this location is used, then a stuck-at 1 fault can be injected by applying an OR function on the contents of this location. The logical value 1 is in this case used in the mask for representing those bits that must be stuck at 1. Symmetrically, the stuck-at 0 fault model can be implemented by applying an AND function on the same contents. In this case, the mask signals the set of bits to be stacked at 0 with the logical value 0.

3.2 Injection Trigger

There are two possible fault injection triggering schemes:

- Temporal: The fault injection is activated depending on temporal parameters related to the execution of the system workload, for example a random time since application start.

- Spatial: The fault injection is activated when the workload reaches a state or condition. It can simply be the execution of a specified instruction or it can be the activation of some event in the system, for example, an exception.

To implement a temporal trigger the application has to be launched and the injection tool has to wait during a period (defined as the injection time) before modifying the contents of the selected register or memory word. To implement a spatial trigger a watchpoint or breakpoint has to be set at the memory address or memory-mapped register whose access activates the fault injection procedure.

A more general trigger can be also defined as the combination of both a spatial and a temporal trigger. In this case, the moment of the fault injection determined by a watchpoint or breakpoint event (spatial aspect) plus a certain delay (temporal aspect). A purely temporal trigger can be thus viewed as a general trigger with a delay after a reset of the system. A purely spatial trigger can be also defined in terms of a general trigger with a zero delay after a system event.

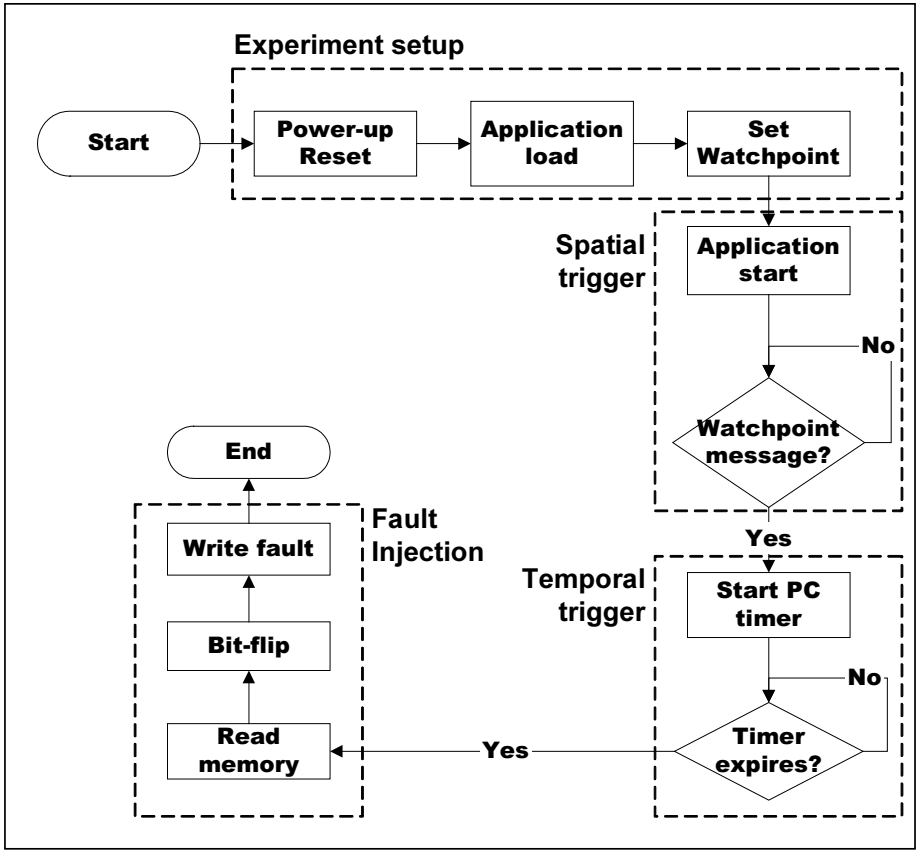


Fig. 1. Procedure for injecting faults in memory with a general trigger

3.3 Fault Injection Procedure

The main advantage of using Nexus to inject faults is the possibility of reading and writing any memory position on-the-fly, i.e., in parallel with the application execution. Thus, it is not necessary to stop the application or to alter its normal execution (or its original structure) neither when injecting a fault nor when monitoring its internal activity. This is why the technique does not induce temporal and spatial overheads in the target system.

During the experiment setup, two memory addresses must be defined. The first, let us say memory address A, is used for synchronising the execution flow of the application with the moment when the injection is performed. The second, let us say memory address B, indicates the memory address where the fault has to be injected.

As commented in Section 3.2, the time of injecting a fault is determined by combining a spatial and a temporal trigger (see Fig. 1). As far as synchronization is concerned, a watchpoint needs to be programmed at memory address A. When the execution flow of the application reaches this memory address, the watchpoint is activated and a timer is then started. In order to avoid the use of system resources that could be allocated to the application, this timer must be programmed externally to the considered target system (for instance in another computer).

Once the timer expires, a fault is injected in memory address B. It is worth noting that faults can be injected in both code and data memory segments. Faults are injected according to the models and procedures explained in Section 3.1.

4 Case Study: An Automotive Electronic Control Unit

The presented technique has been used to study the dependability of an embedded SoC application. The application studied is a reduced version of a real-life diesel engine electronic control unit (ECU), with a reduced number of parameter tables, sensors and actuators, although maintaining the representativeness of this kind of applications.

4.1 The Engine Electronic Control Unit (ECU)

In a diesel engine, there are two different but physically coupled processes requiring control. The first is fuel injection and the second is air management. Fig. 2 depicts the ECU control loops devoted to the supervision and regulation of these physical processes. The control system of our case study ECU is composed of a number of proportional-integral (PI) regulators and on-off controllers configured by parameter tables in which the controller behavior is modeled.

A. Fuel injection control loops

The fuel injection timing and the rail pressure control loops handle the common-rail fuel injection system of the engine.

The fuel injection timing has two inputs: the engine revolutions per minute and the throttle position. According to their values, this control loop computes the timing and the duration of fuel injections.

The rail pressure control is a PI controller in charge of tracking the pressure specified in the parameter tables for each engine condition. The output of this controller is a PWM[2] signal that is applied to the rail valve acting on the pressure.

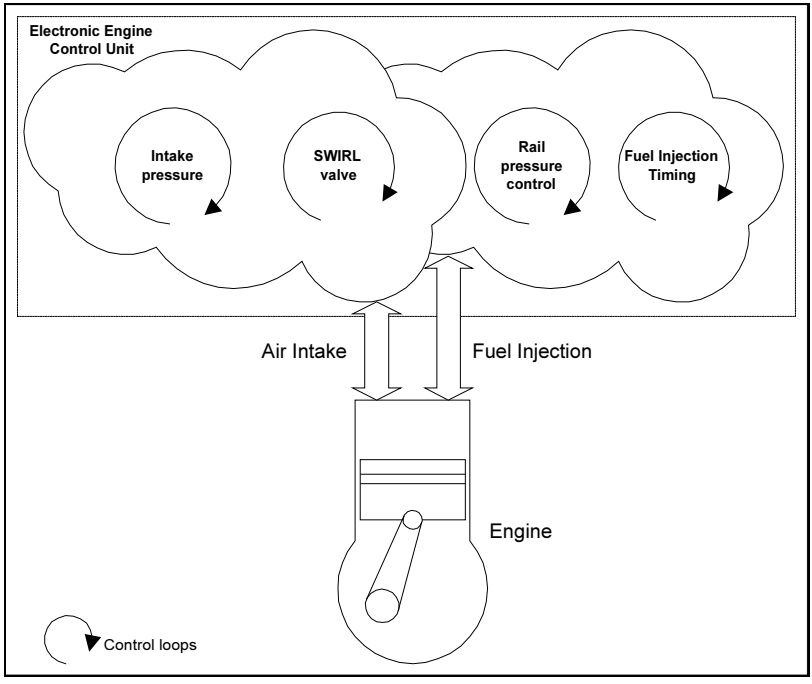


Fig. 2. Automotive engine embedded application

B. Air management control loops.

In the case of the air control loops the inputs to the algorithm are the engine revolutions per minute, the engine load and the intake pressure. The selected regulators can only have two possible states: on and off. The first regulator is a wastegate valve that is used to act on the engine intake pressure. The second one is a SWIRL valve that modifies the flowing of air inside the engine.

---

[2] *PWM* is the acronym of Pulse-Width Modulation.

4.2 Fault Injection Setup

To study the behavior of the ECU in presence of faults, we have developed a fault injection tool implementing our fault injection technique. The name of this tool is INERTE, the acronym of Integrated NExus-based Real-Time fault injection tool for Embedded systems [9]. The three main functional components of INERTE (the experiment generator module, the fault injector and the analysis tool) are depicted in Fig 3. Each one of these modules is briefly described in the following paragraphs.

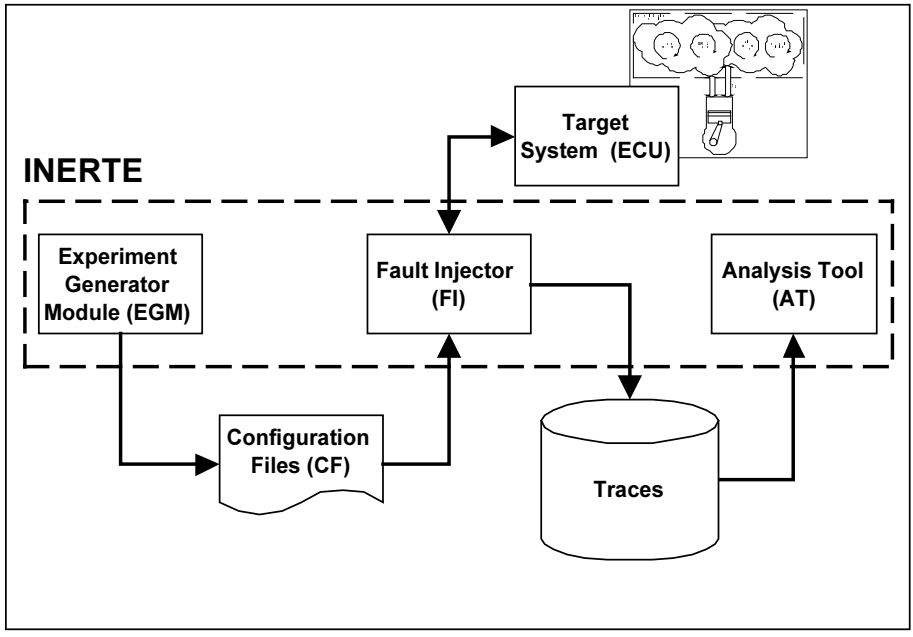


Fig. 3. Block diagram of INERTE injecting faults on the ECU

A. Experiment Generator Module (EGM)

The inputs of the EGM are the memory address range where faults can be injected, the number of injections to be performed and the random distribution (uniform, normal or erlang) of these injections. Using this information, the EGM generates the configuration files (CFs) that the fault injector will later use in order to drive the fault-injection campaigns. Each CF defines the configuration of a fault injection experiment. Basically, it contains the memory address where the fault must be injected, the mask to be used during the injection of the fault and the information required for configuring the spatial and the temporal triggers in each experiment.

B. Fault Injector (FI)

The fault injector is implemented as a script. This script contains complex sequences of processor debugger commands. A commercial Nexus debug tool (Trace32 from Lauterbach GmbH in our case) interprets and executes this script. Inputs for this script are the fault injection parameters that the EGM stores in each configuration file.

Before each experiment, the target system (in our case the ECU) suffers a reset in order to return to its initial state. Then, the ECU begins its normal execution and the FI injects a fault and waits for its activation. As we have already stated, Nexus enables fault injections to be performed without stopping the normal execution of the target system. Furthermore, the activity of the target system can be also traced without any temporal overhead. The end of each experiment occurs after a pre-determined time from the injection of the fault (notion of *observation time*). The trace of the system activity is then stored in a database and the FI starts the next experiment.

C. Analysis Tool (AT)

The AT is a parser that compares the traces obtained from fault-injection each experiment with those associated with fault-free runs of the target system (notion of *golden-runs)*. The AT extracts timing information and searches for differences between these traces. It also deduces whether or not an injected fault has been activated and the error detection mechanisms have been properly triggered. The tool is also able to deduce whether or not an error provokes a failure. In our case, an error leads to a failure when it has an impact on the control outputs the ECU applies to the engine. Latencies between these events are also computed.

4.3 Experimental Procedure

Each fault injection campaign is composed by several experimentation sequences. An experimentation sequence is an arrangement of fault injection experiments with a common fault-free run (golden-run). Fig. 4 shows the procedure followed for running an experimentation sequence in a fault injection campaign. The target application (in our case the ECU) is firstly executed without injecting any fault. The associated execution trace is the golden-run. Then, a number of fault injection experiments are run. The execution trace of each one of these experiments is stored for a later analysis.

In each experiment, the following steps are carried out. First, the target application is reset. Then, a start-up time is devoted to the initialization of the target system. At the end of this time, a (general) trigger is programmed. This trigger has the responsibility of launching the fault injection process according to the parameters defined in the configuration file of the experiment. We call the moment at which the fault is injected in the target application the fault injection time. Although this time can be shorter than the start-up time, this situation is only of interest when studying the start-up of the system in presence of faults. Typically, the fault injection time is

longer than the start-up time. This seems normal since the start-up time only represents a low percentage of the global run-time of the target system. In our experiments, we always consider fault injection times longer than start-up times.

Once the fault injected, it remains dormant until the application exercises the memory location where it is stored. At this moment (notion of fault activation time), this fault is activated, i.e., it becomes an error. The generated error can be detected or not. If it is the case, the error recovery mechanisms execute and two new times can be defined: (i) the error detection latency, which is the difference between the error detection time and the fault activation time (see Fig. 4); and (ii) the recovery time, which can be defined as the execution time of the system error recovery mechanisms.

If an error remains undetected, the activity of the target application must be monitored during the defined observation time. In our particular case, this time must be several orders greater than the time required for running each control loop of the ECU. As a result, the trace retrieved from the system will be rich enough in order to deduce, in case of failure, how the application fails.

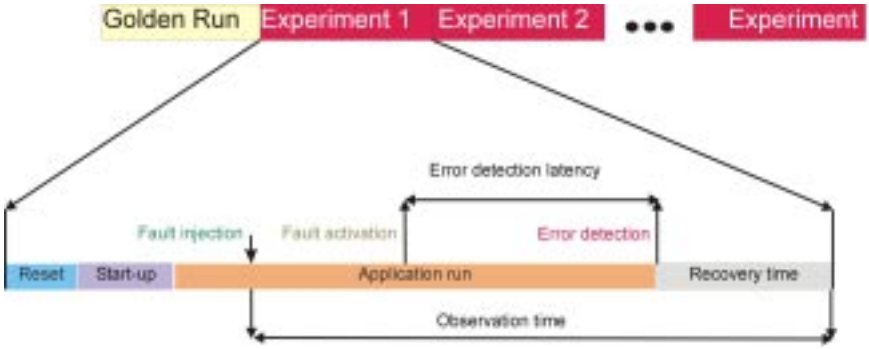


Fig. 4. Experimental procedure

4.4 Experimental Setup and Preliminary Results

For the purpose of our experiments, our ECU runs on a commercial embedded processor evaluation board, the CMD-565 from Axiom manufacturing. The core of our setup is an embedded processor, the Motorola MPC565, providing a Nexus port. On this processor, we run microC/OS-II, a real-time microkernel with a preemptive scheduler. This microkernel provides the run-time support required by the ECU.

In order to interact with the evaluation board, the INERTE fault injector uses a commercial Nexus debugger, named TRACE32-PowerTrace/NEXUS from Lauterbach GmbH. This debugger enables an USB connection between the PC running INERTE and the Nexus circuitry of the MPC565 embedded processor. The debugger is programmable through its own application software, called Trace32, using a scripting language. The PC platform running INERTE is based on a

conventional AMD Athlon XP 1800 executing the Professional release of Windows 2000 operating system.

Using this experimental setup, a fault injection campaign of 3000 experiments was made. 1000 faults targeted read-only data memory segments, the next 1000 experiments focused on read-write data memory segments and the last 1000 faults were injected in code memory segments. Special care was made in avoiding fault injection in empty memory regions. In the non-empty regions, faults were randomly injected following the single bit-flip fault model described in Section 3.1. The input supplied to the ECU was a sequence of acceleration-deceleration of the engine. In the acceleration, the engine goes from 0 to 100% of the engine speed. The reverse process is followed during the deceleration of the engine. It is worth noting that the fault injection campaign is deterministic and thus, it can be reproduced in case of necessity.

Fig 5 depicts the results of the experiments. As this Figure 5.a shows only 12% of the injections performed have been activated. This implies that in the 88% of the experiments the injected faults have not affected at all the behaviour of the system. From that viewpoint, this result indicates that targeting faults with a random distribution in the ECU is not a time effective approach.

Among those experiments resulting in fault activations (see Figure 5.b), 16% of the resulting errors remained dormant. This implies that, during the experiment observation time, the 16 % of the activated faults have not led the system to a failure and have not been detected by the system error detection mechanisms. In order to quantify the effect of these faults in the system behaviour, experiments with longer observation times are required.

Errors affecting the normal behaviour of the ECU are distributed as follows:

- 37% of the errors had provoked an ECU failure. We say that the ECU fails when the control actions it applies on the engine in presence of faults deviates from the ones that it provides in absence of them. A failure in the ECU may provoke some noise and vibration in the engine, but it may also have more catastrophic consequences leading to damage or to stop the engine.

- 44% of the errors have been detected by the error detection mechanisms of the system. As Figure 5.c shows, these mechanisms are MPC565 built-in exceptions. The description of the meaning of each one of these exceptions is out of the scope of this paper. We will also underline that 31% of these faults were "*Machine Check Exceptions*" (MCE). These exceptions are raised by the processor in order to signal the corruption of its internal. The point here is that MCE exceptions are difficult to recover and most of times require the reset of the system.

- 3 % of the errors were also detected but too late. Actually, these errors had led the ECU to provide wrong control outputs (ECU failure) before being internally detected. In this case, the engine may fail although recovery actions were taken in the ECU in order to fix the error.

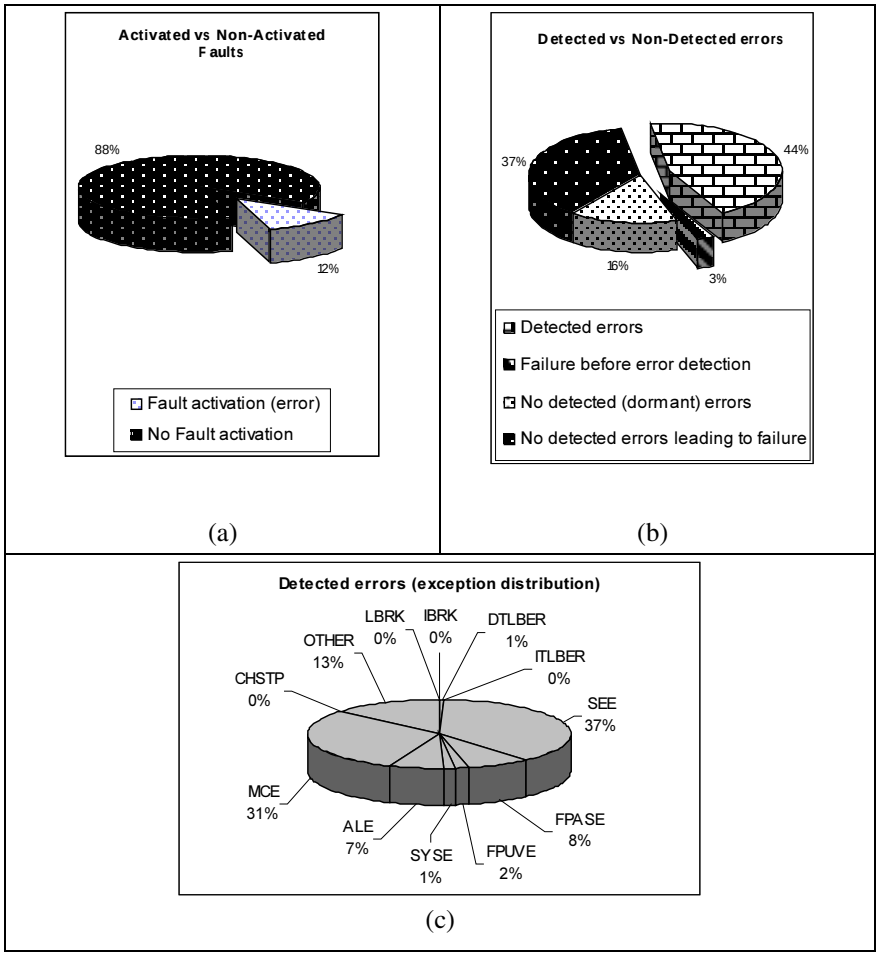


Fig. 5. Experimental results

These preliminary results are quite promising since, as we have shown, they provide a lot of interesting information of the behaviour of the target system (the ECU) in presence of faults. However, the presented results are restricted to coverage measures. We are currently improving INERTE in order to compute also latencies.

5 Related Work

To the best of our knowledge, the only SWIFI tool that uses the debugging mechanisms implemented in embedded processors is an implementation of Flex-Fi [10]. Flex-Fi is an integrated tool with different implementations. One of them [11] uses the BDM (Background Debugging Mode) interface included in Motorola

processors to have access to the processor and control it. BDM is a protocol designed to control an embedded processor through a dedicated serial channel. A dedicated circuitry is included in the embedded processor to execute the normal functions in a debugging environment, such as memory loading, executing, setting breakpoints, etc. Through BDM faults can be injected at run-time on a processor without needing to include external code to the application. This is done without the need of including special debugging code or using any generic processor resource. When the BDM mode is entered, instructions stop executing normally and the special circuitry lets the external commands arriving through the serial port take control. A breakpoint is inserted in a specific word in memory to implement an injection. When executing that word a specified number of times, the execution is interrupted, the memory or the registers that must be injected are modified and the execution starts again. As in most SWIFI tools, the injected fault model is the bit-flip. The Flex-Fi approach has two major drawbacks. First, it produces an important slow down in the execution of the application. Second, it can only be used in those processors including the Motorola BDM port.

Although not focused on embedded processors Xception [6] was the first to use the processor debugging features to carry out the fault injection. It is oriented towards injection on processor functional units and because of that, the way injections are done is dependent on the functional unit being injected. For example, the application is executed until the injection is triggered and enters the trace mode. Then the application is stepped to the instruction to be injected, which is decoded and the parameter to be modified is chosen. In Xception faults can be injected in the processor functional units, such as: Instruction Execution Control Unit (IECU), Integer Unit (IU), Floating Point Unit (FPU), etc, and on memory. Injection trigger can be a combination of the following conditions: operand search or load from a specific address, storage at a specific address and time since application start. To be able to inject on RT systems, in a version of Xception [12] the intrusion is reduced to the same order of magnitude of the RT kernel interrupt latency and inside the workload output jitter.

Another SWIFI tool is MAFALDA [13], which is a tool oriented to the validation of microkernels. As in INERTE, MAFALDA can inject faults by flipping bits in the memory code and data segments. In this case, the aim is to emulate hardware and software faults in the microkernel. So, chosen bits are randomly corrupted in the address space of the microkernel modules. As in Xception, MAFALDA uses processor debugging mechanisms to implement fault injection, but also as Xception embedded processors are not targeted. From the temporal overhead point of view, a version of MAFALDA [14] copes with it by stopping the system clock before the injection code starts executing and restarting it when returning to the system program. With this approach the temporal overhead of the tool is virtually eliminated, but the real physical environment cannot be used.

6 Conclusions

This paper introduces a new technique for injecting faults in embedded (System-on-Chip) applications. The main innovation of this technique relies on the use of the Nexus standard for solving the well-known problems of spatial and temporal overhead that arise when performing software-implemented fault injection. From a high-level viewpoint, the technique addresses issues concerning either how, when and where faults are injected. Furthermore, the use of a standard enlarges the portability of the approach, which can be applied to any embedded system complying the Nexus (class 3) specification.

The development of INERTE has shown the feasibility and the applicability of the proposed fault injection technique. The results obtained from the first experiments performed shows that, as expected, the tool does not interfere with the normal activity of the target application, an engine ECU in our case. The provided indexes can be considered as values characterizing the dependability of the ECU. Following this reasoning, we are integrating our approach in a more general one whose goal will be the dependability benchmarking of such automotive control applications.

On the other hand, the Nexus standard is in phase of revision and its specification remains open. As commented in the paper, the current version of the specification is oriented to the provision of support for the debugging of embedded systems. We are convinced that our work will provide an original and useful input to this specification. In fact, the results that we have obtained shows the interest of the standard (as it is) not only for debugging purposes but also for studying the dependability of SoC-oriented embedded systems.

Acknowledgements. Authors thank to Francisco Gonzalez Ibáñez for his technical advises and the long discussions that have improved very much the quality of this paper.

References

1. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., Martins, E., Powel, D.: Fault Injection for Dependability Validation: A Methodology and Some Applications. IEEE Transactions on Software Engineering, Vol. 16, No. 2, 1990, pp. 166–182
2. M. Sueh, T. Tsai, R. Iyer, "Fault Injection Techniques and Tools", *IEEE. Computer*, pp. 75–82, April 1997.
3. E. Fuchs, "Validating the Fail-Silent Assumption of the MARS Architecture", in proc. 6th Dependable Computing for Critical Applications (DCCA-6), (H. K. A. Avizienis, J.-C. Laprie, Ed.), pp.225–47, 1998.
4. 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, pages 417–426, New York, NY, USA, June 2000. IEEE.
5. T.K.Tsai, R.K.Iyer, D.Jewitt, "An Approach towards Benchmarking of Fault-Tolerant Commercial Systems", in Proc. of FTCS-26, pp.314–323, Sendai, Japan,1996.

6. Cunha, J.C., Rela, M.Z., Silva, J.G.: Can Software Implemented Fault-Injection be Used on Real-time systems?. in Proc. 3rd European Dependable Computing Conference (EDCC-3), Prague, Czech Republic, pp. 209–226, 1999.
7. IEEE-ISTO 5001-1999. The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface. http://www.ieee-isto.org/Nexus5001. 1999
8. Gil, P., Arlat, J., Madeira, H., Crouzet, Y., Jarboui, T., Kanoun, K., Marteau, T., Durães, J., Vieira, M., Gil, D., Baraza, J.C., Gracia, J.: Fault Representativeness. Deliverable (ETIE2) of the European Project Dependability Benchmarking Dbench (IST-2000-25425) funded by the European Community under the "Information Society Technologies" Programme (1998-2002). LAAS-CNRS, Toulouse France 2002.
9. Yuste, P., Andrés, D., Lemus, L., Serrano, J.J., Gil, P.: INERTE: Integrated Nexus-based Real Time fault injection tool for Embedded systems. In Proc. DSN'03. San Francisco, USA. June 2003.
10. Benso, A., Rebaudengo, M., Sonza Reorda, M.: FlexFi: a flexible Fault Injection environment for microprocessor-based systems. SAFECOMP 1999: 18th International Conference on Computer Safety, Reliability and Security, (Lecture Notes in Computer Science, Springer Verlag, A. Pasquini (Ed.)), pp. 323–335
11. Rebaudengo, M., Sonza Reorda, M.: Evaluating the Fault Tolerance Capabilities of Embedded Systems via BDM M. In: Proceedings VTS99: 17th IEEE VLSI Test Symposium, 1999, pp. 452–457
12. Carreira, J., Madeira, H., Silva, J. G.: Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. IEEE Transactions on Software Engineering, vol. 24, 1998. pp. 125–136.
13. Rodriguez, M., Salles, F., Fabre, J.C., Arlat, J.: MAFALDA: Microkernel Assessment by Fault Injection and Design Aid. In: Proceedings of the 3rd European Dependable Computing Conference (EDCC-3), 1999, pp. 143–160
14. M. Rodríguez, A. Albinet , J. Arlat, "MAFALDA-RT: A Tool for Dependability Assessment of Real-Time Systems", in Proc. IEEE International Conference on Dependable Systems and Networks (DSN 2002), Washington DC (USA), 2002.

Constraints on the Use of Boundary-Scan for Fault Injection

Luís Santos[1,2] and Mário Zenha Rela[2]

[1] Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra
3031-601 Coimbra, Portugal
lsantos@isec.pt
[2] CISUC / Departamento de Engenharia Informática
Universidade de Coimbra
3030-290 Coimbra, Portugal
mzrela@dei.uc.pt

Abstract. The Boundary-Scan technology was proposed fifteen years ago to overcome the limitations of testing digital devices due to the increasing complexity and greater miniaturization of integrated circuits and boards. The use of pin-level fault-injection faced similar difficulties and became obsolete for that reason. In this paper we discuss the use of the Boundary-Scan infrastructure for fault-injection purposes. Several fundamental constraints of such approach are identified. Generic digital systems and processors with Boundary-Scan based On-Chip Debugging (OCD) are considered as target system candidates. We observe that by combining OCD mechanisms with modified boundary-scan cells most of the constraints reported are solved. Finally, some key properties of the technology such as the orthogonality to the purely functional architecture and the low abstraction level access as well as the standard interface and description language provided make it a good candidate to provide a standardized flexible fault injection framework.

1 Introduction

The astonishing pace of miniaturization of digital electronics led us to a paradoxical situation: when observation is most needed most of its behavior is hidden inside the integrated circuits.

This is a major problem now that computers pervade every aspect of our lives. From smart cards, mobile phones, automatic teller machines, traffic controllers to life-support medical devices, there is a myriad of complex systems employing digital integrated circuits with multiple functions. Guaranteeing its

† This work was partially supported by the Portuguese Foundation for Science and Technology under the FEDER program of the European Union, through the R&D Unit 326/94 (CISUC) and by the Portuguese Agency of Innovation (AdI) through the project Boundary-Scan for Fault-Injection (BSCAN4FI).

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 39–55, 2003.

correct functioning has become a vital requirement in our technologically driven societies.

For many years fault-injection has been regarded as a major player in dependability assurance. It is commonly used for deriving coverage probabilities, assessing the effectiveness and validating fault and error handling mechanisms. Fault-injection addresses a topic that can't be adequately handled by the more traditional testing techniques, because rather than focus on the correct functioning of systems, it addresses the point of anticipating anomalous behavior under unexpected circumstances.

Many fault-injectors have been developed and described in the literature, with very different approaches on how to disturb the target system. Physical [1, 2], software [3] and simulated [4] fault injection methodologies have been devised for this purpose. A comprehensive description of fault injection methods and tools is given in [5].

The point is that different approaches to fault-injection can lead to different results, since the different system abstractions considered are seldom comparable [6]. Thus different tools provide complementary rather than alternative approaches. For example, while it is true that SoftWare Implemented Fault Injection (SWIFI) provides a level of control and efficiency hardly achievable by other techniques, it is also true that pin-level fault-injection may induce situations impossible to emulate through software (e.g. faults during a system restart) or when the use of SWIFI is cumbersome or simply can't be applied (e.g. Application-Specific Integrated Circuits (ASICs)).

Nevertheless, pin-level fault-injection has been almost abandoned due to several reasons: its target dependency prevents tool reusability, the technical constraints posed by the backdriving effects may lead to target damage and the increasingly high working frequencies makes a controlled disturbance a technically complex goal [7].

So, despite the perception that at the physical level no other technique can provide the same insight on unpredictable failure modes and anomalous system behavior, pin-level fault-injection is restricted to a niche in automotive electronics and some Integrated Circuit (IC) makers.

In this paper we discuss whether and how the Boundary-Scan, an IEEE standard test technology present on digital systems for more than a decade, can be effectively used to support pin-level fault-injection. This research aims at extending the capabilities of Xception [3] — currently a pure software based fault-injection tool — to cover the physical layer abstraction through an integrated fault-injection framework.

The paper is organized as follows: in the next section we present an overview of Boundary-Scan. In Sect. 3 we make a critical analysis on the possible use of boundary scan as a support for fault injection. Several important constraints are then identified. The two following sections present and discuss the related work. Section 6 ends the paper by summarizing the main conclusions of the research.

2 Boundary-Scan Overview

Started in 1985 by the JETAG, the Joint European Test Action Group (later renamed JTAG due to American participation), and next endorsed by IEEE, the Boundary-Scan technology becomes a standard in 1990. Since then it received three revisions, being the last one dated in 2001 and known as IEEE Std 1149.1-2001 — *Standard Test Access Port and Boundary-Scan Architecture* [8].

The driving force behind the standard development were two continuing trends with significant adverse impact on the cost of testing printed circuit boards in the 80's: increasing complexity and greater miniaturization [9]. Complexity makes functional testing longer due to the need to propagate test data through the complex ICs while applying tests to other chips on the board. Miniaturization limits the physical probing, a method through which structural tests were performed.

To overcome those constraints IEEE 1149.1 provides a logical test-access mechanism into the ICs themselves. The boundary-scan path consists of a series of Boundary-Scan Cells (BSCs) added at every digital I/O pin on a component. The resulting Boundary-Scan Register (BSR) and other test features are accessed through a standard interface — the Test Access Port (TAP). Through it, test instructions and test data are shifted in, and test data results are shifted-out.

BSCs allow to control, observe, or take both actions on system signals. Figure 1 illustrates a possible implementation of a control-and-observe cell. A system signal present on a pin can be observed on the rising edge of *ClockDR*, while *ShiftDR* is low. The sample captured is stored on the left flip-flop. Control is achieved through the *Mode* control signal. While high, the value present on the right flip-flop/latch drives the signal output.

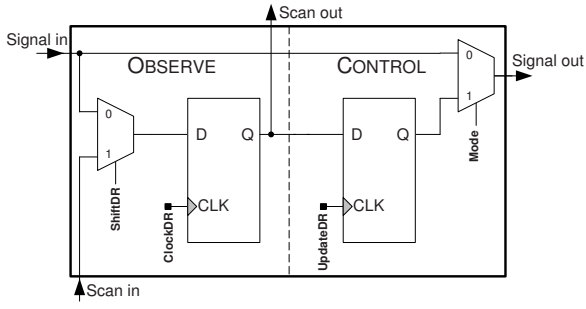


Fig. 1. Universal Boundary-Scan Cell

Figure 2 presents the overall test logic architecture. As shown, the BSR is just one of the test data registers present. All of them share the same generic structure: a shift-register matched with an optional latch. The entire logic is synchronously accessed through the four mandatory pins of the TAP: Test Mode Select Input (TMS), Test Clock Input (TCK), Test Data Input (TDI) and

Test Data Output (TDO). The first two pins drive the TAP controller — a state machine that defines the register present between the last two pins (TDI, TDO) and the sequence of operations performed on it: *capture*, *shift* and *update*. Capture loads the shift register with data present on its parallel inputs. The optional shift operation allows the captured data to be examined on TDO and new input data to be loaded from TDI. Update makes the new data to become active, i.e. loaded on the latch and present on the parallel outputs of the selected register. The optional pin Test Reset Input (TRST*) may be used to asynchronously reset the test logic, so that it becomes transparent to the system logic.

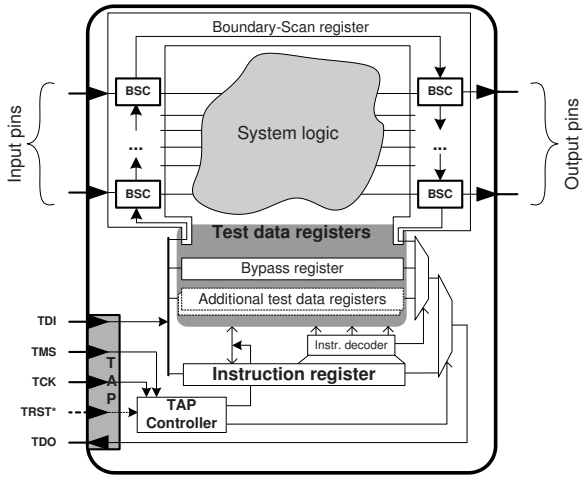


Fig. 2. Boundary-Scan architecture

At a system level the test logic of each individual component is serially chained through TDI and TDO, and parallel controlled by TCK, TMS and TRST* (Fig. 3). The tester begins its activity by shifting in the desired instruction stream into the Instruction Register (IR) to program the test logic of the chained components. As soon as an instruction becomes active on a given component, its instruction decoder selects a single test data register to the TDI → TDO path so the scan chain is never interrupted. By using TMS and TCK it's then possible to command the timing of the *capture* → [*shift*] → *update* sequence against the data registers on the chain. When desired the tester may reintroduce the instruction registers between TDI and TDO, reprogram covered components with new instructions and run a different test.

The IEEE 1149.1 standard consider two sets of public instructions; mandatory: BYPASS, SAMPLE, PRELOAD, EXTEST; and optional (but commonly implemented): INTEST, RUNBIST, CLAMP, HIGHZ, IDCODE, USERCODE. This set may be supplemented with private instructions intended

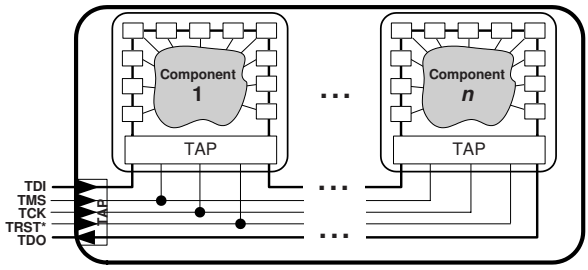


Fig. 3. Boundary-scannable system design

solely for the use of the component manufacturer. The most relevant test instructions will now be briefly described.

The BYPASS instruction selects the Bypass register as the serial path between TDI and TDO during the *shift* operation. The selected register consists of a single shift-register stage and is used to provide a minimum-length serial path to traverse a component out of the effective test.

SAMPLE allows a snapshot of the normal operation of the component to be taken (*capture*) and examined (*shift*) without disturbance. The BSR is selected to the serial path. The PRELOAD instruction allows data values to be loaded onto the latched parallel outputs (*update*) of the BSR before selection of other test instructions.

The EXTEST (external test) instruction allows testing of off-chip circuitry and system level interconnections, i.e. solves one of the key problems for which the standard was proposed. The BSR is selected to the serial path. Its cells at output pins are used to apply test stimuli, while those at input pins capture test results. Captured results are scanned out through TDO while the next set of test input vector is scanned in from TDI. The first stimuli must be loaded with the PRELOAD instruction. The external test is achieved by programming two directly connected components with the EXTEST instruction.

The optional INTEST instruction allows internal testing of on-chip system logic. BSR is again the selected register. Test stimuli are shifted in one at a time and applied to the on-chip system logic, while test results captured by output cells are shifted out. To achieve this, the component must operate in single-step mode while programmed with INTEST.

The optional CLAMP and HIGHZ instructions select the bypass register as the serial path between TDI and TDO. While doing so, CLAMP allows the state of the signals driven from the component to be fixed by the parallel output of the BSR. HIGHZ, by its turn, put all the system logic outputs in an inactive state.

3 Constraints on the Use of Boundary-Scan for Fault Injection

The pin-level fault injection technique accomplished by tools such as MESSALINE [1] or RIFLE [10] is now obsolete due to the increasing complexity and greater miniaturization of digital circuits. As mentioned in Sect. 2, IEEE 1149.1 technology was proposed to deal with similar challenges felt by the test community. So, it seems natural to propose its use as an alternative fault injection technique. This was suggested since the standard's inception. Sedmak in [11], while enumerating potential IEEE 1149.1 applications, speculate about that possibility. He suggested three ways to accomplish this: a) use the SAMPLE/PRELOAD followed by another instruction to expose erroneous values to output pins; b) alternatively, use HIGHZ/CLAMP to obtain the same effects; c) implement a user defined instruction which will force/invert signal values on outputs.

After a brief description of the fault injection steps, this section will identify the most relevant constraints on the use of the IEEE 1149.1 for fault injection. This will show that some of the Sedmak's suggestions are valid only in very particular conditions.

3.1 The Fault Injection Steps

Before applying fault injection the target and its environment must be studied. From that analysis results a fault model. It mainly identifies relevant subsets of the following space:

$$Instant \times Locality \times Dynamics \times Type$$

The fault model is then used to drive the fault injection campaign. During it, each point of the fault model space is evaluated through an experiment that encompasses three main steps, as depicted on Fig. 4.

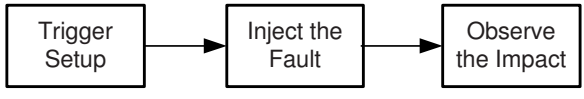


Fig. 4. Steps of a fault injection experiment

First, the trigger condition must be defined. It will identify the injection's *instant*. Temporal (e.g. time since an event), spatial (e.g. access to a given memory address) or event driven conditions (e.g. interruption) are commonly used. Second, the fault must be injected. Its *locality* (e.g. which pin(s)), *dynamics* (for how long) and *type* (e.g. *bit-flip*) fully characterize the process. Finally, its impact must be observed. This step can also be triggered by temporal (experiment timeout), spatial or event driven (e.g error detected) conditions.

3.2 A Fault Injection Experiment

We shall now evaluate if Boundary-Scan can support the fault injection experiment steps. The discussion will be illustrated assuming the target system depicted in Fig. 5. It is composed by two, possible distinct, IEEE 1149.1 compliant components (A and B) directly connected. Both scan chains are serial connected and available at the system level, for injector use, through its Test Access Port (TAP).

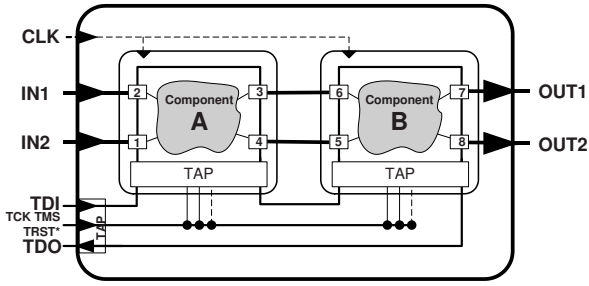


Fig. 5. An IEEE 1149.1 compliant target system

Setting up the Trigger. Let's assume that the trigger condition for a given fault injection experiment is the presence of a specific system input combination. To handle this requirement through BScan the system inputs must be continuously observed through BSC cells 1 and 2. This can be accomplished by programming the test logic of A with the non-intrusive SAMPLE instruction.

The exact sample instant can be accurately controlled by the TMS signal. If the system is asynchronous, to be effective, the sample period must be coherent with the observed signals frequency. Otherwise, the observation process can easily miss a trigger sample. On the other hand, if a clock (CLK) synchronizes the system behavior, the sample instant must occur immediately before (or while) the system itself samples the relevant inputs. So, it is required to *synchronize the* TAP *controller with the system operation, a feature not foreseen by the standard during pure observation* (Constraint C1). External logic and system clock access are necessary to overcome this constraint.

As explained on Sect. 2, to collect the sampled data we must shift it serially through TDO. To reduce the shift time, the scan-chain length must be minimized and the TCK frequency maximized. Although desiring to observe just the first two cells, it is mandatory to observe all the four cells present on A because *the minimum unit covered by any BScan instruction is an entire component* (C2). The length added by the remaining components can be minimized, but not avoided, to one bit per component, programming them with the BYPASS instruction. This reveals another constraint: *the time required to perform*

a test operation on a given component is a function of the system's boundary scan contents where it lives on (C3).

In the scenario considered, 5 TCKs are required to get out the sampled data. Albeit constant, 3 additional TCKs are demanded by the TAP controller's state machine, to become ready to capture the next sample. So, while observing the system considered, samples are prevented from being lost if the following relation is ensured: $f_{CLK} < \frac{1}{5+3} \times f_{TCK}$. Unfortunately, due to the IEEE 1149.1's architecture, the relationship between both clocks actually tends to be the opposite: *the test clock frequency* (f_{TCK}) *is usually a fraction of the target system clock frequency* (f_{CLK}) (C4). A way to overcome this constraint is to directly control the target system clock by external logic — an approach that severely restricts the application of the technique.

Injecting the Fault. Let's assume that we intended to fault the line covered by the BSC cell 3. The mandatory instruction EXTEST comes as the natural choice to inject faults by BScan. According to the standard, while programming A with EXTEST, all the output cells will become controlled — in fact a manifestation of the C2 constraint. This limitation makes fault *locality* control impossible for asynchronous systems and very laborious for synchronous ones.

Assuming we are targeting a synchronous system we must collect the actual contents of the output pins from A immediately before applying the EXTEST. If the injection in course was triggered by a BScan based observation like the one previously described, the data needed is already available. To apply a *bit-flip* fault, the third bit of the observed vector must be inverted, the fault vector PRELOADed and the EXTEST executed. During all these procedures the target system state must not change. For the scenario under analysis all these steps require at least, 14.5 TCKs (assuming that A and B have two bits instruction registers — the IR's minimal size). As can be seen, the demanded ratio between system and test logic frequencies imposed by constraint C4 becomes even more adverse during the injection step.

For a single-clock fault the system must go forward just one clock cycle before the fault is removed. For multi-clock faults (*stuck-at 0/1*) the procedures above must be repeated at every system clock cycle. Both scenarios conflict with constraint C1. Finally, to quickly remove the fault no additional time is required when the TRST* optional pin is available. *Otherwise, a synchronous reset must be performed, demanding 2.5 additional* TCK*s* (C5).

Additionally, BScan also enables to drive outputs to an inactive state (Z). *However, this is usually only accomplished indirectly, by switching a given output control signal common to a set of lines such as an address or data bus* (C6). So, no fine control of *locality* is available for such type of faults, even considering only synchronous targets.

So far it is being assumed that the internal logic of A and its inputs behave normally when the component is programmed with EXTEST. However, this is not the common behavior. *While programmed with* EXTEST*, a component must also be immune to noise combinations of values on its inputs. To prevent*

possible damages or misoperation, BSC cells of input pins may drive constant or preloaded values to the internal logic (C7). When present, this constraint turns EXTEST useless for fault injection.

Finally, if C7 doesn't prevent its use for fault injection, EXTEST may be used with less stringent time constraints for replayable target system where temporal or event driven triggers are used. In such scenario it's possible to program the test logic to sample the system scan chain exactly when the trigger condition takes place. This enables to know in advance the fault vector desired, so that it is possible to have the system PRELOADed with the exact fault on a second run. When reproducing the experiment, the EXTEST must be activated on the trigger condition and its extent controlled as desired (and allowed by constraint C5).

Concerning to the optional INTEST instruction, its use becomes advantageous only when it is useful to inject a fault and immediately observe its propagation, step by step, along the system. Since its adoption mandates single-step operation support, INTEST doesn't raise synchronization problems. To inject a fault similar to the one considered with EXTEST, the fault vector must be first preloaded into the B's BSR. After that, A must be programmed with BYPASS and B with INTEST. Then, the fault vector is applied to inputs of B, the result observed on outputs 2.5 TCKs later, and the captured data shifted out through TDO.

If B is followed by other components with INTEST support, it's even possible to evaluate the fault propagation through the system. To do so we must apply to their inputs the B's previously captured outputs instead.

The CLAMP instruction can fully replace EXTEST (although subjected to the same constraints) because the observation phase is not required. Since the HIGHZ instruction puts all the component's system logic outputs in an inactive state it can only be used to mimic a complete component failure.

Regarding to this step, a final constraint must be pointed out: *according to the standard, after the use of any of the suggested instructions, the on-chip system logic can be in an indeterminate state that will persist until a system reset is applied* (C8). If this behavior really occurs with a given instruction it will be useless for fault injection. Since there is no way to leave the system running normally after the injection, it would be impossible to observe the fault impact.

Observing the Impact. The impact observation step is usually triggered by temporal driven conditions. However, if spatial conditions are adopted, BScan technology can support the process by following the same approach (and accompanying constraints) presented for the fault trigger setup.

Regarding the observation itself, this is a trivial process easily achieved through the SAMPLE instruction. If the study of the fault propagation is required we can also consider, when possible, the use of INTEST instruction as exposed in 3.2.

3.3 Constraints Summary

The constraints previously identified on the use of boundary scan for fault injection are now summarized.

C1 Boundary scan based observation is not synchronized with the observed system operation.
C2 Boundary scan instructions are component-oriented, not cell-oriented.
C3 The time required to perform a boundary scan operation on a given component varies with the system composition it integrates.
C4 The test clock frequency is usually a small fraction of the system clock frequency.
C5 When the TRST* optional pin is not present an injected fault stays active for a minimum of 2.5 test clock cycles.
C6 Even controlling the system clock, *stuck-at Z* faults can not usually have its scope limited to individual cells.
C7 By possibly isolating the component's input pins, the EXTEST boundary scan instruction can not usually be explored for fault injection while the system is running.
C8 Intrusive boundary scan instructions like EXTEST, INTEST, CLAMP, HIGHZ may be useless for fault injection whenever (by design) they leave the system logic in an unknown state.

4 Pin-Level Faults

Several researchers have proposed improvements to the BScan standard to avoid or minimize some of the constraints felt while attempting to use it for fault injection. The modifications suggested consider different boundary scan cell designs, additional cells, new instructions, or hybrid strategies. A brief summary of such work is now presented and discussed.

4.1 Proposals

The first known work proposing the use of IEEE 1149.1 standard for fault injection appears one year after the standard approval [12]. The BSC cells themselves are not modified. However, when covering output pins intended to be faulted, an additional cell must precede them — the injection cell, also integrated in the BSR as depicted on Fig. 6.

The new cell is intended to store the *fault flag*, which states when the signal covered must be actually faulted or not. The original cell stores the *fault* value. Both cells take the role described just when the new proposed Fault Injection (FI) instruction is active — an event that asserts the FIRE signal. In the remaining scenarios the original cell works as expected by the standard. Constraint C2 presented above is eliminated, allowing the injection of multiple and uncorrelated faults while the system is running. During a fault injection, the control mode is now dictated on a per cell basis, and not globally driven to all

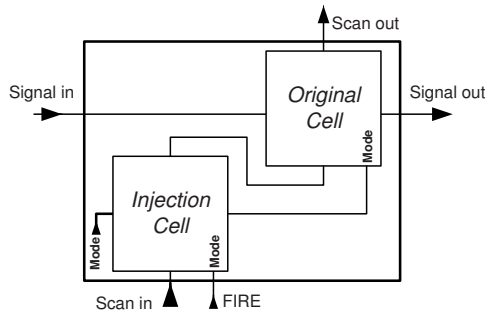


Fig. 6. Wilcox et al. proposal

BSC cells, as happens in a conventional implementation. The obvious drawback of this solution comes from the additional logic required and the double sized boundary scan register.

In 1994 Chau proposed to solve the same constraint using less additional logic [13]. Basically, it was proposed the use of both flip-flops of the original cell to store the injection fault and flag (Fig. 7). To control the extent of the injection an additional pin (Injection Enable) is required. The more limited approach of storing the Injection Enable signal and the injection fault on two bits of the instruction register was also considered. Beyond the additional pin, this design has a serious drawback: the output of faulty pins ripple while the faulty data values are scanned in.

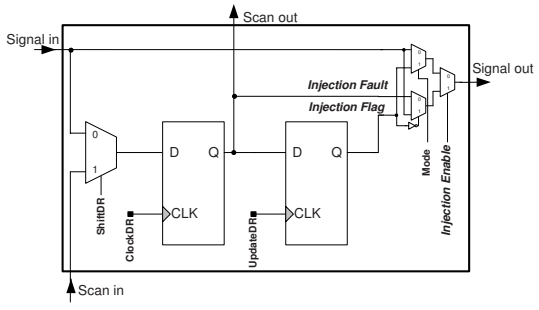


Fig. 7. Chau proposal

In [14] a successful adoption of the Wilcox et al. solution on ASIC of network switches produced by Northern Telecom (Canada) is reported. The early aim of the team was to inject faults through BScan to test the switch's diagnostic software. However, as the structural tests became easier to use at system level the diagnostic software also begun to explore it. This strategy turned the adopted

fault injection scheme useless since the artificial faults injected were flushed during the diagnostic software activity.

In the work cited, Nadeau-Dostie at al. proposed a slightly modified scheme to encompass the new demands. Basically, the FI instruction was modified to behave like a switch that inverts the fault injection state (present/absent). That strategy allows maintaining the FIRE signal asserted while other IEEE 1149.1 instructions are executed. Additionally, the injection cell was also modified to avoid that faulted cells change their behavior accordingly to the current instruction.

The last know work on this subject come from Ke [15]. As the author highlighted, the requirement that all pins are controlled by the same logic limits the BScan usage for many applications. This is actually a more generic view of the C2 constraint. To overcome it, a new instruction, called PINCONTROL, was proposed. While it is active an output pin can be configured in one of four possible states (1, 0, Z, Signal in) depending on values in the cell's flip-flops (Fig. 8).

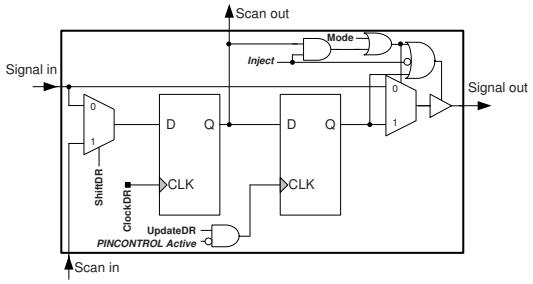


Fig. 8. Ke proposal

Fault injection, considered here just as an application example, can be implemented by considering an additional *Inject* signal. Its waveform, driven by the TAP control state machine when PINCONTROL is in effect, allows injecting *stuck-at 0/1/Z* faults in the desired pins. The fault dynamics can be controlled to an extent of $(1.5 + n)$ TCKs.

4.2 Missing Requirements

All the works cited overlook an in-depth analysis of the fault injection process like the one presented in the previous section. Probably, this explains why they just focus on the second step of the fault injection experiments (Fig. 4), and mainly on C2 — the most obvious constraint but not the only one, as shown.

Three common missing requirements of such proposals are identified. First, no trigger support was discussed. This makes the BScan technology useless to support all the requirements of conventional fault injection experiments. At a first glance, considering a new design for the BSC cell seems to be a prohibitive

approach, since the trigger support will demand a great increase on memory elements present. A more sophisticated approach is needed to cover such requirement. Second, no support for the transient *bit-flip* fault type was provided. The literature is rich in studies (e.g. [16]) that emphasis it as the most representative fault type of digital systems. Even considering a randomized injection approach with the modified cells proposed it is difficult to ensure the single system clock duration of such faults. This gap is strictly related with the last missed requirement: no rigorous *dynamics* control was provided. From the constraint C5 it becomes obvious that a rigorous extent control can only be achieved through a built-in mechanism directly synchronized with the component operation, and not driven by the TAP signals as proposed.

Finally, it is also important to have in mind that the adoption of new designs with specific fault injection support is not foreseen in a near future. First, there isn't a standardized approach. Second, the IC producers, who really define the designs, may not consider cost effective the trade-off between the silicon area increment and the return value obtained. For the moment this path seems to be limited to in-house strategies like the one reported by Northern Telecom.

5 State Element Faults

In the beginning of Sect. 3 several drawbacks were shown on the use of pin-level fault injection. For certain kinds of modern ICs, like complex ASICs, DSPs and microprocessors, an additional limitation must also be considered: the growing pin-hidden operation. This fact, sustained by advanced techniques such as internal memories, pre-fetching and speculative execution, makes acting on the boundary of such circuits a less effective approach. At the same time, despite its specificity, these ICs are actually the more common targets in need of fault injection activity. Fortunatly, several vendors enrich the complex ICs they produce with an extensive set of additional test data registers integrated into the IEEE 1149.1 infrastructure (Fig. 2). According to the standard, these registers may be provided to allow access to any test-support features embedded in the design. These features might include scan-test, self-test registers, or access to functional registers in the design (e.g. via scannable shadow registers).

A work using those test data registers, that claims to use the standard IEEE 1149.1 technology for fault injection, will now be briefly described. The approach followed is then compared with similar works. Finally, the generic adoption of such scheme is discussed.

5.1 BScan Based OCD Fault Injection

In [17] is described FIMBULL (later renamed GOOFI [18]), a tool that use the TAP port to accomplish all the steps required to inject faults on the Saab Ericsson Space's Thor microprocessor. The BScan architecture of the target and the detailed fault injection approach are described on [19]. The Thor microprocessor uses the TAP port to provide access to its OCD mechanisms. This is

accomplished by the presence of several test data registers that cover the cache, the general purpose registers file and the debugging registers.

The trigger condition to inject faults is set by programming the breakpoint registers available through IEEE 1149.1. When a breakpoint condition occurs, execution of the workload stops and fault injection takes place. Fault injection is made by reading the contents of all the scan-chains, inverting the bits stated in the fault model and writing back the fault injected scan-chains to the CPU. After a fault is injected, the execution starts from where the processor was halted and continues until the termination condition is reached (i.e. the workload finishes, an error is detected or a time-out is reached). The system state is then logged by reading the 3032 internal state elements accessible through BScan.

The trigger support and the test/system logic synchronization challenges are solved, not directly through IEEE 1149.1 standard mechanisms, but rather by the IEEE 1149.1 based OCD support that Thor offers. This scheme avoids several of the most problematic constraints identified on Sect. 3. However, a new demand is raised: by relaying on built-in mechanisms, the fault injection tool needs a way to be notified that the trigger condition is already reached and the processor is halted. In this context a new constraint of the standard must be identified: IEEE 1149.1 *assumes that the unit under test is a passive actor of the test activity* (C9). There is no foreseen way in the standard that allows the tested component to notify the tester about some event. Although undocumented, we found that this constraint was solved by the use of an additional TRACE signal pin. When set by the CPU it causes an interrupt to which the tool is able to react upon. Although ingenious, this is a non-pure IEEE 1149.1 approach.

Regarding to the scenario studied on 3.2 another subtle difference must be mentioned. We consider digital lines as fault targets while GOOFI faults state elements with transient bit-flips only. With this approach single-step operation is unnecessary after the injection. Nevertheless, single stepping, a common supported OCD feature, is actually used by the referred injector to study in detail the error propagation.

5.2 Related Approaches

GOOFI actually explores the same OCD mechanisms originally proposed for fault injection support back in 1995 by the Xception tool [3]. Xception accesses these features and provides fault injection through a software module running in the same processor as the target application. The Chalmers's tool brings two significant advantages: it allows to fault locations inaccessible to the software and avoids changes on the workload. However, an important drawback is also present: the temporal intrusiveness imposed is quite high and doesn't scale with target speed improvements (see C4). Reported values on [19] reveals that the injection of a fault by GOOFI requires 21403 TCKs. RT-Xception, an improved version of Xception, requires near 2200 machine instructions executed at full speed in the worst scenario [20]. To compare both approaches we must consider that the maximum f_{TCK} is never greater than 40 MHz and that modern processors run at speeds greater than 1 GHz.

A more recent tool, named FlexFI, also explores the OCD mechanisms of the Motorola MC68332 to perform fault injection [21]. It has however two important drawbacks. First, it explores Background Debug Mode (BDM), a scheme that imposes a proprietary serial debug interface developed by Motorola. Second, it relays on instruction-made breakpoints, rather than conditional breakpoints as proposed by Xception. As reported, this strategy jointly with the half clock frequency imposed by the BDM mode, makes the runtime slowdown by a factor of 70 to 90 times.

5.3 Target Dependence

Although accessing the target system through a standard interface, the approach followed by GOOFI poses some portability limitations. First, using breakpoints to deal with trigger requirements of a fault injection experiment limits the approach to processor-based targets. Second, even being a microprocessor, it is also needed that the OCD mechanism provided is accessible through the IEEE 1149.1 TAP port. Finally, as described above, it is mandatory the presence of an additional target-driven notification path. After surveying we conclude that the second point described is actually a not so strong restriction. Many vendors like ARM, IBM, NEC, Motorola and Intel adopt JTAG based OCD strategies in a great set of products. However, in spite of using a standard access interface, the OCD approaches are considerable distinct. The third restriction is actually weak: the surveyed solutions have usually extra pins, including one reserved to target-driven notifications. Curiously, the most problematic factor comes from a non-technical issue: the information about private IEEE 1149.1 instructions and additional test data registers are confidential. Usually, it is just available to emulator vendors under a Non Disclosure Agreement, nevertheless a practice that fully conforms with the BScan standard.

6 Conclusion

The most relevant constraints on the use of BScan technology for fault injection experiments were identified and generic digital systems and processors with BScan based OCD were considered as target systems candidates. A survey on the related work was presented, and the proposed workarounds discussed and compared.

As a standalone technique, the boundary scan infrastructure, as dictated by the IEEE 1149.1 standard, can't cope with the requirements of a fault injection campaign. However, as demonstrated by previous works, some modifications of the test infrastructure and the on-chip debugging integration allow to achieve significant results. OCD brings support for the initial and the final phases of a fault injection experiment. In fact, defining the injection instant is easily accomplished by programming the breakpoint registers through the TAP port. Additionally, while observing the fault impact, capabilities like single-stepping stresses the value of BScan based OCD integration. Time intrusiveness, a major

limitation of this strategy, can be mitigated in this phase by exploiting more sophisticated OCD resources like internal execution trace buffers as provided by some modern processors (e.g. OpenRISC, Leon, etc.). Finally, during the injection phase, if the modified cells proposed at pin-level are also considered in internal state elements, the most problematic constraint of the technology — location control, becomes solved.

Unfortunately currently there is no target where all of those capabilities are combined. Even if it does it makes sense to question whether the most important aspect of such support is the boundary scan infrastructure. On our perspective the answer is yes. The first reason comes from this technology ability to access all the desired abstraction levels present on the target orthogonally to the purely functional aspects. Technologies such as BDM are usually limited to high level layers such as the instruction set level which force injectors to do fault emulation instead. Moreover, being implemented on top of the instruction set or even the target microinstructions, they pose serious intrusiveness constraints. Second, the BScan access is performed through a standard interface. This factor combined with the Boundary-Scan Description Language (BSDL) ([8]) makes us believe it will be possible to build high portable fault injectores - a reality already in the digital test equipment field. In fact, having the fault injection resources formally described in a text file makes it possible to develop automated fault injection procedures. Finally, the IEEE 1149.1 standard is flexible and extensible enough. All these reasons make boundary scan technology an excellent framework candidate to develop standard rich fault injection support in complex integrated circuits.

Acknowledgments.
We thank Gustavo Alves from Instituto Superior de Engenharia do Porto for its support on the Boundary-Scan technology and Peter Folkesson from University of Chalmers for its availability to clarify some details of the GOOFI tool.

References

1. Jean Arlat, Yves Crouzet, and Jean-Claude Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *FTCS-19*, pages 348–355, Chicago IL, 1989. IEEE Computer Society.
2. Johan Karlsson, Peter Lidén, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–11 13–23, 1994.
3. João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: Software fault injection and monitoring in processor functional units. In *DCCA-5*, pages 135–149, Urbana, Champaign-USA, 1995. Springer-Verlag.
4. Eric Jenn, Jean Arlat, Marcus Rimén, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models: The MEFISTO tool. In *FTCS-24*, pages 66–75, Austin, USA, 1994.
5. Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

6. Johan Karlsson. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *IFIP Working Conf. on Dependable Computing for Critical Applications*, pages 150–161, Urbana-Champaign, USA, 1995. Springer-Verlag.
7. J. R. Martínez, P. J. Gil, G. Martín, C. Pérez, J. J. Serrano. Experimental Validation of High-Speed Fault-Tolerant Systems Using Physical Fault Injection. In *DCCA-7*, pages 233–249, S. Jose, California, 1999.
8. IEEE Std 1149.1-2001. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. New York, 2001.
9. Colin M. Maunder and R.E. Tulloss. An introduction to the boundary scan standard: ANSI/IEEE Std 1149.1. *Journal of Electronic Testing*, 2:27–42, 1991.
10. Henrique Madeira, Mário Rela, Francisco Moreira, and J.G. Silva. RIFLE: A general purpose pin-level fault injector. In *EDCC-1*, volume 852 of *Lecture Notes on Computer Science*, pages 199–216, Berlin-Germany, 1994. Springer-Verlag.
11. Richard M. Sedmak. Boundary-scan: Beyond production test. In *IEEE VLSI Test Symposium*, pages 415–420, Cherry Hill, NJ, 1994.
12. Phil Wilcox, Jim Hjartarson, and Robert Hum. Software verification by fault insertion. U.S. Patent no. 5,130,988 (1992), 1992.
13. Savio Chau. Fault injection boundary scan design for verification of fault tolerant systems. In *ITC-94*, pages 677–681, Washington D.C., 1994.
14. Benoit Nadeau-Dostie, Harry Hulvershorn, and Saman M. I. Adham. A new hardware fault insertion scheme for system diagnostics verification. In *ITC-95*, pages 994–1002, Washington D.C., 1995.
15. Wuudiann Ke. Hybrid pin control using boundary-scan and its applications. In *ATS-96*, pages 44–49, Hsinchu, Taiwan, 1996.
16. Parag K. Lala. Fault Tolerant and Fault Testable Hardware design Prentice-Hall, London, 1985.
17. Peter M. Folkesson, Sven Svensson, and Johan Karlsson. A comparison of simulation based and scan chain implemented fault injection. In IEEE Computer Society, editor, *FTCS-28*, pages 284–293, Munich, Germany, 1998.
18. Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. GOOFI: Generic Object-Oriented Fault Injection tool. In *DSN-2001*, Goteborg, Sweden, 2001.
19. Peter M. Folkesson. *Assessment and Comparison of Physical Fault Injection Techniques*. Phd thesis, Chalmers University of Technology, 1999.
20. João Carlos Cunha, Mário Zenha Rela, and João Gabriel Silva. Can software implemented fault-injection be used on real-time systems? In *European Dependable Computing Conference*, Prague, Czech Republic, 1999.
21. Alfredo Benso, Maurizio Rebaudengo, and Matteo Sonza Reorda. FlexFI: A flexible fault injection environment for microprocessor-based systems. In *SAFECOMP-99*, pages 323–335, Toulouse, France, 1999. Springer Verlag.

A Strategy for Validating an ODBMS Component Using a High-Level Software Fault Injection Tool

Regina Lúcia de Oliveira Moraes[1] and Eliane Martins[2]

[1] State University of Campinas (UNICAMP),
Superior Center of Technology Education (CESET), regina@ceset.unicamp.br ,
http://www.ceset.unicamp.br/~regina
[2] State University of Campinas (UNICAMP),
Institute of Computing (IC), eliane@ic.unicamp.br,
http://www.ic.unicamp.br/~eliane

Abstract. We present a strategy for the validation of an Object Oriented Database Management System (ODBMS), an off-the-shelf component software, when we deliberately introduced faults and observe its behavior in the presence of these faults. A tool designed for the injection of faults in OO applications, specially Java language systems, was used. This Software Fault Injection tool, Jaca, was developed at UNICAMP and has the ability to inject faults in objects' attributes and methods. For the experiments we used an ODBMS performance test benchmark, Wisconsin OO7. The experiments were aimed at validating the robustness of the ODBMS component in the presence of errors originated at the application. For that purpose a fault injection strategy was proposed to help answer the questions: Where to inject? What error model to use? This strategy was applied, so some results and their analysis are presented. Improvement for Jaca were addressed as a experiments' result.

1 Introduction

Software development needs to complete the life-cycle in a short time with a high level of reliability, availability, safety and security. To achieve this, software's project is reducing its custom-built software to give space to third-parties [8] or Commercial-Off-The-Shelf (COTS) components [9], such as a database management systems.
Despite the increase in productivity, the use of components still presents some difficulties, especially with respect of validation and maintenance, because users generally have access only to information regarding the components' interfaces, which can be insufficient for the aforementioned activities. There is no information regarding the quality, the tests undertaken or information on dependencies that would help to analyze the impact of the component on the overall system. So, components tests is important not only to determine whether they provide the expected services but also to check whether they do not present unexpected harmful behavior. Fault injection is useful to achieve the second goal, in which faults are deliberately introduced into a system in order to observe its behavior. Through the acceleration of errors and fail

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 56–68, 2003.

ures this technique is valuable as it may help a user to better understand how robust the software is, achieve reliability in it, analyze how efficient is it when recovering its normal execution after a non successful transaction, what is the impact of its detection and recovery mechanisms on the application's performance which is interacting with it.

When injecting faults into a target system, at least the following questions should be answered: how to inject faults? Which faults to inject? Where to inject in the target application? For the first question, in this study we used the Jaca tool, developed in a former project [15], which is aimed at injecting during runtime, affecting an object's interface, i.e, public attributes and methods. To answer the other questions we proposed a strategy which combines robustness tests as proposed by Ballista methodology [13], and risk-based testing [3] [25]. The strategy is applied in the validation of an ODBMS component - Ozone [21].

The objectives of the experiments carried out were: i) to validate the proposed strategy and verify its effectiveness ; ii) to evaluate Ozone's capacity of detection and recovering mechanisms iii) to evaluate the Jaca tool, its ability to inject faults in order to recommend improvements.

Section II presents some aspects of fault injection and the Jaca tool, as well as similar studies carried out with different databases. In section III, we describes the Ozone ODBMS and the benchmark used to activate Ozone's functions. Section IV, characterizes the proposed strategy, describing the sets F (Faults/errors to be injected), A (mode of Activation of the component), R (data collected during the experiments) and M (measures indicating conformities, or not, with the desired properties), the FARM model [1]. Section V, addresses the injection campaign used in the strategy's application. Section VI addresses the results obtained, while in section VII, we present our conclusions and the next studies to be developed.

2 Fault Injection Techniques

2.1 Fault Injection

Fault injection techniques have been widely used to evaluate the dependability of a system and to validate error handling mechanisms. This technique is useful to validate the solutions designed to handle with exceptional situations. Fault Injection approaches can vary according to the system life time which it is applied and to the type of faults that are injected.

This study uses the software fault injection approach, which consists of altering a system's code or state in order to emulate software faults as well as faults that occur in external components that somehow affects the software[28]. In software fault injection, faults are introduced in a prototype of the system and have the capacity to inject specific error conditions that permit the activation of those mechanisms[19]. It has become more popular due to its lower costs (it does not require specially developed circuits, as does hardware fault injection do), better versatility (it is easier to

adapt codes to make fault injection in another system than adaptation of circuits) and better control, which facilitates the observation of the system during tests.
For software fault injection the more common faults considered are memory faults (alter the content of memory positions), processor faults (affect the content of registers, the result of calculus, control flux or instruction), bus faults (affect the addressing lines or data that are being transmitted by the bus) and, in distributed systems, the communication faults (affect messages that are transmitting through a communication channel: they can be lost, altered, duplicated or delayed)[12].
Recently more attention has been given to the consequences of software faults. Software faults represent the faults resulting from mistakes committed by developers during system development or in modifications made in the phases of maintenance. Software faults have turned out to be the main cause of field failures.

2.2 The Jaca Tool

Software Fault Injection can affect either the code (source or assembler) or the state of a target system. To alter a system's state, a tool is needed to inject faults or errors during runtime. These tools differ according to the mechanism used to trigger faults[12]. Most of these tools are aimed at emulating hardware faults, so faults are injected at low-level, affecting processor registers, I/O device drivers, memory positions. With the increasing importance of software faults, some studies present tools aimed to inject faults at higher level (application source code) during runtime.
Jaca offers mechanisms for the injection of high level faults in object-oriented systems written in Java language. Jaca is an evolution of the FIRE tool [24], uses reflective programming. The reflection mechanism introduces a new architectural model by the definition of two levels: the meta-level (implements fault injection and monitoring features) and the base level (implements the system's functionalities) [16]. Computational reflection allows the target system's instrumentation to carry out their functions through introspection (useful for the system's monitoring) or alter the system during runtime (useful for the injection) without changing the system's structure. Jaca does not need the application source code to perform fault injection. This occurs because Jaca was implemented using the Javassist reflection toolkit [6], which allows the instrumentation to be introduced at byte code level during load time. Jaca's current version can affect the public interface of an application by altering values of attributes, method's parameters and return values. Jaca is described in more detail in [15], [18].

2.3 Related Work

The use of fault injection to validate component-based systems is an active research area. In what concerns the validation of DBMS components there are also some works. In [14] a server is tested in the presence of faults introduced at the NFS (Network File System). Faults were introduced by a human operator, who shut down the machine where the NFS server resided. Another work presents the availability of a client-server application based on the Microsoft SQL Server 2000 [2]. Faults were inserted at the database server using a disk emulator, a machine that is perceived as a

disk by others sharing the same SCSI (Small Computer System Interface) bus, emulating in such way disk and bus faults.
In Coimbra University a group realized a series of fault injection experiments for the validation of different versions of the Oracle DBMS[9] [11] [27]. Most of these works used the XceptionNT tool, which is Windows NT version of the Xception, a fault injector developed by the group [10]. Hardware exceptions are used to trigger a fault injector that operates at the exception handler level.
In Brazil, the Federal University of Rio Grande do Sul also realized studies in database validation using in-house developed tools. In [23], the DBMS Interbase was validated using the FiDE tool, that emulates hardware faults through processor registers and memory corruption. In [17] the validation of the fault-tolerance mechanisms of the DBMS Progress is presented. Two tools developed by the group were used: FiDE, to emulate hardware faults such as memory corruption and disk I/O faults, and ComFIRM, to emulate communication faults with the database server.
Our work differs from those in basically two aspects. First, we use an OO database management system (ODBMS), instead of a relational one. Second, the errors we introduced during runtime were aimed at emulating software faults, as in [11], but fault injection were performed at high-level, affecting attributes and methods (parameters and return values) of Java applications.

3 The Target System

3.1 The Target Component

Ozone is an object manager written in Java which allows for Java persistent objects in a transactional environment. It is an Open Source project under a LGPL license (*GNU Library General Public License*). Persistent objects are programmed following the syntax of the programming language.
The Ozone environment is based on a client/server architecture, where clients connect to the database using “sockets” with a protocol that plays a similar role as RMI (Remote Method Invocation). To avoid object replication, Ozone uses a unique instance architecture, where the actual instance resides in the server. At the client, objects are controlled via “proxy” objects, which can be seen as a persistent reference. To handle these proxies, an external interface is added to each class, which is linked to the original class by the Ozone Pos Processor (OPP) generating a new source file.
Ozone can be used standalone, in which case it resides in the same machine as the client application, or in a distributed architecture, in which case it resides at the server side. The experiments performed so far use a local configuration.

3.2 The Target Application

As in other works relative to DBMS validation by fault injection, we also used a benchmark to activate the target component. Searching on the existing benchmarks, the implementation of Wisconsin OO7 [4] was found on the Ozone’s website [21].

This benchmark was originally written to be used as framework for the development of CAD and CASE tools. The main component of the OO7 benchmark is a set of *composite parts*. Each composite part corresponds to a design primitive such as a procedure in a CASE application; the set of all composite parts forms the *design library*. Associated with each composite part is a *document* object, which documents the referenced composite part. The composite part also has an associated graph of *atomic parts,* which, for example, can represent a variable in a Case procedure. Assembly objects are more complex structures, which may be composed of composite parts (base assembly) or other assembly objects (complex assembly). Assemblies are organized in hierarchies, each of which constitutes a module. Associated with a module is a manual, which documents the module.
The benchmark can create three database sizes: large, medium and small. In this study we are using the small size, which contains a single module or one assembly hierarchy with seven levels, an assembly object in one level being composed of two other assemblies. The assemblies in the lowest level are defined by composite parts with a total of 500 per module, 20 atomic parts each, giving 10,000 atomic parts[4].
The Ozone version of the benchmark implements three main functions: store objects creating an assembly hierarchy (create), search root objects (query match) and traverse the composite part objects' hierarchy, navigating from object to object (query traversal)[4]. These methods were used in the experiments reported here to check database's consistency, as is explained in VI.1.

4 The Fault Injection Strategy

For Jaca tool, an application can be viewed as a set of objects which interact through the exchange of messages, where the errors introduced may affect both the attributes as well as the methods (parameter and return values).
To validate Ozone's robustness we proceeded as follows: i) faults were introduced into the application (OO7) to observe how robust Ozone is with respect to errors occurred in the application using it; and ii) faults were introduced into Ozone to observe its robustness with respect to errors originated in the target component itself. This work reports results from the experiments in (i). A strategy was proposed aimed to answer the following questions: i) where to inject the faults, specifically what objects to inject and what attributes and methods to select within an object; and ii) which error model to use.

4.1 Where to Inject Faults

The idea of risk-based testing is to allocate test effort in the parts of code that are most error prone, and where failure would have the highest impact. So, the first task of risk-based testing is to determine how likely it is that each part of the software will fail [26].One way to determine error prone parts of a system is by collecting field data, as in [5]. When these data are not available complexity metrics can help since the parts of the code that are more complex are more prone to errors[25]. The second task is then to select the complexity metrics to be used. Once identified the metrics, it

is necessary to define guidelines to assist in identifying the risky parts of the code. This is achieved by the establishment of threshold values for the metrics being considered. Classes for which two or more metrics exceed their threshold are considered as risky.
Besides complexity metrics other criteria were used: i)whether the application class has an association with Ozone classes, indicating exchange messages at execution time, propagating, in this way, the errors to the database server; ii)visibility, since we are affecting only the application's public interface, and iii)inheritance of the component.

4.2 Which Error Model to Inject

Based on Ballista's approach for robustness tests, together with the ones proposed in [28], the possible values to be used for each type of data are:

- Integer: 0, 1, -1, MinInt, MaxInt, neighbor value (current value ± 1)[1]
- Real floating point: 0, 1, -1, DBLMin, DBLMax, neighbor value (current value * 0.95 or * 1.05)
- Boolean: inversion of estate (true -> false; false ->true)
- String: null, biggest string, string with all ASCII, string with pernicious file modes and printf format.

These values potentially represent exceptional test values for each data type.

5 Strategy's Application

The strategy can be resumed as showed bellow and will be demonstrated in this section with a case study: 1) Define the complexity metrics to be used; 2) Establish a threshold for these metrics; 3) Obtain the metrics for the target system classes; 4) Identify the classes which have direct interaction with ODBMS; 5) Identify the risky classes: the ones with one or more metrics whose values are beyond the threshold and which satisfy criteria in 4; 6) For each class, apply the same criteria in 2 and 3 for their methods; 7) For each of those methods choose the parameters or return values of types compatible with those types that Jaca can injected; 8)Write a fault specification for each critical value aforementioned in accordance with the selected error model.

5.1 Identification of Classes

The benchmark has 22 classes, divided in 11 interface classes and 11 classes that implement these interfaces. Utilizing RSM software[26], Panorama tool[22] and the Wisconsin OO7 benchmark classes diagram, we calculated the OO software complexity metrics called CK metrics[7], of all these classes and took the classes which have metrics out of the accepted values. The CK metrics were chosen because they were implemented by the quoted tools. From CK metrics we used the following met-

[1] According to [27] the neighbor value (domain twiddle) must not create values out of the limits for the data type.

rics: **NOM** – Number of Methods, **WMC** – Weighted Methods per Class (sum of methods in a class), **DIT** – Depth of Inheritance Tree (The depth of a class within the inheritance hierarchy, i.e, number of ancestor classes), **NOC** – Number Of Children (The number of children, immediate subclasses subordinate to a class in the hierarchy). Table 1 presents the OO7's classes and their corresponding metrics. The values below the metric's names (showed between parentheses) indicate the ideal and the maximum acceptable values of each metric for Java applications[25]. These values were obtained in a research with over 20,000 classes from programs written in both C++ and Java[25]. So, we can use these values, since we are testing Java language applications.

The last three columns of Table 1 show the other criteria considered for class selection: classes which directly interact with the ODBMS's classes; classes that have public visibility and classes which inherits from Ozone's interface. This procedure cover step one to five from the proposed strategy.

Table 1. Identification of classes to be injected

Class	WMC (25;40)[2]	NOM (0;50)	DIT (2;5)	NOC	Direct Association with Component	PublVisibility.?	Inher the component
BenchmarkImpl	45	13	1	0	Yes	Yes	Yes
OO7_Manual Impl	10	10	1	0	Yes	Yes	Yes
OO7_ConnectionImpl	9	9	1	0	Yes	Yes	Yes
OO7_DocumentImpl	7	7	1	0	Yes	Yes	Yes
OO7_DesignObjectImpl	7	7	1	0	Yes	Yes	Yes
OO7_ModuleImpl	7	7	2	0	No	Yes	Yes
OO7_CompositePartImpl	11	11	2	0	No	Yes	Yes
OO7_BaseAssemblyImpl	5	5	3	0	No	Yes	Yes
OO7_ComplexAssemblyImpl	3	3	3	0	No	Yes	Yes
OO7_AssemblyImpl	5	5	2	0	No	Yes	Yes
OO7_AtomicPartImpl	13	13	2	0	No	Yes	Yes

Table 2. Methods of the Classes to be injected by Jaca

Class	Public Attributes	Public Methods				
		Name	Parameter Type	Return Type	Cyclom Complexity	Uses Component ?
Bench markImpl	None	main create getAtomicPartOid	args:String[] anScale:Int None	None None Long	**11** 1 1	No Yes **Yes**
OO7_ Manual-Impl	None	setTitle setId setText title Id text	x:String x:Long x:String None None None	None None None String Long String	1 1 1 1 1 1	No No No No No No

[2] Ideal and acceptable limit values according to [25]. This values were obtained over three years tests, where over 20,000 Java classes where collected and analyzed.

5.2 Identification of Methods and Attributes to Be Injected

Having chosen the classes, the next step consists of selecting public methods and attributes which have data types compatible with those that Jaca can inject. Amongst all possibilities of injection we look for methods which have high complexity and methods which use any component's methods. Then, we choose, from these methods, parameters and return values having compatible data types. The errors were uniformly distributed among these parameters. Table 2 presents some classes and their respective methods, parameters and return values. For sake of space we do not show the totality of the methods here. OO7 classes have no public attributes. The procedures described here cover the step 6 from the strategy.

5.3 Identification of Values to Be Injected

Next we determine values for parameters and return of methods. Table 3 shows part of the table which presents the injected values and the operation realized by Jaca to introduce these values. The apostrophe (') associated with the parameter's name indicates the final value obtained. It is important to mention that for the MaxInt and MinInt (32767 and –32768 respectively) we used an operand a little smaller than the extreme value (32760 was used) to avoid an overflow value. This procedure covers the strategy's seventh step. As the last step, we create the specification and monitor classes files, based on Table 3.

Table 3. Injected values

Parameter: Type→Method	**Operation used for injection[3]**
anScale:Int→ create	Addition (+) anScale'= anScale + 1
	Subtraction (-) anScale'= anScale – 1
	Addition (+) anScale'= anScale + 32760 (extreme integer's values)
	Subtraction (-) anScale'= anScale – 32760 (extrem integer's values)
	Multiplication (*) anScale'= anScale * 0 (critical integer's values)

6 Results and Analysis

As local configuration was used for these experiments, a single machine was utilized, where Ozone and OO7 reside. This machine is a notebook with a Pentium III processor, 366 MHz, 64 MB of RAM memory, and 6 GB of hard disk. The operational system used was Conectiva Linux 6.0 (Linux version 2.2.17.14d). Future tests will be done with a client-server platform, to verify the other aspects related to communication. Further results are presented in [20].

[3] Operations implemented by Jaca according to the type of attribute / parameter / return of method

6.1 Collected Data

Our objective is to observe Ozone's behavior in the presence of faults. To that end, data are collected from several sources. (i) From Ozone's interface: number of stored clusters, exceptions thrown by Ozone, but not treated. (ii) From Benchmark's interface on Jaca: exceptions thrown that are not treated by the application. (iii) From Ozone's log: data that are not similar to those of Ozone's interface. (iv) From Jaca's log: specification of the injected faults. (v) From stored data: to determine whether database consistency was guaranteed. (vi) To check other database properties we implemented further traversals and queries, but for the results presented here only the functions already present at OO7 were needed.

6.2 Characterization of Ozone's Behavior

In order to characterize Ozone's behavior in the presence of application faults we can envisage the following situations: i) ideal case, in which both OO7 and Ozone have normal termination and the database created is in a consistent state. ii) an exception is generated at OO7, as consequence of fault injection, but Ozone has normal termination and the database created is in a consistent state. This case characterizes the robustness of Ozone with respect to application failures; iii) an exception is thrown by Ozone, which terminates abnormally, but the database created is in a consistent state; and finally, iv) Ozone terminates abnormally and the database created is in an inconsistent state. This case characterizes failure of the database manager, in that it allows stored data to be corrupted in consequence of non-successful transaction.
To certify the consistent state of the database, we developed additional functions in order to verify if all transactions committed were in fact stored in the database and no data was lost. We verified all requests in each execution of the benchmark considering the actual request and if interruption occurs, we verify the stored data to certify the non-residual data.

6.3 Description of the Fault Injection Campaign

Table 4 presents the experiments performed. The first column gives an ID for the groups of experiments. The second column indicates whether fault injection was triggered at the first method's invocation or later. We also vary the repetition pattern by injecting faults permanently (at all method's invocations), transiently (faults are injected only once) and intermittently (faults are injected at predefined periods). The third column shows the number of method's parameters or return values of types compatible with those types that Jaca can injected. The fourth column presents the patterns values to numeric types based on Ballista's approach (five patterns values) multiplied by five as each fault was repeated five times as suggested in [4]. Finally, the last column presents the total experiments per start time and repetition pattern.
The sequence of operations used for test was: 1) to open the connection with the Ozone component; 2) to provoke the injection activating the faults through the benchmark; 3) to close the database connection, in case it remains opened after fault injection; 4) to reopen the connection with the database; 5) to verify the stored files

through a Query Match to check whether all atomic parts were created; 6) to verify the stored files through a Query Traversal to check whether the assembly hierarchy was correctly created; 7) to close the connection with the Ozone component; 8) to store the results.

Table 4. Distribution of Tests

Classes of Experiments	Start Time	Repetition Pattern	N°of Integer/ Real Parameters/ Return Values	N° of Injection based on Balista	Total Injection
1P	First Occurrence	Permanent	18	25	450
1T		Transient	18	25	450
1I		Intermittent	18	25	450
2P	After First Occurrence	Permanent	18	25	450
2T		Transient	18	25	450
2I		Intermittent	18	25	450
	Total N° of Tests				**2700**

6.4 Results

The 2700 experiments were uniformly distributed among all methods, that gave a total of 150 experiments per method. A total of 45 failures occurred in this campaign, all of them occurred for the class BenchmarkImpl_Proxy when method *create* was injected. These failures occurred for experiments 1P, 1I and 1T,(at the first method's invocation) These results were observed for the error models: neighbor positive value, MAXINT and MININT. The injection executed with a positive neighbor value, causes the increase of the quantity of information to be stored (observed through the number clusters presented on Ozone's interface), once the injected parameter was used to control the size of the database. Due to lack of storage or memory resources, the application was not capable of being completed and it caused an exception.

In fact, case (iv) mentioned in section VI.2 occurred, in that an exception was raised by OO7 as well as by Ozone and accused on both interfaces, but the object database did not recover its state. This was observed when injecting a positive neighbor value (see VI.1) in which case an exception occurs and the execution was interrupted. After the occurrence, we did a new connection with the database and invoked a query match, where no errors were detected, that is, the root objects were adequately created. However, when the query traversal were invoked, a *NullPointerException* occurred, indicating that the assembly hierarchy was only partially created. Similar situation occurs when injecting maximum and minimum integer or real values. An exception was also raised for both OO7 and Ozone, in the beginning of the execution (*Array Index Out of Bound*) and the same inconsistency on Ozone was observed.

The same tests were conducted by injecting faults beyond the first interaction and none presented failures, not even those that fail during the first campaign, since the fault location *create* method is invoked at the beginning of the OO7 application execution to define the database's size and create the assembly hierarchy according to the defined size , but it is not activated later.

It is worth noting that experiments were performed with Ozone version 1.0. When reporting these results to their developers, they said these problems in Ozone's recov-

ery mechanism was already corrected in version 1.1. In what concerns the fault injection strategy proposed, we could notice that failures occurred, as expected, when injecting the class that presented greater complexity and interacted with the database component. However, differently of what was expected, it was the method *create*, and not the method *main*, where failures occurred.

7 Conclusion and Future Works

This paper presented a strategy for fault injection validation of an ODBMS component, Ozone. To activate Ozone's functions we used a benchmark application, OO7, that was found in the Ozone's Website. For fault injection we used a tool, Jaca, developed to inject faults in applications written in this language. Jaca injects interface faults, that is, it affects methods and attributes at an object's public interface.
The strategy was proposed to help select objects and methods for injection based on potential risk. To identify the risky objects and methods the strategy considers several factors: complexity metrics, interaction with the Ozone component, among others.
The experiments reported here were aimed at verifying Ozone's behavior in the presence of faults in the application. From the 2700 experiments performed, failure occurrence was observed in 45 of them, which was characterized by database corruption.
Results obtained confirmed in part what was inferred by the fault injection strategy, that is, the class with greater complexity that interacts with the Ozone was the one responsible for the observed failures. On the other hand, it was not the method with greater complexity, but the one which implemented a critical function that was responsible for the results reported above. This indicates that the strategy must be further improved to consider other criteria, such as the frequency of use, the importance of the class or method to the target application, among others.
Although further experiments are required, we can state that: i) a strategy based on risk is useful for selecting fault locations when no field data is available; and ii) Jaca is useful to validate off-the-shelf components, for which only the API is available.
Further experiments are being carried out, this time injecting faults directly into the target component's classes. In the long term we also envisage to validate Ozone version 1.1 to compare the improvements with respect to the actual version.

Acknowledgments. Roberto Carlos Torres who implemented an interface to the Jaca tool, making it more user friendly, and also Fabiana Cristina de Souza Alves for implementing new transactions to the actual Ozone's site OO7 benchmark.

References

1. Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J. C.; Laprie, J. C.; Martins, E.; Powell, D. "Fault Injection for Dependability Validation–A Methodology and some Applications". *IEEE Transactions on Software Engineering*, 16 (2), Feb/1990, pag 166–182.
2. Brown, A. "Availability Benchmarking of a Database Systems", EECS Computer Science Division, University of California at Berkeley, 2000.

3. Bach, J "Heuristic risk-based testing" , Software Testing and Quality Engineering Magazine, Nov/1999.
4. Carey, M. J. ; DeWitt, D. J. ; Naughton, J. F. "The OO7 Benchmark" http://www.columbia.edu/, 1994.
5. Chillarege, R.; Christmansson, J "Generation of an Error Set that Emulates Software Faults-Based on Fields Data, 26th International Syposium on Fault-Tolerant Computing, pp 304-13, Sendai, Japan Jun/1996.
6. Chiba, Shigeru. "Javassist – A Reflection-based Programming Wizard for Java". *Proc of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Oct/1998.
7. Chidamber; Kemerer "Principal Components of Orthogonal Object-Oriented Metrics" http://satc.gsfc.nasa.gov, 1994.
8. Clapp,J.A;Taub, A "Management Guide to Software Maintenance in COTS-Based Systems", Eletronic Systems Center, Nov/1998.
9. Costa, D.; Madeira, H "Experimental Assessment of COTS DBMS Robustness under Transient Faults" , Pacific Rim Dependability Computing, Hong Kong,,1999.
10. Carreira, J.; Madeira, H.; Silva, J. G. "Xception: Software Fault Injection and Monitoring in Processor Functional Units". *5th IFIP International Working Conference on Dependable Computing for Critical Applications*. Urbana-Champaign, EUA, 1995, _ág 135–149.
11. Costa, D.; Rilho, T.; Madeira, H " Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault Injection" , New York, DSN 2000.
12. Hsueh, Mei-Chen; Tsai, Timothy; Iyer, Ravishankar. "Fault Injection Techniques and Tools". *IEEE Computer*, Apr/1997, pag 75–82.
13. Koopman, Phil; Siewiorek, Dan; DeVale, Kobey; DeVale, John; Fernsler, Kim; Guttendorf, Dave; Kropp, Nathan; Pan, Jiantao; Shelton, Charles; Shi, Ying "Ballista Project : COTS Software Robustness Testing", Carnegie Mellon University, http://www.ece.cmu.edu/~koopman/ballista/, 2003.
14. Lee, J. Bill; Herwadkar, Rahul V. "Oracle Network Storage System Compatibility Fault Injection Tests" Oracle, www.oracle.com, March/1999.
15. Leme, Nelson G. M. "A Software Fault Injection Systems based on Patterns" Master Thesis, UNICAMP, Brasil, 2001. (in Portuguese).
16. Maes P "Concepts and Experiments in Computational Reflection" Proc. OOPSLA'87, p. 147–155, 1987.
17. Manfredini, Ricardo Augusto, "Conduction of Injection Fault's experiments in Distributed Database", Máster Thesis, Federal University of Rio Grande do Sul, supervising Profa. Dra. Taysy Silva Weber, 2001 (in Portuguese).
18. Martins, E.; Rubira, C. M. F.; Leme N.G.M. "Jaca: A reflective fault injection tool based on patterns" *Proc of the 2002 Intern Conference on Dependable Systems & Networks, Washington D.C. USA*, 23-267 June/2002 pag 483–487.
19. Martins, Eliane "Injection Faults in dependable systems", I Regional Symposium of Fault Tolerance Systems, Campinas, 1996 (in Portuguese), pag 181–196.
20. Moraes, R; Martins, E "Testing Component-based Applications in the Presence of Faults" Proc.of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2003)", Jul/2003
21. Site of Ozone – Object Oriented Database Management System www.ozone-db.org/, 2002.
22. Site of Panorama Tool - www.softwareautomation.com/, 2003.
23. Rodegheri, P. R. "Validation of Fault Tolerance Mechanisms of SGBD Interbase through Fault Injection", Master Thesis, UFRGS, 2001 (in Portuguese).

24. Rosa, Amanda. *Uma Arquitetura Reflexiva para Injetar Falhas em Aplicações Orientadas a Objetos*. Master Thesis, UNICAMP, Campinas, Brasil, 1998. (in Portuguese).
25. Rosenberg, L; Stapko, R; Gallo, A "Risk-based Object Oriented Testing ", 13th International Software / Internet Quality Week (QW2000) , San Francisco, California USA, 2000.
26. Resource Standard Metrics, Version 6.1, http://msquaredtechnologies.com/m2rsm/rsm.htm, 2003.
27. Vieira, M.; Madeira, H "Recovery and Performance Balance of COTS DBMS in Presence of Operator Fault" , IPDS 2002, Bethesda, Wahington DC, 2002.
28. Voas, Jeffrey; McGraw, Gary. *Software Fault Injection: Inoculating Programs against Errors*. John Wiley & Sons, New York, EUA, 1998.

Heavy-Ion Fault Injections in the Time-Triggered Communication Protocol

Håkan Sivencrona[1], Per Johannessen[2], Mattias Persson[3], and Jan Torin[3]

[1] SP Swedish National Testing and Research Institute, Department of Electronics, Boras, Sweden
sivis@computer.org

[2] Volvo Car Corporation, Department of Systems Architecture, Goteborg, Sweden
pjohann1@volvocars.com

[3] Chalmers University of Technology, Department of Computer Engineering, Goteborg, Sweden
{d98mp, torin}@ce.chalmers.se

Abstract. In dependable distributed systems, the communication link is a critical component with strict dependability requirements. The Time-Triggered Protocol (TTP/C) was developed to meet these requirements. To validate this design, one node in a TTP/C cluster was injected with faults using heavy-ions. It was a prototype implementation and cluster sizes of four and five nodes were tested. The experimental results show that arbitrary faults in one node can cause inconsistencies in the cluster and jeopardize the operation of correctly working nodes and the whole cluster. Further, the system's vulnerability to arbitrary failures in single nodes for a cluster with a broadcast bus is shown. Experiments with varying cluster sizes indicate a relationship between cluster size and system vulnerability thus it seems to be important to further analyze if and why cluster sizes need to be taken into account when validating distributed systems. The described inconsistencies resulted from asymmetric value faults, asymmetric timing faults or arbitrary single node failures.

1 Introduction

The communication component is a critical link in any distributed computer system. For safety critical systems, TDMA, Time Division Multiple Access, communication concepts will likely be the best alternative to meet dependability requirements. The Time-Triggered Protocol, TTP/C, is a TDMA protocol designed to handle highly dependent applications implemented in distributed networks and is described in [1].
TTP/C basically provides four services; clock synchronization, clique avoidance, deterministic message sending, and membership service [2]. To avoid collisions on the communication bus, the local time in all nodes within the cluster must be synchronized to establish a global time in the cluster. Due to different clock drifts in the local nodes there is a need for clock synchronization mechanisms. A message that

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 69–80, 2003.

is received in a TTP/C system is considered valid as long as it is syntactically correct and on time. The nodes are only responsible for the delivery of correct messages to the other nodes and the data that the application wishes to send is not checked for correctness. Further, the membership service is a function that allows a communication system to maintain and establish a consistent view of correct working nodes in a cluster at any time.

TTP/C is designed to tolerate any single fault in a cluster. Further, TTP/C should isolate and tolerate single node failures during normal synchronized operation. Faults that only affect single logical units should also be tolerated. In [1] it is further assumed that the controller is free of design faults.

The membership agreement protocol, clique avoidance algorithms, and clock synchronization in the TTP/C protocol, have been thoroughly verified against its specification with the single fault assumption, using both fault-injections [3, 4] and formal methods [5]. Nevertheless, the behavior of distributed systems is very complex, which makes it hard to verify.

Verification only examines if an implementation meets its specification, while validation is the process of investigating if the system will fulfill its required purposes, if implemented according to the specification. Therefore, validation also captures faults in the specification. One overall purpose of verification and validation is to remove deficiencies, both in the specification and the implementation. Early experiments in the FIT project [4] indicated a need for more detailed investigations of TTP/C, some of these results are described in [6].

The purpose of the experiments presented in this paper was to continue this validation of TTP/C by fault injections in a single node, particularly for the fail silence property. This was achieved by stressing the system with heavy-ion fault-injections. The experimental results are related to the membership service that is offered in the bus-topology and the C1 implementation of the communication controller for the Time Triggered Protocol [1]. It is shown that faults in a single node can jeopardize the operation of other correctly working nodes in the cluster as well as the whole communication system.

The remainder of the paper is organized as follows. Section 2 describes the method and the heavy-ion fault injection used to test TTP/C, particularly the experimental setup. In Section 3, the experimental results from both four-node and five-node clusters are presented. These results are analyzed in Section 4. Finally, the paper is concluded in Section 5, in which possible solutions to detected impairments and future work is described.

2 Method

To test the fault tolerance of a TTP/C cluster with arbitrary faults, the communication controller in one of the nodes in a cluster was exposed to heavy-ions from a Californium-252 source with a half decay time of 2,5 years.

2.1 Heavy-Ion Fault-Injection

Fault-injection using heavy-ions is a very useful technique to inject a system with faults [7]. The ions can hit parts in the silicon structure that cannot be reached by other methods such as software fault injection. Hence, it is possible to do a more thorough testing of circuits by combining heavy-ion fault-injections with other methods. However, it is not possible to control the timing or location of the fault, neither to avoid multiple-faults with heavy-ion fault-injections.

2.2 The Test Set-Up

The system used for the experiments consisted of four and five nodes connected to a broadcast bus. Fig. 1 shows a five-node test set-up. There were also some experiments with a nine-node set-up. All nodes in the cluster had a brake-by-wire application allocated to it. Node 4 in the cluster was exposed to heavy-ion fault-injections, denoted FI-node. Monitoring software was implemented in all host processors. This software was used to log the nodes' status register data from the Communication Network Interface [1], CNI, into a ring buffer and to detect faulty messages. The logged information contained operational data of the cluster such as reception time of messages and membership information. The FI-node was also equipped with a current guard and a reset device. The current guard protected the FI-node from latch-ups. In case the FI-node did not reintegrate within specified time, Node 3 could automatically reset it via the reset device shown in Fig. 1.
One experiment lasted until a node, other than the FI-node failed or the whole synchronization collapsed. The experiments include several faults that were handled correctly by the system. Once a node failed it was not allowed to transmit and entered a freeze state. At this point in time, the hosts received an interrupt and sent the logged data about the FI-node's behavior, to a PC via RS232. The received information was saved and the cluster was reset for the next experiment. All fail silence violations of the FI-node were logged to relate the number of fail silence violations leading to cluster failure from the violations that was handled correctly by the cluster.

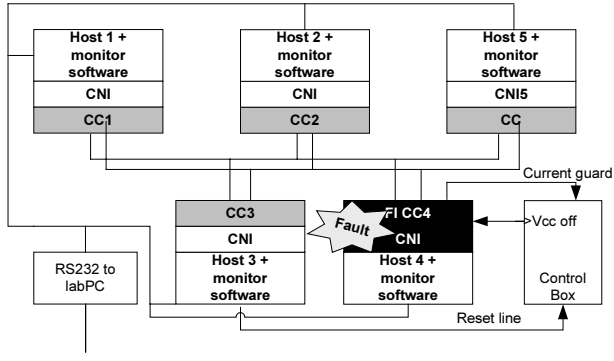


Fig. 1. The system under test with five nodes connected to a physical broadcast bus. Each node consists of a communication controller, CC, Communication Network Interface, CNI, and host

The communication controller chip in the FI-node was mounted on a printed circuit board inside a vacuum chamber. Inside the chamber, the Californium-252 source was mounted on top of the chip as seen in Fig. 2. Since Californium-252 heavy-ions have low dynamic energy, the injected silicon device had to be exposed to the heavy-ions with its lid removed. Further, it was possible to adjust the average fault-injection rate with the air pressure. To control the air pressure in the chamber, it was equipped with a valve and connected to a vacuum pump. In this set-up, the failure rate of the fault-injected node was approximately calibrated so that one fault manifested every fourth second. This could be compared to the length of one communication cycle, which was 20ms throughout the experiments. Hence, in average there was one fault manifestation every second hundred TDMA round. This is considerably larger compared to normal reintegration time that takes three to twelve TDMA rounds.

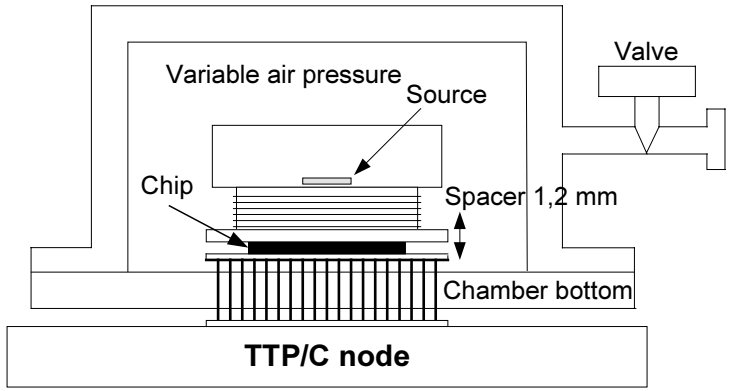


Fig. 2. The vacuum chamber with the Californium-252 source placed above the chip with the lid removed

3 Results

An overview of the results from the fault injections, from the two different clusters with four and five nodes, is presented in this section.

An example of collected data is shown in Table 1 and Table 2. Table 1 shows data logged by Node 2 during three TDMA rounds in the four-node set-up is shown in Table 1. The rows, contain CNI status register data where the columns represent information such as local membership vector (Memb), controller state (C-ST) as well as error diagnosis field (Diag), logged from several TDMA slots.

Table 2 shows the last two TDMA slots as shown in Table 1 but logged by Node 1. In Table 1 and Table 2, Node 1 and Node 2 differ in their opinion about the last shown FI-node transmission. This is a typical logged asymmetric fault.

Following the experiments, the Diag column in the status register printouts, as seen in both tables, was analyzed off-line. The complete documentation of the logged scenarios can be found in [9].

Table 1. Status register information logged by Node 2 revealing a fail-silence violation (Diag = 0202) where "02" represents an invalid frame

Node 2											
Sending node	**Memb**	**CTime**	**C-ST**	**Stat**	**-W2-**	**-W3-**	**-W4-**	**-W5-**	**Diag**	**-W2-**	**Time d**
1	000F	3433	3418	0000	000A	0001	030A	5000	1818	0606	6
2	000F	3DEF	3DC8	0000	000A	0001	030A	5800	1818	0000	0
3	000F	47B3	4778	0100	000A	0001	030A	5000	0808	0303	3
FI	000F	51E3	5128	0200	000A	0001	030A	5000	0808	0303	3
1	000F	5B3C	5B28	0000	000A	0001	030A	5000	0808	0303	3
2	000F	64FF	64D8	0000	000A	0001	030A	5800	0808	0000	0
3	000F	6EC3	6E88	0100	000A	0001	030A	5000	0808	0808	8
FI	0007	789D	7838	0100	010A	0001	030A	5000	**0202**	1616	22
1	0006	8200	8238	0101	010A	0001	030A	A200	1414	0606	6

Table 2. Status register information logged by Node 1 of the last two TDMA slots as logged by Node 2 in Table 1. In contrast to Node 2, Node 1 receives a correct frame (Diag = 0808) where "08" represents a correct frame

Node 1											
Sending node	**Memb**	**CTime**	**C-ST**	**Stat**	**-W2-**	**-W3-**	**-W4-**	**-W5-**	**Diag**	**-W2-**	**Time d**
FI	000F	78F3	7838	0000	000A	0000	FD0A	5000	**0808**	1010	16
1	000F	82B6	8238	0000	000A	0000	010A	5800	1808	0000	0

3.1 Results from the Four-Node Cluster

Table 3 shows the distribution of the failure causes that were detected during 78 fault-injecting experiments in the four-node cluster with a total number of 37.054 manifested faults. Of these faults, 16.502 were experienced when limited logging capabilities was implemented. Most manifesting faults, null frames, did not result in fail silence violations. The FI-node behaved correctly, it froze, restarted and reintegrated. However, in approximately 12% of all detected errors the FI-node continued its transmission breaking the fail-silence assumption, which led to that the system had to resolve a more threatening situation. In 99.8% of all detected faults, the cluster remained synchronized, which means that the error detection mechanisms together with the fault handling mechanisms in the FI-node and the system were successful. This percentage is the system's coverage factor. Hence, 0,2 % of all manifested and detectable faults lead to some sort of system degradation.

Table 3. Distribution of the causes for the 78 system failures out of 37.054 experienced faults for the four-node's cluster

Failure type	**System failure**	**Fraction of all faults leading to system failure**
Asymmetric time fault	21.8%	0.046%
Asymmetric value fault	14.1%	0.030%
Babbling idiot	1.3%	0.003%
Reintegration failure	62%	0.132%
Total	**100% (78)**	**0.211%**

3.2 Preliminary Results from the Five-Node Cluster

Table 4 shows the distribution of the failure causes that was detected during 16 fault-injecting experiments in a five-node system with a total amount of 12.707 manifested faults. Another five experiments with 22.337 manifested faults presented in Table 5, had a slightly changed set-up where the FI-node was reset after 50 TDMA rounds if it did not reintegrate autonomously. There is a small difference in system coverage between the two set-ups, however it is too small to draw further conclusions.

Table 4. Distribution of the causes for the 16 system failures out of 12.707 experienced faults for the five-node's cluster without automatic reset

Failure type	System failure	Fraction of all faults leading to system failure
Asymmetric time fault	100%	0.13%
Asymmetric value fault	0%	0.0%
Babbling idiot	0%	0.0%
Reintegration failure	0%	0.0%
Total	**100% (16)**	**0.13%**

3.3 Summary of Results

A summary of the experiments for both four and five nodes is shown in Table 5. The injected faults manifested through Cyclic Redundancy Check, CRC, faults or invalid frames. A CRC fault occurs when the checksum is invalid and invalid frames are either unreadable or not on time. These manifestations are depending on the system consequence divided in asymmetric time or value faults, babbling idiots and reintegration failures, and further discussed in Section 4.

Table 5. Detected faults from the fault-injection experiments

Cluster size	Number of experiments	Number of faults	CRC fault	Invalid frame	Fail-silence (null frames)	System coverage
4	39	20804	12.6%	0.87%	86.5%	99.81%
4 (limited logging)	39	16250	N/A	N/A	N/A	99.77%
5	16	12707	11.5%	0.55%	87.9%	99.87%
5 (automatic reset)	5	22337	11.1%	0.51%	88.3%	99.98%
Total	**99**	**72098**	**-**	**-**	**-**	**-**

4 Discussion

The four different failure types resulting in system failure presented in chapter 3 are further analyzed in this section in order to explain possible causes of the system behavior. The failure types are asymmetric timing faults, asymmetric value faults, babbling idiots and reintegration failures. Both described asymmetric faults belong to the Byzantine fault class as described in [10]. The experienced faults are typical faults that could be expected in a time-triggered distributed communication system.

4.1 Asymmetric Timing Faults

Inconsistency in a TTP/C cluster can be the result from asymmetric faults [11]. One type is the Slightly-out-of-Specification, SoS fault [11]. A fault scenario due to SoS faults in the time domain is shown in Fig. 3. The corresponding node partitioning is shown in Table 6. To make the asymmetric timing fault behavior more apparent the shown scenario is from a nine-node cluster. In four- and five-node clusters, the system usually shuts down instead of becoming partitioned.
The scenario started when the nodes in the cluster perceived the arrival time of the FI-node's messages differently. In one TDMA round, TDMA round 3, some nodes (1, 3, 4, 6, and 7) received the message from the FI-node slightly before the end of allowed receive window while some other nodes (2, 5 and 8) received the message slightly too late, according to the nodes' local clocks. This caused an inconsistency in the cluster and the membership agreement was utilized to resolve the situation by forcing the minority clique (nodes 2, 5 and 8) to reintegrate. In the next TDMA round, TDMA round 4, some other nodes (1, 7, and 9) received the messages to late and had to reintegrate.

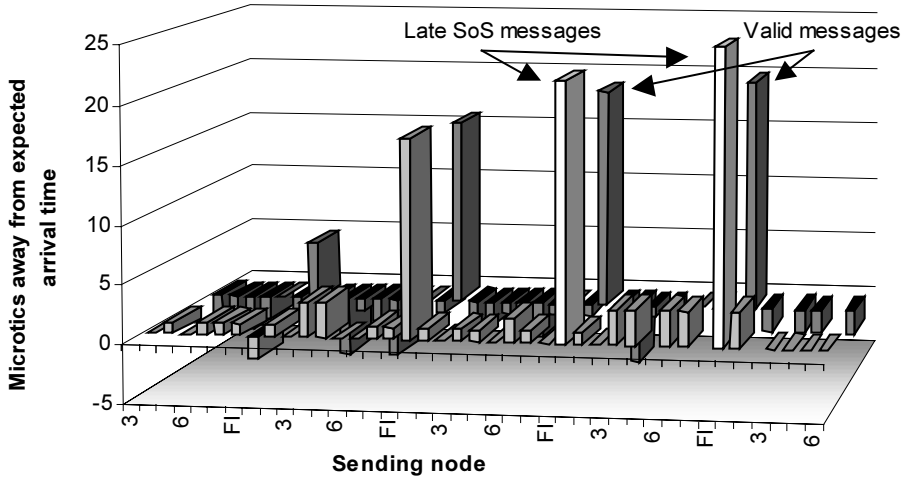


Fig. 3. System behavior as logged by Node 2 and Node 7 after one SoS message sent by the FI-node. The two nodes perceived the message differently during two consecutive TDMA-rounds

Table 6. Detected cliques from the fault scenario shown in Fig. 3

Cluster status	Majority clique	Expelled clique 1	Expelled clique 2
Normal operation	1, 2, 3, 4, 5, 6, 7, 8, 9		
First SoS failure	1, 3, 4, 6, 7, 9	2, 5, 8	
Second SoS failure	3, 4, 6		1, 7, 9

The membership agreement [8] is a service in TTP/C where every sending node of the system is represented by a unique identification, which is stored in all nodes as a

membership vector. The membership vector is implicitly distributed to other nodes every time a node sends a message.
If the membership vectors are different between the sending and receiving nodes, the CRC becomes erroneous. When a CRC error is found, the receiving node raises a membership error for the sending node locally and the corresponding membership vector value is set to false. If a node finds that it is not in agreement with the majority of the active nodes in the cluster, it is not allowed to send and has to reintegrate. In this case the transient loss of nodes could jeopardize the application and thus have impact on the system safety.
The presented failure scenario in Fig. 3 resulted from a persistent fault. It took two TDMA rounds to resolve the fault effect and six out of nine nodes, including the FI-node, had to restart. The clique avoidance algorithm [8] and the membership agreement algorithm resolved the presented inconsistent state but did not explicitly identify the faulty node.
The reason behind most timing faults is clocks with different speed. This motivates the clock synchronization algorithm. However, there must be specified limits for clock drifts to protect the system from bad clocks and degraded performance.
In TTP/C, the clock synchronization information is extracted from the arrival time of the last four messages [1]. The two most extreme clock values are removed and the average value of the remaining clocks is used. The nodes adjust their clocks with a correction term depending on this average, to remain synchronized within the cluster.
In the implementation of TTP/C, used for these experiments, the limit for maximum allowed clock deviation was 20 microtics. A sending node is not allowed to transmit a message if the own clock differs more than half this value from the global clock. If a message is received within the maximum allowed clock deviation, a receiving node will accept it as valid. Otherwise, the message will be regarded as invalid and a membership error will be raised.
If not all nodes have the same view of the received messages, it will result in an inconsistency, which has to be handled, with the risk of judging correct nodes as faulty and thus force them to reintegrate.
One interesting experience was that out of all manifested asymmetric timing failures, 55 % were caused by a sudden, significant change of the FI-node's transmission start while the other failures occurred after a deviation during several TDMA round for the FI-node, as shown in Fig. 3.
In the five-node cluster, SoS timing failures were still occurring and in approximately 80 % of the fail-silence violations the cluster remained synchronized. The same number for the four-node cluster was less than 20 %. This indicates that a larger cluster is less sensitive for faults affecting the cluster synchronization. Otherwise no significant changes of the result were found concerning the appearance of asymmetric timing faults between the two cluster sizes.

4.2 Asymmetric Value Faults

One detected failure type was when the log files showed that two or more nodes had an inconsistent view of one message sent by the FI-node in the value domain. This could be due to different internal states or that the nodes interpreted one bit in the

message differently. Typically in the four-node cluster, two nodes did not accept the message sent by the FI-node while the third node did. This caused an inconsistent state in the system where the membership agreement functionality was important to resolve the inconsistency. It is interesting to note that the order of acceptance and rejection of the message affected the outcome of the inconsistency, i.e. whether the clique containing the faulty node or the clique with correct nodes had to reintegrate.

4.3 Babbling Idiots

A broadcast bus is by default hard to protect from common mode failures such as short circuits and babbling idiots. TTP/C has a bus guardian functionality to protect the time slots from babbling idiots. However, this protection mechanism is not independent from the communication controller. Therefore, the bus is more sensitive if one node fails uncontrolled. Such failures could destroy other nodes' communication.

In the case shown in Fig. 4, the FI-node was sending in the wrong time slot, affecting the transmission of Node 2. Thus Node 2 was declared faulty by Node 1, itself and Node 3. In the next time slot the transmission of Node 3 was also destroyed by the FI-node. The FI-node itself is erroneous. Therefore, Node 1 was not allowed to send since it was out of synchronization, even if it otherwise was correct. Before this scenario the FI-node was detected sending correctly.

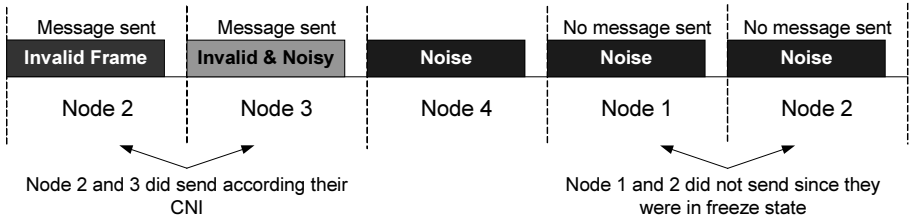


Fig. 4. A communication blackout where a Babbling idiot corrupts the sending Node 2 and Node 3

Even though Node 1 and Node 2 had stopped sending they detected noisy traffic in their slots. This was most probably the FI-node that was still sending on the bus, as a babbling idiot. During this fault manifestation no nodes were able to reintegrate since traffic was detected and prohibited reintegration. The FI-node had to be manually reset.

Methods to prevent this failure from occurring are to either have stronger fault containment regions, an independent bus guardian on node level or through a star topology [12]. This failure type was not detected in the five-node set-up, which could be explained by too few experiments.

4.4 Reintegration Failures

A reintegration failure manifests when any node that reintegrates deflects from its dedicated time slot and affects others' transmissions. To reintegrate the FI-node sends a cold start frame [1] on one channel. Since TTP/C uses redundant channels, this reintegration message is not capable of destroying other nodes' transmission completely.
However, for the four-node cluster a noteworthy reintegration failure was documented. When the FI-node restarted and initialized it did not necessarily detect any ongoing traffic or the time when it should reintegrate was wrong. When the cold start frame collided with other traffic, the other nodes did, in contradiction to the specification, not accept the correct redundant frame. Together with a missing transmission from the FI-node in the next slot, the system collapsed.
This was the most common cause for system failures in the four-node cluster. However, the system should have remained synchronized since three nodes, and under some circumstance even two nodes, are enough to maintain a synchronized cluster. This fault is likely to be an implementation fault since it cannot be mapped into a specified behavior or by violation of the fault assumption. The reason behind the reintegration errors for the four-node cluster is partially explained in [12]. No reintegration errors were detected in the five-node cluster.

5 Conclusions

The results show that heavy-ion fault injection is efficient in stressing silicon designs to arbitrary failure modes. Coverage against arbitrary faults is intuitively the only realistic approach for safety critical applications but difficult to achieve. Distributed communication systems must thus be designed with careful consideration to an arbitrary fault hypothesis, including asymmetric faults.
In the experiments the local fault detection and fault handling mechanisms in the FI-node were effective, but did not guarantee protection against all faults. Between 11 and 14 percent of all internal faults remained undetected by the FI-node, since the messages were sent, and affected the operation of other nodes.
The presented failures could be the result of both implementation and specification faults. Failures due to specification faults are fundamental TTP/C issues and we believe that failures due to asymmetric faults need to be further considered and handled by the TTP/C protocol specification since membership agreement in combination with asymmetric faults became a path across the fault containment regions. This has also been addressed and investigated recently when a star-coupler was proposed, designed and tested using software fault injection, SWIFI and heavy-ions. No asymmetric faults where shown to propagate during these experiments [12].
One possibility to protect the system against arbitrarily malfunctioning nodes is a redesign of the membership agreement protocol together with related fault handling mechanisms since the current implementation cannot guarantee that a faulty node is identified or restarted. The current membership protocol resolves inconsistencies through a pessimistic approach where the minority clique needs to reintegrate. We

believe that partitioning due to asymmetric faults could be resolved smoother. The authors are currently investigating this issue, including decoupling of timing and value errors from the membership agreement protocol.
Specifically the experience gathered during the experiments shows that systems relying on only four nodes are not advisable to have in any distributed safety critical design for TTP/C. TTP/C is specified to be fully operational and fault tolerant with only four nodes in the cluster implementation. However, it cannot be considered fully robust since the system is sensitive during reintegration sequences and to asymmetric faults. The cluster cannot establish a three-node system in the presence of asymmetric faults. Three nodes is the minimum for basic mechanisms to work acceptably, e.g. clock synchronization.
One interesting conclusion is that a validation process based on one single cluster size is not valid for a general system. The four-node cluster showed a slightly different failure pattern and failure types, e.g. reintegration errors. This conclusion can be supported by formal verification and number theory. Ongoing experiments include tests with cluster sizes of six to nine nodes to determine the robustness against asymmetric faults and algorithms to tolerate SoS timing faults [13].
Further, it is difficult to derive useful fault coverage without investigating the impact of application, different set-ups etc. This is a problem for all fault injection methods.

Acknowledgements. This research was partially funded by the European commission, IST-1999-10748 as a part of the FIT project, Fault Injection in the Time Triggered Architecture as well as SP Swedish National Testing and Research Institute, Volvo Car Corporation, The Program Board for the Swedish Automotive Research Program at The Swedish Agency for Innovation Systems and through a Honeywell University grant.
The authors furthermore wish to thank the staff at the Department of Subatomic Physics for their kind support concerning provision of the Californium-252 source.

References

1. Kopetz, H.: TTP/C Protocol, Available at http://www.ttpforum.org
2. Kopetz, H. Grünsteidl, G. and Reisinger, J.: Fault-Tolerant Membership Service in a Synchronous Distributed Real-Time System. Technical Report, Technical University Vienna.
3. Karlsson, J. Folkesson, P. Arlat, J. Crouzet, Y. Leber G. and Reisinger, J.: Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture. In the Proceedings of DCCA-5, Urbana-Champaign, IL, USA, September (1995)
4. FIT project at http://www.cordis.lu/ist/projects/99-10748.htm
5. Rushby, J.: Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. IEEE Transactions on Software Engineering, Volume 25, Number 5, September, pp: 651–660, (1999)
6. Sivencrona, H. Johannessen, P. Torin, J.: Protocol Membership in Dependable Distributed Communication Systems – A Question of Brittleness, Paper no: 2003-01-0108, SAE World Congress, Detroit, USA, (2003)

7. Karlsson, J. et al.: Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms, IEEE Micro, 14 (1), pp.8–23 (1994)
8. Bauer, G. Paulitsch, M.: An Investigation of Membership and Clique Avoidance in TTP/C. In Proceedings of The 19th IEEE Symposium on Reliable Distributed Systems, pp. 118.124, Nuremberg, Germany (2000)
9. Sivencrona, H. Persson, M.: Detected Errors in a Time-triggered System Utilizing Heavy Ion Fault Injection. Technical Report no: 02–13, Department of Computer Engineering, Chalmers University of Technology (2002)
10. Driscoll, K. Hall, B. Sivencrona, H. Zumsteg, P.: Byzantine Fault Tolerance, from Theory to Reality, In proceedings of The 22nd International Conference on Computer Safety, Reliability and Security, (SAFECOMP), Edinburgh, Scotland (2003)
11. Ademaj, A.: Slightly-Off-Specification Failures in the Time-Triggered Architecture. Seventh Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'02), Cannes, France, (2002)
12. Ademaj, A. Sivencrona, H. Bauer, G. Torin, J.: Evaluation of Fault Handling of the Time-Triggered Architecture with Bus and Star Topology. In the proceedings of the International Conference on Dependable Systems and Networks, DSN 2003, San Francisco, CA, USA (2003)
13. Sivencrona, H. Persson, M. Torin, J.: Using Heavy-ion Fault Injection to Evaluate Fault Tolerance with respect to Cluster Size in a Time-triggered Communication System, In proceedings of The 6th Design & Diagnostics of Electronic Circuits & Systems, (DDECS), Poznan, Poland (2003)

Dependability and Performance Evaluation of Intrusion-Tolerant Server Architectures*

Vishu Gupta, Vinh Lam, HariGovind V. Ramasamy,
William H. Sanders, and Sankalp Singh**

Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801, USA
{vishu, lam, ramasamy, whs, sankalps}@crhc.uiuc.edu

Abstract. In this work, we present a first effort at quantitatively comparing the strengths and limitations of various intrusion-tolerant server architectures. We study four representative architectures, and use stochastic models to quantify the costs and benefits of each from both the performance and dependability perspectives. We present results characterizing throughput and availability, the effectiveness of architectural defense mechanisms, and the impact of the performance versus dependability tradeoff. We believe that the results of this evaluation will help system architects to make informed choices for building more secure and survivable server systems.

1 Introduction

Intrusion tolerance [6] is an approach to handling malicious attacks, in which the impracticability of making a system fully secure against all attacks is recognized and intrusions are expected, but the system is designed to provide proper service in spite of them (possibly in a degraded mode). Intrusion tolerance has the potential to become a very useful approach in building server architectures that withstand attacks. Several such intrusion-tolerant server architectures have been conceived in both academia and industry, including KARMA [7], ITSI [14], ITUA [3], and PBFT [2]. However, there has not been any comparative study of their performance and dependability. There are many challenges in doing such a study. First, it is difficult to identify representative architectures that cover the various design possibilities for building intrusion-tolerant architectures. Second, the problem of coming up with detailed yet reasonably high-level models of chosen representative architectures that could be comprehensively evaluated is a fairly complex one. The models should represent the design differences between architectures without getting tied down to low-level details. Third, coming up with appropriate measures that bring out the relative strengths and weaknesses of the representative architectures is a complex problem in itself.

* This research has been supported by DARPA contract F30602-00-C-0172.
** Names are in alphabetical order. Authors made equal contributions to the research.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 81–101, 2003.

In this paper, the above challenges are addressed for the first time (to the best of our knowledge), and a fairly comprehensive comparison of intrusion-tolerant server architectures is presented. We realize that given the many variations in implementing intrusion-tolerant systems, any comparative study is feasible only if we identify *classes* of intrusion-tolerant architectures and limit our comparison to abstract architectures that are representative of these classes. In this work, we identify four classes of intrusion-tolerant server architectures based on how requests are handled and how decisions are made in response to intrusions. In modeling the effectiveness of these classes of intrusion-tolerant architectures, we realize that the performance and dependability of these intrusion-tolerant systems cannot be quantified in a deterministic manner, because the systems do not provide complete immunity to all possible intrusion methods. An attractive option for evaluating intrusion-tolerant systems is via probabilistic modeling [11], as shown by Singh et al. [15], who validated an intrusion-tolerant replication system, with variations in internal algorithms, using probabilistic models.

In this paper, we evaluate and compare the strengths and weaknesses of the four architectures in probabilistic terms. We use Stochastic Activity Networks (SANs) [12] as our representation of the models for the architectures. By varying the parameters of the models, we obtain information about performance and intrusion tolerance characteristics of the different architectures.

2 Intrusion-Tolerant Server Architectures

We consider intrusion-tolerant architectures that follow a *client-service system* paradigm (for example, a web browser as a client and a collection of web servers as the service system). All such systems are based on replication of information across a set of servers, and rely on a distributed architecture that routes incoming requests among several server nodes in a user-transparent way. All such systems also have some mechanism by which the incoming requests are spread among the servers. We consider only those mechanisms for routing the requests among the server nodes that do not require the clients to know that there are replicated servers in the service system and that do not divulge any information about which of the replicated servers actually service a particular client's request. This "hiding" of the servers from clients is necessary for anonymity and security purposes. Client-based, DNS-based, and server-based routing mechanisms (see [1] for a detailed classification of the various approaches for routing requests among multiple servers) do not satisfy the requirement of "hiding." The appropriate routing mechanism is the dispatcher-based approach, in which a single virtual IP address is used for the entire service system. The dispatching mechanism could be centralized, in which case it would route requests to individual servers, or it could be logically distributed among the servers, in which case the requests would be multicast to the servers.

We explored the design space for intrusion-tolerant systems that satisfy the above criteria, and identified the following dimensions along which architectures can vary: (1) how the client requests get routed to the servers, (2) whether the

decisions to reconfigure the system in response to intrusions are made centrally or in a distributed manner, and (3) whether multiple requests are served concurrently by different servers. Based on the above, we partitioned the design space into four classes. In this paper, we model four abstract architectures, each of which is representative of one of those classes. All four architectures that we evaluate have the following components in one form or another:

Client: The client is a program, like a web browser, that establishes connections to the service system in order to satisfy user requests.

Service: This component implements the protocols to service an incoming client request. For example, it could be an HTTP server.

Intrusion Detector: This component could be a combination of multiple third-party intrusion detection tools and protocol-specific intrusion detection (in which violations of the protocol specification are treated as intrusions).

Configuration Manager Daemon: The Configuration Manager Daemon (or CMDaemon for short) uses the Intrusion Detector component to keep track of whether or not the service has been compromised, and implements strategies for recovering from attacks. There is one CMDaemon component for each Service component. Each CMDaemon monitors one Service component and may run in the same host as that Service component.

Configuration Manager: The Configuration Manager receives reports from the CMDaemons about the well-being of the Service Components that they monitor. It decides how to recover when an intrusion is reported, and instructs the CMDaemons about this decision. Each CMDaemon then implements those instructions in their respective Service components.

Gateway: This is the component whose IP address is known to the clients as the IP address of the service system. It serves as the dispatcher that controls the routing of the client requests to the Service components, helping to mask the identities of the Service components' operating systems and the service application. In architectures that do not have the Gateway component, all the servers receive all the client requests. That is done in various ways; for example, all the servers could be configured to be members of an IP multicast group. Clients would send their requests to this multicast address.

Firewall: This component filters incoming requests based on certain policies.

Database: The Database component is the store for the information that clients want to access. In this paper, we are not concerned about the exact organization of this component. Interested readers are referred to [5].

The four architectures differ in how the above components interact with each other, their placement, and which of them are trusted. A "trusted" component is one that is assumed not to fail. We now describe each of the four architectures in more detail.

Centralized Routing Centralized Management (CRCM). The goal of the CRCM design is to employ a small number of trusted components to protect a large set of servers and databases. In this design, a Firewall component filters the incoming requests, looking for signatures of commonly known attacks. The Gateway is a trusted component. An incoming request passes through the

Firewall to reach the Gateway, which then forwards the request to a randomly chosen server from the active server set. The Gateway also masks server-specific and OS-specific information from all the replies. The service system consists of a large collection of servers. They share the same filesystem, but may run different operating systems and different web-server software versions. In addition to the server software, each host that is part of the service system also runs a CMDaemon, which is responsible for detecting attacks via various mechanisms (e.g., integrity-checking of various critical files and checking of the process states). The CMDaemons report the health of the local server to the Configuration Manager, which is a trusted component. The Manager continually checks the integrity of the CMDaemons. If there is an intrusion detection, the Manager cleans the server state, and could roll back the potentially erroneous transactions committed by the intruded server. The Manager informs the Gateway about the current active server set. The Gateway uses that information in the selection of servers to process client requests.

Multicast Routing Centralized Management (MRCM). The MRCM architecture achieves intrusion tolerance through hardened, heterogeneous platforms. This hardening is achieved by embedding firewalls in each server host, and having extensive alert and intrusion-detection capabilities in each server host. Those capabilities form the CMDaemon component. There are no additional front-end firewalls like those in CRCM. Scalability is achieved through the ability to add additional platforms easily, and maintainability is achieved through the ability to remove and service platforms easily. All the servers receive all the requests sent to the single virtual IP address of the service. The service rules on each server determine what traffic to process and what to throw away. For example, rules could be based on the source IP address of the client. In essence, those service rules form a load-balancing policy. The load-balancing policy could be changed at the behest of the Configuration Manager (for example, when an intrusion is detected and the intruded host shut down), and the clients previously serviced by the intruded host would need to be distributed among the correct hosts. When an intrusion is detected, the Configuration Manager could instruct the servers to implement the new load-balancing policy by giving them an updated set of service rules. Through the CMDaemon on a host, the Configuration Manager could also update the filtering policies on the host-embedded firewalls so that traffic from specified clients is blocked or audited.

State Machine Replication (SMR). The SMR architecture employs a state-machine-replication-based approach [13] that tolerates malicious faults. A replication protocol that tolerates Byzantine faults, similar to [2], could be used (with some modifications to ensure user transparency) for this architecture. The requirement for an algorithm tolerating Byzantine faults is that it must have at least $3f + 1$ servers, where f is the number of simultaneous faults that need to be tolerated. SMR does not require an extensive firewall like those in the CRCM and MRCM architectures. Unlike CRCM and MRCM, there is no centralized trusted Configuration Manager and local CMDaemons. Instead, the Configuration Management is now distributed among the servers. The distributed Con-

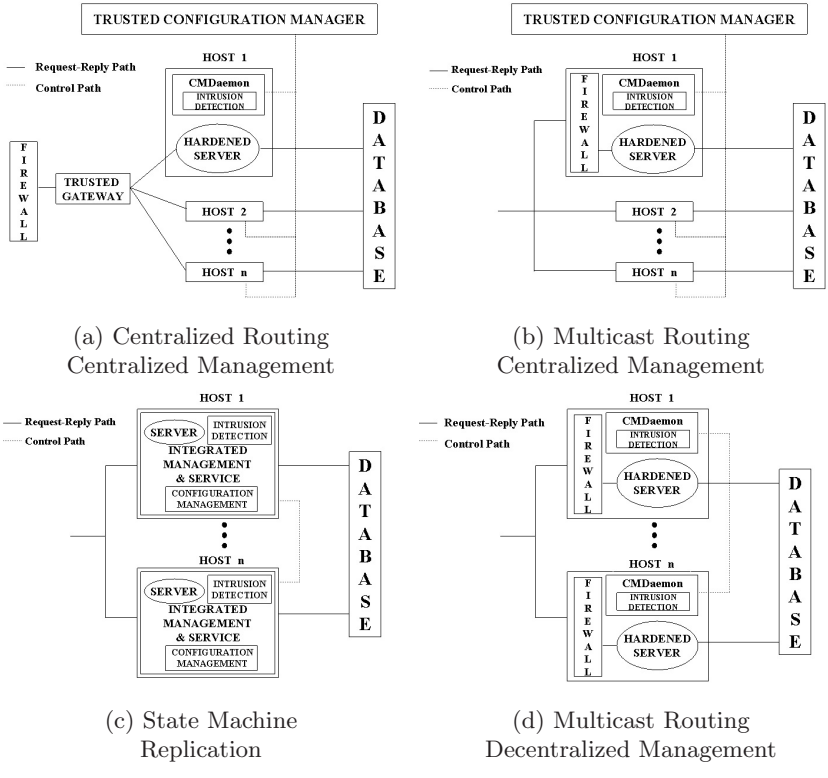


Fig. 1. Architecture Block Diagrams

Table 1. Summary of the design features of the four architectures

Feature	CRCM	MRCM	SMR	MRDM
Parallelism in processing requests	Yes	Yes	No	Yes
Strict correctness of replies guaranteed	No	No	Yes	No
Configuration Manager	Centralized	Centralized	Distributed	Distributed
Required number of servers for uninterrupted service when f servers are compromised	f+1	f+1	$3f+1$	$3f+1$
Forwarding of client request by Gateway	to a randomly selected server	to all servers	to all servers	to a randomly selected server
Servicing of request	by the randomly selected server	based on source IP	by all servers	by the randomly selected server
Trusted components	2	1	0	0

figuration Management and Service components are integrated into one logical unit. This integrated Management and Service unit is replicated across the set of servers, and the Byzantine-fault-tolerant protocol ensures that all correct servers maintain consistent state information for this integrated unit. As in MRCM, all requests reach all the servers. The set of servers processes one request at a time. The servers agree on the reply to be sent to the client, as well as on any updates to be made to the back-end database, through a Byzantine agreement protocol. SMR ensures that all replies sent to clients and updates made to the database are correct, as long as there are no more than f simultaneous corruptions in the system (we call this the *Byzantine agreement requirement*), but involves a large performance overhead due to the fact that all the requests are serialized and processed by the entire set of servers one at a time.

Multicast Routing Decentralized Management (MRDM). The MRDM design is a hybrid of the previous 3 architectures, and tries to achieve a trade-off between the better throughput performance achieved by the parallelism of the CRCM and MRCM architectures, and the strict correctness achieved by the SMR architecture, without relying on any trusted components. It does so by separating the service component in the SMR architecture from the configuration management. As in the SMR architecture, the Configuration Manager is distributed across the host nodes. However, unlike in SMR, the server nodes do not all process the same request at the same time. A firewall component embedded in each host (similar to the one in MRCM) could be used to filter out incoming requests based on specified policies. The incoming request is randomly routed to one of the servers (like in CRCM). Each host runs a server component and a configuration management component (which represents an integrated Configuration Manager, CMDaemon, and Intrusion Detector component). The servers can process requests independently from each other (unlike in SMR), but the configuration management components across all the hosts coordinate with each other, distribute knowledge about intrusions, and come to agreement about the configuration changes that need to be made in response to intrusions. At the core of the configuration management component could be an intrusion-tolerant group membership protocol (such as the one in [10]) that requires the participation of at least $3f + 1$ nodes to tolerate f simultaneous faults. By separating the service component from the management component, we are able to retain the parallelism of the CRCM and MRCM architectures, and by distributing the management component, we remove the need for having a central trusted Configuration Manager. However, MRDM does not guarantee strict correctness of replies (as SMR does), since the intruded node could still be servicing some requests, and potentially sending erroneous replies, during the time period between the intrusion of a node and the detection of the intrusion. The SMR architecture, on the other hand, masks the effects of a subset of intruded servers, as long as the threshold requirement of f is satisfied.

2.1 Assumptions and Attack Model

We assume *staged* attacks, which means that there is a non-negligible time between successive node infiltrations. That gives the defense some time to react. None of the above architectures can defend against a situation in which all the hosts are simultaneously intruded. They also cannot defend against a situation in which the attacker intrudes the various nodes in stages, but the compromised nodes show no observable signs of an intrusion until all the nodes have been intruded (this is essentially the same as the first situation). For the staged attack assumption to be true, node failures must not be strongly correlated. That could be achieved, for instance, by running different implementations of the service code and/or the operating system.

Within the staged attack model, there could be two kinds of attacks on a single host: *multi-phase attacks* that require a sequence of attacks in order to successfully compromise the host (for example, an attacker could upload a file line-by-line using the Windows "echo" command), and *single-phase attacks* that successfully compromise the host in one shot (for example, the attacker could guess the correct password and gain root access on the first attempt).

The CRCM and MRDM architectures employ *dispersion*, i.e., because of the random selection of servers by the Gateway, requests from the same client could be processed by different servers. That decreases the probability that different phases of a multi-phase attack will reach the same server. That, in turn, increases the time required to exploit any single web server using multi-phase attacks.

3 SAN Models for the Intrusion-Tolerant Architectures

Stochastic Activity Networks, or SANs, are a convenient, graphical, high-level language for capturing the stochastic (or random) behavior of a system. A SAN has the following components: *places* (denoted by circles), which contain tokens (the term "marking" is used to indicate the number of tokens in a place) and are like variables; *tokens*, which indicate the "value" or "state" of a place; *activities* (denoted by vertical ovals), which change the number of tokens in places; *input arcs*, which connect places to transitions; *output arcs*, which connect transitions to places; *input gates* (denoted by triangles pointing left), which are used to define complex enabling predicates and completion functions; *output gates* (denoted by triangles pointing to the right), which are used to define complex completion functions; *cases* (denoted by small circles on activities), which are used to specify probabilistic choices; and *instantaneous activities* (denoted by vertical lines), which are used to specify zero-timed events. An activity is enabled if for every connected input gate, the enabling predicate contained in it is true, and for each input arc, there is at least one token in the connected place. Each case has a probability associated with it and represents a probabilistic choice of the action to take when an activity completes. When an activity completes, one token is added to each place connected by an output arc, and functions contained in connected output gates and input gates are executed. The output

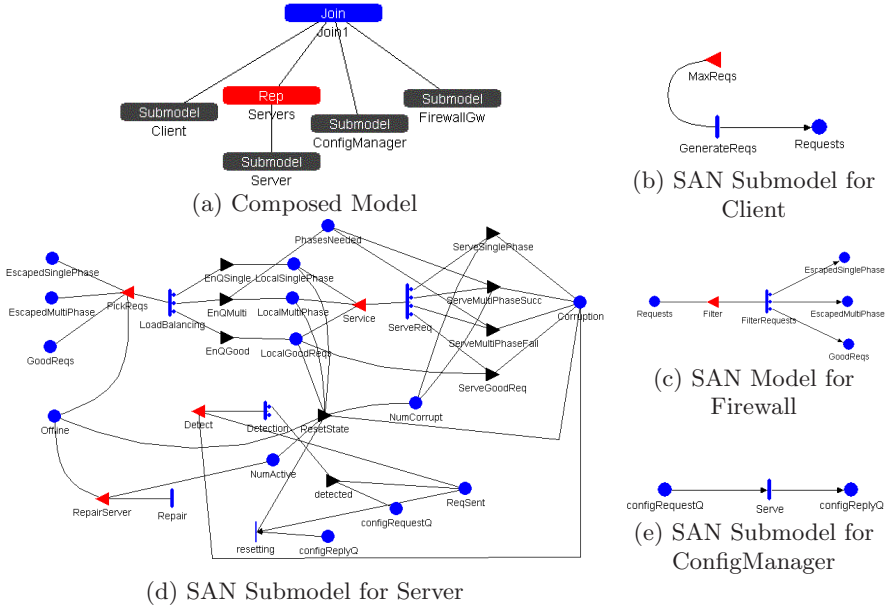
gate and input gate functions are usually expressed using pseudo-C code. The times between enabling and firing of activities can be distributed according to a variety of probability distributions, and the parameters of the distribution can be a function of the state.

We have modeled the four architectures described in Section 2 as composed SANs. Atomic models were built for various components of each architecture, and complete models were then built using replicate and join operations. The salient features that we have modeled for each architecture include generation of client requests and attacks, organization of firewalls and filtering of requests, organization of servers and distribution of requests to servers, servicing of requests and effect of attacks, detection mechanisms, system reconfiguration upon detection of corruption, and repair of affected components. We have used exponential distribution for the timed activities in all the models. We believe this is a realistic assumption, because the request arrival process and servicing of requests by servers (especially web servers) are largely memoryless, and hence are well-represented by exponential inter-arrival times and exponential service times. Single-phase attacks and the subsequent phases in a given multi-phase attack are generated with some probability on the incoming requests; hence, they also have an exponential distribution in our SAN models. We developed that approach in order to keep the attack model fairly simple; we focused the complexity in the models to reveal the differences among various architectures. We understand that we may need sophisticated attack models in order to model the intrusion response behavior of the architectures more accurately. That may be the focus of another study. Due to space limitations, here we provide only a high-level description of the models of the individual architectures. A much more elaborate description of the SAN models is presented in [8].

Centralized Routing Centralized Management (CRCM). The composed model for CRCM (Figure 2(a)) consists of four atomic SAN submodels: **Client**, **Server**, **ConfigManager**, and **FirewallGw**. The **Server** submodel is replicated `NumServers` times, where `NumServers` is a global variable indicating the number of hosts running servers. Since requests have to pass through a firewall and a gateway before they are distributed to individual servers, we have a single unreplicated **Client** SAN (Figure 2(b)) to model the generation of incoming requests from the clients.

The **FirewallGw** SAN in Figure 2(c) models the firewall that filters incoming requests with known attack signatures. We model general attacks, including the ones that are not malformed client requests, as a part of the request stream. That is acceptable, since the request stream models the path all attacks follow (all packets pass through the firewall to reach the servers), and since effects of single-phase and multi-phase attacks are similar (they result in corruption of a server).

The **Server** SAN in Figure 2(d) models the centralized distribution of client requests to individual servers, servicing of requests, corruption of servers due to attacks, dispersion of multi-phase attacks, and detection of corruption and the system's response to it. The local place *Corruption* keeps track of the level



(a) Composed Model

(b) SAN Submodel for Client

(c) SAN Model for Firewall

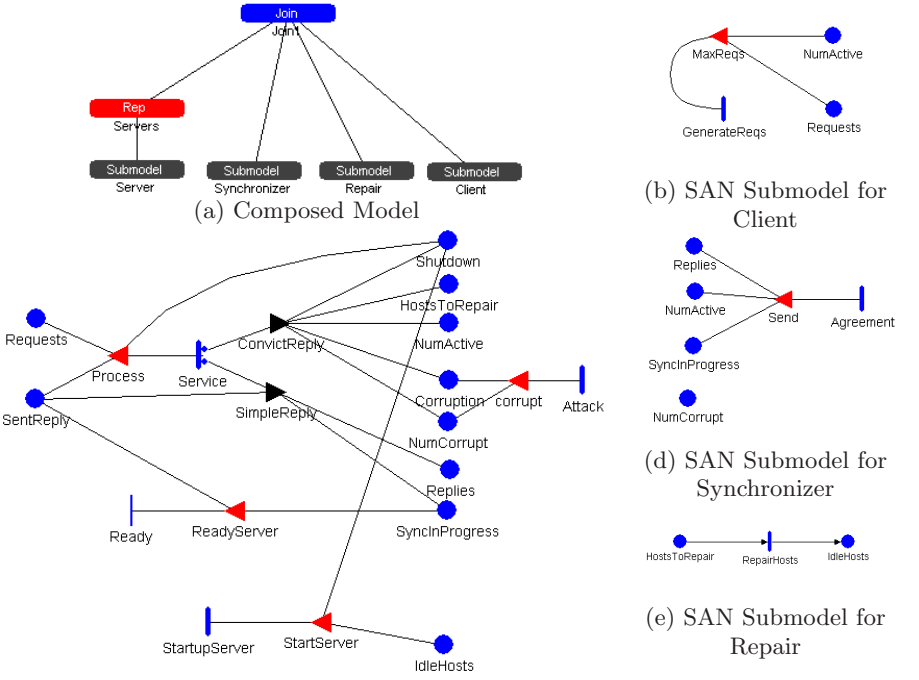
(d) SAN Submodel for Server

(e) SAN Submodel for ConfigManager

Fig. 2. SAN Models for CRCM

of corruption of this server. A marking of 0 implies no corruption at all, and a marking of `MaxPhases` implies complete corruption, which is sufficient to influence the server's behavior. A value in between indicates that some phases of a multi-phase attack have been successful, but that the system is not corrupt enough to behave incorrectly. We model dispersion by having the probability of success of a phase in a multi-phase attack be the reciprocal of the marking of *NumActive*, a shared place that keeps track of the number of servers online. That accurately models the fact that each phase randomly goes to any of the active servers. The probability of successful detection is proportional to the number of changes that have been made to the configuration of the server (represented by the number of successful attack phases, which is equal to the marking of *Corruption*). Because of model size and complexity, we do not model false alarms. However, that does not constitute a shortcoming of our models, given that our focus in the models is on the *effect* of intrusion reports. Hence, we model a composite of actual attacks and false alarms (or, equivalently, correct and false intrusion reports). Upon successful detection, the Configuration Manager causes the server to be taken offline. The Manager informs the load-balancing gateway about this change, and the latter no longer forwards new requests to the server. The activity *Repair* represents the process of reinitializing the state of the server, after which the server can receive requests again.

Multicast Routing Centralized Management (MRCM). The composed model and atomic SAN submodels for MRCM are similar to those for



(a) Composed Model

(b) SAN Submodel for Client

(c) SAN Submodel for Server

(d) SAN Submodel for Synchronizer

(e) SAN Submodel for Repair

Fig. 3. SAN Models for SMR

CRCM. Here we point out the major differences. Since a firewall is now present on each host running the server, the **FirewallGw** and **Server** submodels are joined to form a model of each host. The resulting submodel is replicated `NumServers` times to form a model of the set of servers. Requests for each server are generated separately; there is no centralized request generation as in CRCM. This is done to model each request going to all servers, and exactly one of them picking it up for service, while others discard it. We model the redistribution of requests when a server goes offline by setting the rate of *FilterRequests* to be weighted by the fraction of the total number of servers that are currently active. Since there is no dispersion in this architecture, if the case corresponding to multi-phase attack is chosen in *ServeReq*, the phase is always successful, resulting in an increase in the marking of *Corruption*.

State Machine Replication (SMR). The composed model for SMR (Figure 3(a)) consists of four atomic SAN submodels: **Client**, **Server**, **Synchronizer**, and **Repair**. The **Client** SAN in Figure 3(b) models the centralized generation of incoming requests to the system, since each request is sent to all the active servers. The **Server** SAN (Figure 3(c)) models the processing of client requests by a server, attacks on a server, performance of Byzantine agreement

between servers before a reply is sent back to the client, exhibition of incorrect behavior by corrupt servers, the subsequent exclusion of corrupt servers from the server group (provided there are enough uncorrupted servers for agreement), restarting of new servers on standby hosts, and repair of excluded hosts. Since the system reacts identically to single-phase and multi-phase attacks (since each request is sent to all servers), we have modeled both by a single activity. Also, since each server has a publicly visible IP address and there is no firewall, we have modeled the attack generation explicitly, instead of having it be a part of the request stream. On firing, the marking of the local place *Corruption* is set to 1, and the marking of the shared place *NumCorrupt* is incremented. The activity *Service* represents the processing of a client request by the server, and the reaching of Byzantine agreement among the servers on the reply. If the marking of *Corruption* is 1, the probability of the case corresponding to the output gate *ConvictReply* is `probMisbehavior`, a global variable that represents the probability that a corrupt replica will exhibit corrupt behavior during the agreement process. Upon misbehavior, the server is taken offline, and the marking of the shared place *HostsToRepair* is incremented, since the host on which the server was running is also excluded, and we need to repair this host and bring it back into the system. If the marking of *Corruption* is 0, the case corresponding to the output gate *SimpleReply* is chosen with a probability of 1. The activity *StartupServer* represents the starting of a new server on a standby host, to replace one that has been shut down. We include standby hosts for SMR, because Byzantine agreement among hosts is the only way of detecting corruption, and it is necessary to have the corrupt server replaced quickly (by a server running on a standby host) to maintain the same level of intrusion tolerance. The SAN representation of the **Synchronizer** submodel (Figure 3(d)) models the completion of the response to a client request. It is needed since the servers have to maintain the same state. The SAN in Figure 3(e) models the repair process of the excluded hosts, which results in their transition to the standby state.

Multicast Routing Decentralized Management (MRDM). The composed model and atomic SAN submodels for MRDM are similar to those for MRCM. The major differences are as follows. The composed model for MRDM does not have a **ConfigManager** submodel, since the management decision is taken in a decentralized manner using Byzantine agreement; in the **Server** submodel for MRDM, upon detection, a corrupt server is taken offline only if the other servers can reach a Byzantine agreement on shutting it down. Since multi-phase attacks are dispersed in MRDM, the probability of success of an attack phase in the **Server** submodel varies inversely with the number of active servers.

4 Results

We used the Möbius [4] tool to build the SANs, define performance and intrusion tolerance measures, design studies on the models, simulate the models, and

obtain values for the measures defined on various studies. The measures defined on each model for use in the studies are as follows:

Productive Throughput: This measure characterizes the number of requests that the system replies to correctly per time unit. We assume that all correct servers reply correctly to the requests they receive, and all corrupt servers reply incorrectly to the requests they receive. We study the expected value of this measure averaged over a time interval.

Unproductive Throughput: This measure characterizes the number of requests that the system replied to incorrectly per time unit.

Strong Unavailability for an interval: This measure characterizes the fraction of time the service was *improper* in the given time interval. For this measure, the service was defined to be improper (for the CRCM, MRCM, and MRDM architectures) if at least one active server was in a corrupt, undetected state, or all servers were offline for repair. For SMR, the service is improper if more than a third of the active servers are corrupt. Hence, a strongly available system does not send an incorrect reply to any request.

Weak Unavailability for an interval: Here, we use a weaker definition of proper service. The service is proper if at least one correct server is online. This measure is not defined on models for SMR. The above two unavailability measures characterize the survivability of the systems as perceived by a user.

Fraction of Corrupt Servers: This measure characterizes the fraction of active servers that are corrupt at a given instant of time.

We designed several studies on the models to determine how various architectures behave when we vary some important system parameters, and to determine the range of parameter values for which a particular architecture is superior over others, with respect to intrusion tolerance and performance characteristics. The input parameters we varied are the number of hosts in the system, the rate of single-phase attacks on the system, the rate of multi-phase attacks on the system, the quality of the detection mechanism being used, and the rate at which components taken offline are repaired and brought back into the system.

Unless otherwise specified, we used the values given below for various input parameters. We need to emphasize here that the reader need not be particularly concerned about our specific choice of parameter values, because the aim of these experiments is to present performance and dependability *trends/patterns* of these architectures relative to each other, rather than exact values. It is very hard (if not impossible) to come up with any single universally applicable choice of values, because these architectures could be deployed in widely varying situations. However, using our SAN models, we can quite easily conduct these experiments for a large range of parameter values.

We consider a time unit of one minute. Request arrival rate was set to 100 requests (to the entire service system) per minute for all the architectures. Cumulative attack rates were set to be 12 and 6 per hour for single and multi-phase attacks respectively.

The local detection components running on each server check for corruption once every two minutes for CRCM, and once every minute for MRCM and

MRDM. That is justified because CRCM uses a centralized detection mechanism with lightweight daemons running on individual hosts, resulting in slower detection, whereas all the detection in MRCM and MRDM is done locally on each host, resulting in faster detection. The probability of detecting a corruption in each run is set to 0.5. Likewise, in SMR, a corrupt server misbehaves with a probability of 0.5. (In Section 4.1, we explain why the probability of misbehavior in SMR is equivalent to the probability of detection in other architectures.)

The probability that the centralized firewall in CRCM will detect and filter out an attack in CRCM was set to 0.75. The probability that the local firewalls on each host running a service component in MRCM and MRDM will detect and filter out an attack was set to 0.4. We use a higher probability for CRCM since it has a centralized firewall running on a dedicated machine that can detect and filter out attacks more intelligently. However, we realize that the exact degree of difference in a real setting will vary depending on the strength of firewalls actually deployed.

The mean time to repair an offline server was set to 17 minutes in all the architectures.

The total number of hosts was set to 12. So that all architectures would have similar amounts of resources, that number includes the hosts running service components as well as the hosts running trusted components. Hence, CRCM had 10 hosts running service components and 2 hosts running trusted components (the Configuration Manager and Gateway); MRCM had 11 hosts running service components and one host running a trusted component (the Configuration Manager); and SMR and MRDM each had all 12 hosts running service components. SMR had 3 additional hosts in the standby state.

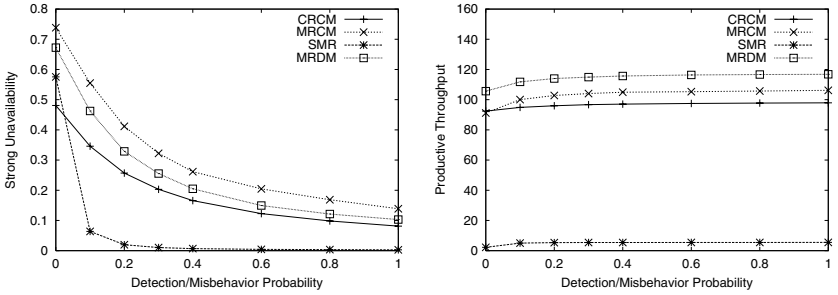
The time interval considered is [0, 30 minutes]. The fraction of corrupt servers is measured at the end of this interval.

We used simulation to solve all the models; all results presented here have a 95% confidence interval.

4.1 Comparison under Varying Quality of Detection

For the CRCM, MRCM, and MRDM architectures, the *quality of detection* is the probability with which an intrusion detection system can ascertain that a system has been compromised, given that the system is actually corrupt. SMR does not have a separate intrusion detection system, and detects intrusion primarily through Byzantine agreement by the group; the group members can know a corrupted member is corrupted only when it shows some misbehavior during the agreement, by deviating from the protocol specification. That is modeled by the probability of misbehavior. We varied the detection probability from 0.0 (no intrusion detection) to 1.0 (perfect intrusion detection). For SMR, the probability of misbehavior was varied from 0.0 (corrupt server does not misbehave at all) to 1.0 (corrupt server always misbehaves).

Figure 4(a) shows that in the absence of an intrusion detection mechanism (or equivalently, absence of misbehavior in SMR), the strong unavailability of any



(a) Strong Unavailability (b) Productive Throughput

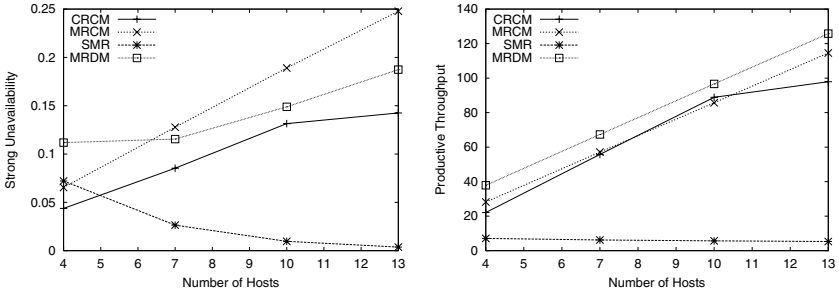
Fig. 4. Varying Detection/Misbehavior Probability

architecture depends primarily on the architecture's defense against intrusion attempts. Thus, CRCM shows the best performance and the least unavailability, because it has a strong firewall and better handling of multi-phase attacks. All the other architectures suffer because of weaker firewalls; MRCM performs the worst because it is most susceptible to multi-phase attacks, due to lack of dispersion. When the probability of detection increases, all architectures become more available, but among CRCM, MRCM, and MRDM, the CRCM architecture remains the best and MRCM the worst for the same reasons. We notice that SMR is initially very sensitive to any increase in the probability of misbehavior, because as long as the Byzantine agreement requirement is met, any corrupt misbehaving servers can be immediately eliminated. However, for large values of the misbehavior probability, it becomes increasingly difficult for more than one-third of the group to be corrupt at any one time (which is the criterion for unavailability in SMR).

Figure 4(b) shows that SMR has the least amount of productive throughput, because all servers process every request. Its throughput does not change for misbehavior probabilities greater than 0.3, because above that value it is almost always available. A trend that is observed in all architectures is that beyond a certain detection probability (approximately 0.3 for the input parameter values used in this study), throughput does not show an appreciable increase. The reason is that throughput depends primarily on the system's total service capacity (given by the service rate) and the arrival rate, and these parameters were kept constant in our studies. Among the CRCM, MRCM, and MRDM architectures, the differences in productive throughput is due to the fact that MRDM has two more servers than CRCM and one more server than MRCM.

4.2 Comparison under Varying Numbers of Hosts in the System

Varying the number of hosts in the system from 4 to 13 implies that the number of hosts serving requests (servers) varies from 2 to 11 in CRCM, from 3 to 12 in MRCM, and from 4 to 13 in SMR and MRDM. For 4 hosts, SMR and MRDM

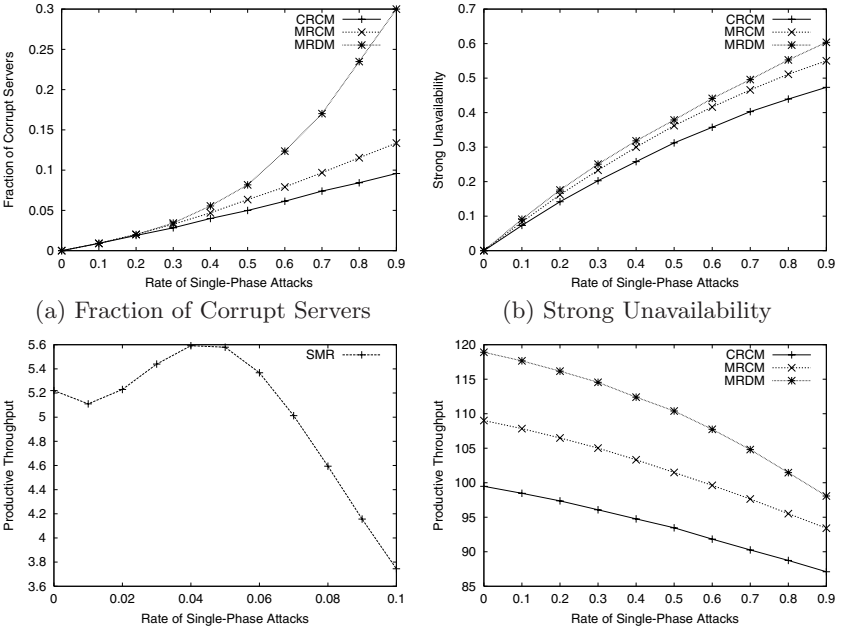


(a) Strong Unavailability (b) Productive Throughput

Fig. 5. Varying Number of Hosts

are more unavailable than CRCM and MRCM (see Figure 5(a)), because they require Byzantine agreement in order to exclude corrupt servers, and 4 servers can tolerate at most one corruption. Given enough time, it may be easy to corrupt one server, and beyond that point, no further corruptions can be tolerated, hence affecting availability. Also, MRDM performs worse than SMR, because MRDM is considered unavailable in the strong sense even when one server is corrupt, while SMR is considered available until one-third of the servers are corrupt. SMR shows decreasing unavailability with an increasing number of hosts, because larger group size enables it to tolerate a larger number of simultaneous faults. However, unavailability for CRCM and MRCM increases with the number of hosts; that may seem counter intuitive, but the greater number of hosts means that there is a greater chance that one host will be corrupt and online. Like SMR, a larger number of servers makes it easier for MRDM to detect corrupt servers and exclude them. On the other hand, a larger number of servers makes it more likely that MRDM will have a corrupt server online. Because of these opposing forces, MRDM's unavailability initially remains unchanged, and starts increasing later, because the negative effect of having more servers becomes more dominant. We also note that CRCM does not show an appreciable increase in unavailability above 10 hosts. The reason is that for the chosen arrival rate and service rate of the individual servers, the waiting time for any request (and hence any attack) is negligible for 10 hosts, and is unaffected by a further increase in the number of hosts.

In SMR, all hosts process every request, so increasing the number of hosts does not help in increasing throughput; rather, productive throughput (Figure 5(b)) actually falls a little, because of an increase in agreement delays. MRCM and MRDM show steady increase in productive throughput, which is to be expected from parallel processing architectures. On the other hand, CRCM does not show an appreciable increase in productive throughput when the number of hosts goes beyond 10, because at that point the central dispatcher starts acting as a bottleneck in the system, as mentioned before.



(a) Fraction of Corrupt Servers

(b) Strong Unavailability

(c) Productive Throughput for SMR

(d) Productive Throughput for CRCM, MRCM, and MRDM

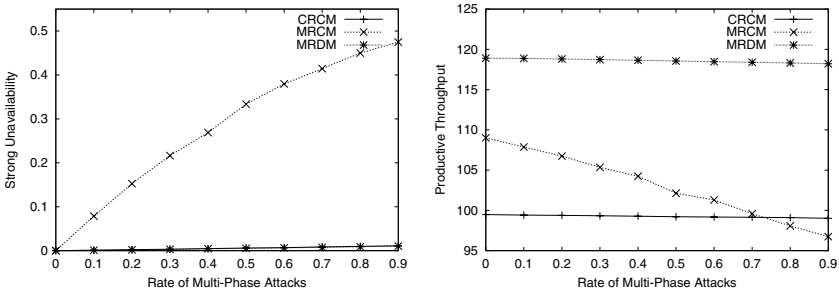
Fig. 6. Variation in Measures with Varying Single-phase Attack Probability

4.3 Comparison under Varying Single-Phase Attack Rates

In this study, we varied the probability that an incoming request is a single-phase attack from 0 to 0.009 (in increments of 0.001) for the CRCM, MRCM, and MRDM architectures. That resulted in the single-phase attack rate varying from 0 to 0.9 (in increments of 0.1), since the request arrival rate is 100. The probability of multi-phase attacks was set to 0. For SMR, the attack rate was varied along the same lines.

Figure 6(a) shows the variation in the fraction of active servers that are corrupt for the CRCM, MRCM, and MRDM architectures. We observe that CRCM performs better than the other two architectures. That can be attributed to CRCM's stronger centralized firewall as compared to the weaker local firewalls in MRCM and MRDM. Since dispersion of multi-phase attacks is not a factor in this study, MRCM performs comparably. The linear increase for CRCM and MRCM is as expected, but there is a rapid deterioration for MRDM. The reason is that in MRDM, for higher attack rates, there is a significant probability that more than a third of the servers will become corrupt before any detection, thus violating the Byzantine agreement requirement, and hence making it impossible for any corrupt server to be removed from the set of active servers.

Figure 6(b) shows the variation in strong unavailability for the CRCM, MRCM, and MRDM architectures. All the architectures perform similarly and



(a) Strong Unavailability (b) Productive Throughput

Fig. 7. Variation in Measures with Varying Multi-phase Attack Probability

are strongly affected by the rate of attacks. CRCM is slightly better due to its strong centralized firewall, and MRDM is slightly worse due to the failure of the Byzantine agreement algorithm for higher attack rates.

Figure 6(c) depicts the variation in productive throughput for the SMR architecture. The performance overhead due to the Byzantine agreement protocol increases with the number of servers in the system. However, instead of increasing linearly, it increases as a step function, with almost fixed-size jumps whenever the number of servers is of the form $3f + 1$ (i.e., jumps at 4, 7, 10, and so on). This has been shown experimentally in [10]. Since the throughput varies inversely with the delay, the gain in throughput with decrease in the number of servers is more substantial when the number of servers is smaller. Increasing the attack rate decreases the number of servers; this decreases the Byzantine agreement overhead, and hence tends to increase the throughput. On the other hand, the probability of enough servers becoming corrupted to violate the Byzantine agreement requirement increases with increasing attack rates, hence decreasing productive throughput. The nature of this graph can be attributed to the competition between these two opposing forces. The former dominates the initial portion of the graph, while the latter dominates when the attack rate is higher. As explained above, the gain in throughput is not much when the expected number of servers online is high, and that leads to the domination of the latter force for very low attack rates, resulting in the initial dip in the graph.

Figure 6(d) shows that productive throughput decreases with increasing attack rates, as fewer correct servers are online. The relative performance of the architectures can be explained by the facts that CRCM has a performance bottleneck of centralized request routing, and that MRDM, MRCM, and CRCM have 12, 11, and 10 servers working in parallel, respectively.

4.4 Comparison under Varying Multi-phase Attack Rates

In this study, we vary the probability that a particular request is part of a multi-phase attack from 0 to 0.009, while keeping the number of single-phase

attacks at 0. Figure 7(a) shows that CRCM and MRDM (coinciding lines) perform better than MRCM with respect to strong unavailability. The reason is that multi-phase attacks in CRCM and MRDM are largely unsuccessful due to dispersion, and have a negligible effect on strong unavailability. The effect on MRCM becomes more evident when we look at the productive throughput for the three architectures in Figure 7(b). Though MRCM starts out better than CRCM because of one additional server, its performance degrades rapidly as we increase the probability of multi-phase attacks.

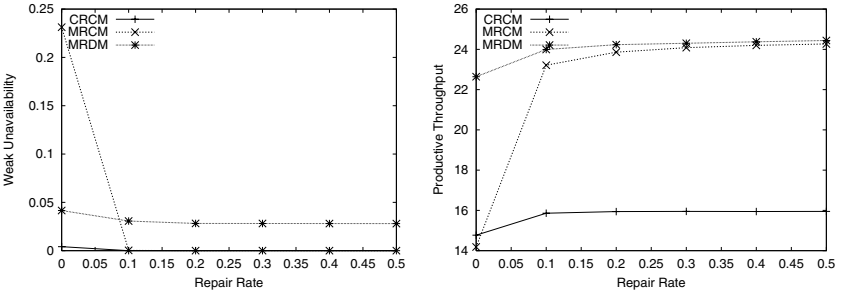
4.5 Comparison under Varying Repair Rates

When a corrupted server is detected, it is removed from the set of active servers, taken offline, and put into repair. After repair, the server is put back into the pool of active servers. The system would eventually fail if there was no repair or the repair was not "fast enough," i.e., if the mean time between successful attacks is shorter than the average time taken to repair a server and put it back into service. Thus, we can intuitively predict that a faster repair rate is crucial for ensuring that the system provides continuous service.

Figure 8 confirms this intuition. In obtaining the data for these graphs, we considered, for all architectures, a set of 4 hosts running the service component. Additional hosts were used for the trusted management components (Gateway, Configuration Manager, and Firewall) if those components are required in the architecture. We varied the repair rate from 0 (no repair) to 0.5 (very fast repair rate: one repair every 2 minutes), while the other parameters were kept constant. The attack rate was kept constant at 0.08 per time unit. As the repair rate varies from 0 upwards, we can see that productive throughput increases until a saturation point. The saturation point is reached when the repair rate is faster than the attack rate. Increasing the repair rate beyond that point has some beneficial effects, but not substantial improvements. A similar trend can be observed from the graphs depicting weak unavailability. The saturation point (for a given estimate of the attack rate) represents the optimal repair rate; it is "optimal" in the sense of getting maximum benefit from minimal cost for repair.

From Figure 8(a), we can see that with no repair, CRCM performs the best, because of its strong firewall and its use of a dispersion mechanism. MRDM and MRCM do not have a strong firewall, but MRDM outperforms MRCM due to dispersion in the former. Since CRCM starts out with low unavailability, it is not affected substantially by an increase in repair rate. MRCM matches the low unavailability of the CRCM architecture after the optimal repair rate has been reached. The MRDM architecture, on the other hand, is not able to attain such low unavailability, even after the saturation point. The reason is that our experiments were conducted with 4 servers, and when the number of correct servers drops to 3, it is not possible to reach Byzantine agreement to remove the next corrupted server from the set of active servers.

Though the CRCM and MRCM architectures outperform the MRDM architecture in availability, with respect to correctness of replies (productive through-



(a) Weak Unavailability (b) Throughput up to $t = 120$ min

Fig. 8. Variation in Measures with Varying Repair Rates

put), MRDM is clearly superior (as seen from Figure 8(b)). The duration between detection of an intrusion and removal of the corrupted server from the active set is shorter for MRDM than for the CRCM and MRCM architectures, due to the fact that it does not have the bottleneck of a centralized manager. Therefore, the number of potentially erroneous replies that a corrupted server could send before being removed would be less for the MRDM architecture than for other architectures. However, we expect that for a greater number of servers, this advantage may become less important for MRDM, because the overhead due to the Byzantine agreement protocol increases significantly as the number of servers increases, as shown experimentally in [10].

5 Conclusion

This work is the first attempt to evaluate intrusion-tolerant server architectures. We define a series of relevant metrics and present a probabilistic evaluation and comparison of four representative intrusion-tolerant server architectures. The results present useful information about the intrusion tolerance and performance characteristics of the architectures, by means of varying system parameters such as the quality of intrusion detection, rate of attacks on the system, amount of resources, and time to repair an intruded server.

The results show that architectures that use a small number of trusted components to secure a large set of servers have better availability than architectures with no trusted components when the level of redundancy in the system is not very large. However, [9] shows that it is difficult, if not impossible, to implement truly trustworthy components. Such architectures also usually employ centralized decision-making, which is a potential performance bottleneck.

State-machine-replication-based architectures that employ Byzantine fault-tolerant protocols for agreement on the request processing have the best intrusion tolerance characteristics, but they have comparatively lower performance. Hence, such architectures are a good choice for implementing mission-critical systems for which the ability to withstand intrusions is more important than performance.

Architectures that employ decentralized decision-making and serve multiple requests in parallel have the best performance for a given amount of resources, since all the resources can be used for request processing. They are superior to centralized architectures, for which a portion of resources need to be set aside for hosting trusted components. However, from an intrusion tolerance perspective, the effectiveness of such decentralized architectures is realized only when there is a sufficient degree of redundancy. We also observe that introducing unpredictability in request routing (dispersion) is highly effective in defense against multi-phase attacks, and that it is critical that the mean time to repair be much less than the mean time between attacks.

We believe that our choice of values for model parameters is reasonable, but more importantly, our models allow system designers to evaluate alternative architectures by assigning different values for those parameters as they deem appropriate. This certainly enhances their ability to make more informed choices between various intrusion-tolerant architectures easily and quickly, before undergoing the expensive process of building and evaluating multiple prototypes.

Acknowledgments. We thank Dr. Marinho Barcellos for his help in improving the manuscript, and Jenny Applequist for her editorial assistance.

References

1. V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic Load Balancing on Web-server Systems," *IEEE Internet Computing*, Vol. 3, No. 3, pp. 28–39, 1999
2. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Third Symp. on Operating Sys. Design and Implementation (OSDI '99)*, pp. 173–186, 1999
3. M. Cukier, J. Lyons, P. Pandey, H. V. Ramasamy, W. H. Sanders, P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, M. Atighetchi, and J. Gossett, "Intrusion Tolerance Approaches in ITUA," FastAbstract in *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pp. B64–B65, 2001
4. D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius Framework and Its Implementation," *IEEE Trans. on Software Engineering*, Vol. 28, No. 10, pp. 956–969, October 2002
5. A. Delis and N. Roussopoulos, "Performance and Scalability of Client-Server Database Architectures," *Proc. Intl Conf. in Very Large Data Bases (VLDB)*, pp. 610–623, 1992
6. Y. Deswarte, L. Blain, J. C. Fabre, "Intrusion Tolerance in Distributed Computing Systems," *Proc. IEEE Symposium on Security and Privacy*, pp. 110–121, 1991
7. Draper Laboratories, Inc., "Kinetic Application of Redundancy to Mitigate Attacks," *DARPA OASIS Program*, http://www.tolerantsystems.org/ProjectSummaries/IT_Using_Masking_Redundancy_and_Dispersion.html
8. V. Gupta, V. Lam, H. V. Ramasamy, W. H. Sanders, and S. Singh, "Dependability and Performance Evaluation of Intrusion-Tolerant Server Architectures," *CRHC Technical Report*, 2003, to appear
9. U. Lindqvist, T. Olovsson, and E. Jonsson, "An Analysis of a Secure System Based on Trusted Components," *Proc. Eleventh Annual Conf. on Computer Assurance (COMPASS '96)*, pp. 213–223, Gaithersburg, Maryland, 1996

10. H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems," *Proc. Intl Conf. on Dependable Sys. and Networks (DSN-2002)*, pp. 229–238, 2002
11. W. H. Sanders, M. Cukier, F. Webber, P. Pal, and R. Watro, "Probabilistic Validation of Intrusion Tolerance," FastAbstract in *Supplemental Volume of the 2002 International Conference on Dependable Systems and Networks*, pp. B78–B79, 2002
12. W. H. Sanders, and J. F. Meyer, "Stochastic Activity Networks: Formal Definitions and Concepts," In *Lectures on Formal Methods and Performance Analysis*, LNCS 2090, Springer-Verlag (E. Brinksma, H. Hermanns, J.P. Katoen, Ed.), Berlin, pp. 315–343, 2001
13. F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, Vol. 22, No. 4, pp. 299–319, 1990
14. Secure Computing Corporation, "Intrusion Tolerant Server Infrastructure," *DARPA OASIS Program*, http://www.tolerantsystems.org/ProjectSummaries/Intrusion_Tolerant_Server_Infrastructure.html
15. S. Singh, M. Cukier, and W. H. Sanders, "Probabilistic Validation of an Intrusion-Tolerant Replication System," *Proc. Intl Conf. on Dependable Sys. and Networking (DSN-2003)*, pp. 615–624, 2003

Building Trust Chains between CORBA Objects

Emerson Ribeiro de Mello[1], Joni da Silva Fraga[1], Altair Olivo Santin[1], and Frank Siqueira[2]

[1] Department of Automation and Systems
{emerson,santin,fraga}@das.ufsc.br,
http://www.das.ufsc.br
[2] Department of Computer Science and Statistics
Federal University of Santa Catarina
Fax/Phone: +55(48) 331-9934
88040-900 Florianópolis, SC - BRAZIL
frank@inf.ufsc.br

Abstract. This work presents an authentication and authorization model that results from the integration of the SPKI/SDSI infrastructure with CORBAsec. The paper presents the main facilities provided by the proposed model, showing the advantages of using the SPKI/SDSI infrastructure. CORBA provides to the model the advantages of interoperable distributed objects in heterogeneous environments. The idea defended in this paper is the better suitability of trust chains, as the SPKI/SDSI, with the characteristics of the world-wide network.

1 Introduction

The Internet provides a powerful communication infrastructure that has enabled the development of distributed applications with global coverage. However, together with the means provided by the Internet, new challenges appeared in the development of distributed applications, such as the aim for flexibility and scalability. CORBA (Common Object Request Broker Architecture), an architecture for open and distributed systems based on objects, appeared to solve such difficulties. CORBA middleware provides for interoperability and portability in distributed applications, as well as location transparency and platform and programming language heterogeneity.

With the wide use of distributed systems, information protection became a necessity. Distributed applications should avail of security mechanisms that guarantee properties such as integrity, confidentiality, authenticity and availability, so that these applications can operate in a correct and effective way. The CORBAsec (CORBA Security) specification appeared in the form of service objects to assist such security requirements in distributed object-oriented applications based on CORBA middleware. CORBAsec is a model capable of supplying functionality such as identification and authentication of principals, access control of method invocations, among others.

The classic authentication and authorization model employed in distributed systems defines, in a name domain, a centralized authentication process preceding the authorization, which can have its distributed controls. Such a model is suitable for environments in which the client is known beforehand; however, this model has shown to be inadequate for Internet applications, for example, when the client is likely to be unknown.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 102–116, 2003.

Developed in order to facilitate the conception of scalable and safe computing systems, SPKI/SDSI provides a fine access control, using a local name space and a simple authorization model based on trust chains (egalitarian model). The present work shows how to integrate the CORBA security model with the SPKI/SDSI infrastructure, providing a decentralized control for authentication and authorization policies with fine granularity. This paper is organized as follows. Section 2 describes the SPKI/SDSI public key infrastructure. Section 3 introduces the CORBA security service. In section 4, the proposed authorization model is presented, integrating the two above-mentioned technologies. Section 5 describes the implementation of the proposed model. In Section 6, related proposals found in the literature are described and compared to this proposal; and section 7 presents our concluding remarks.

2 SPKI/SDSI

The creation of both SPKI (Simple Public Key Infrastructure) and SDSI (Simple Distributed Security Infrastructure) was motivated by the limitations and by the complexity of public key infrastructures based on global name hierarchies, such as X.509. SPKI/SDSI is the union of these two formerly independent proposals. Created at the MIT by Ronald Rivest and Butler Lampson, SDSI [1] is a public key infrastructure (PKI) that has as its main characteristic the use of a local name space. Proposed by Carl Ellison and others, SPKI [2] was designed with the intent of being a simple and flexible authorization model, very well defined and of easy implementation. The union of SPKI and SDSI resulted in an authentication and authorization system for distributed applications.

2.1 SPKI/SDSI Certificates

In SPKI, a principal[1] is represented by a public key, and not by an individual name[2]. This represents a great advantage comparing to the approach based on global name hierarchies, because a public key is more stable than a name in large-scale systems (i.e. two people can have one same name).

Each SPKI principal holds a pair of keys that allows him to create, sign and publish certificates just like a X.509 certificate authority (CA). Version 2.0 of SPKI/SDSI defines two types of certificates: name certificate and authorization certificate. In SPKI/ SDSI, for each public key there exists an associate local name space. A local name is a pair composed of a public key and an arbitrary identifier. The name certificate (see Table 1) is responsible for the publication of local names, allowing other principals to get the public key associated to the locally-defined name.

A name certificate can associate a name to the one public key, to other names inside the local name space of the issuer, or to other name certificates in the name space of other

[1] The term 'principal' usually refers to the individual that is represented by the public key, but it can reference entities representing this individual, such as an institution, a machine, or a process. The principal is always recognized by the security polices of the system as the authorized entity.

[2] Known as keyholder in SPKI.

Table 1. SPKI/SDSI Name Certificate

Issuer	Public key of the certificate issuer, whose signature should follow the certificate.
Name	Arbitrary local name which identifies the subject.
Subject	Public key or name composed of a public key followed by one or more identifiers.
Validity	Period during which the certificate is considered valid.

principals, forming certificate chains. Names propagation in SPKI is possible through trust chains, formed by certificates of linked names.

The authorization certificate (see Table 2) grants a specific authorization, given by the certificate issuer, to the subject of the certificate. The authorization granted by the issuer to the subject can be delegated by this subject, integrally or partially, for other principals if the delegation bit is active. When allowing the delegation of the certificate, the issuer assumes that the subject is trustworthy, because he will be capable of delegating rights to any principal that he wants without consulting the issuer.

Table 2. SPKI/SDSI Authorization Certificate

Issuer	Public key of the certificate issuer, whose signature should follow the certificate.
Subject	Public key or name composed of a public key followed by one or more identifiers.
Tag	Specification of the authorization that will be granted by the issuer to the subject.
Delegation bit	Binary field that indicates if the certificate can be delegated.
Validity	Period during which the certificate is considered valid.

In this way, the trust model allows the construction of authorization chains that begin with the key of the service guard and ends with keys of principals that want to access the service. A principal that wants to delegate a received right must attach the received certificate to a sequence of certificates and send this sequence to the subject, which then has the access right granted.

SPKI is a key-oriented infrastructure and as a consequence, for controlling access to services, it is necessary to resolve all the names that compose the chain in order to obtain the public key of the principal. The chain is considered valid if the first key of the chain is the public key of the resource holder and if the last key is the public key of the principal that wants to access the resource.

2.2 Access Control List (ACL)

A SPKI/SDSI ACL is composed of several entries, similarly to an authorization certificate (see Table 2), with the main difference being the lack of a certificate issuer; therefore,

an ACL is not signed. An entry in an ACL is said to be a delegation of rights given by the issuer, the owner of the service, to the subject [3]. The validity period of the entry is specified by the validity field, and in case this field is not specified the validity period is assumed to be unlimited.

3 CORBA Security Service (CORBAsec)

The CORBAsec specification [4] was designed to enforce security requirements in distributed systems. The CORBA security model is specified in the form of an object service (COSS), and provides secure exchange of information for distributed object applications.

The CORBA security specification defines a model capable of supplying mechanisms for identification and authentication of principals, controlling access to services through invocating remote methods, secure communication (including data encryption to guarantee the confidentiality as well as the integrity of messages), non-repudiation[3], auditing and administration.

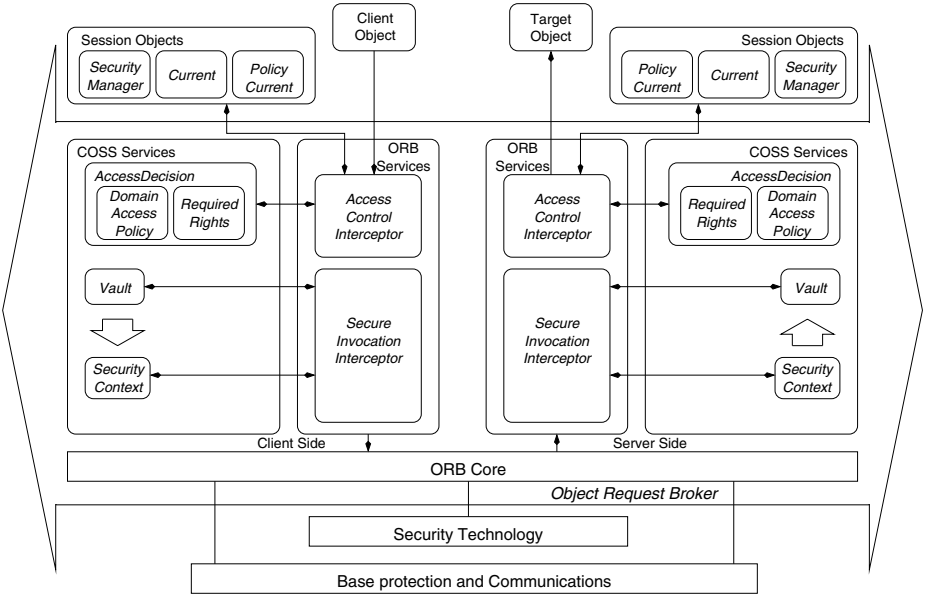


Fig. 1. CORBAsec Structure [5]

In the CORBA security model, objects are located at four levels (Figure 1): application level, which understands the application objects; middleware level, composed by service objects, ORB services and ORB core; security technology level, formed by the underlying security services; and the basic protection level, composed of hardware platforms and local operating systems.

[3] This is optional and it is available only for CORBAsec level 2 (security-aware applications).

Notice that CORBAsec in itself does not provide any security mechanism, such as data encryption. Instead, CORBAsec provides only a standardized architecture and interfaces that can be used to provide security in a distributed object environment. Security mechanisms, such as Kerberos, SPKM, SESAME and SSL, must be provided separately. The study of these mechanisms is not the subject of this paper; a comprehensive survey on security mechanisms is found at [6].

Security in CORBAsec is provided in two levels. Security level 1, also known as ORB level security, supplies security mechanisms for security-unaware applications, guaranteeing access control and event auditing through the ORB. Security level 2, also known as application-level security, has other application and management interfaces besides supporting level 1 functionality. Level 2 is used by security-aware applications, allowing these to customize the way security is implemented.

A principal should first establish its rights before accessing remote objects. The PrincipalAuthenticator object implements the authentication service, returning credentials for the context of the principal. CORBAsec Level 2 allows the principal to change privileges in his credentials, and also allows him to choose which credential will be used for the invocation.

Access control at ORB level is accomplished in a transparent way for applications. During an invocation, or at binding time, the access control interceptor contacts the AccessDecision object that, through the access_allowed method, determines if the invocation should be allowed or not. In order to do so, the rights supplied by the client through the Credentials object are confronted with the rights required to execute the invocation (object RequiredRights).

Besides the access control interceptors that act during an invocation, CORBAsec also employs a secure invocation interceptor, which intercepts calls at a lower level in the sense of supplying integrity and confidentiality in messages transfers necessary to execute the invocation.

The way in which access control is accomplished in CORBAsec imposes to the model some granularity restrictions. The access control consists basically of the confrontation of rights supplied by the client (DomainAccessPolicy object) with the required rights (RequiredRights object).

Such fact happens because the supplied rights are related to the domain and the requested rights are related to methods, causing a granularity problem. Some solutions for such problem are proposed in [7].

4 SPKI/SDSI and CORBAsec Integration Model

In this section are described the mechanisms that supply the SPKI authorization and authentication controls in CORBAsec. In this work, SPKI/SDSI is extended with the use of Federations [8], which have the purpose of grouping principals with common interests, so that they can share SPKI/SDSI certificates.

4.1 SPKI/SDSI Federations

In [[8],[9]] a trust model that seeks the grouping of principals that possess similar interests was proposed. The main goal of SPKI/SDSI Federations is to provide means for

certificate reduction and for the construction of new chains based on sharing certificates among their members.

Federations are composed of three entities: clients, servers and the certificate manager of the federation. The role of the certificate manager is to facilitate the interaction between clients and servers. The manager provides storage and recovery mechanisms, as well as mechanisms for creation of chains of authorization certificates, necessary so that clients can access servers without the need for an intermediate key.

A principal, whether he is a client or a server, when entering in a federation, supplies the name and authorization certificates (belonging to him and that can be delegated) that he considers useful for the federation. In this way, the repository kept by the certificate manager of the federation will contain all the delegable certificates of federation members. Whenever a principal wants to obtain access to a certain resource but does not have the required rights, it can ask the certificate managers of federations to which he is associated to locate members with delegable certificates that allow access to the required resource. Once found a principal with the required rights, a negotiation among principals – the one that wants to obtain rights and the other who can delegate them – is started, intending to have the required rights delegated to the first. The negotiation can comprise from a simple transfer of rights to a more complex process, such as a financial transaction; the nature of the negotiation process is directly related to the semantics of the application.

A principal can join as many federations as he is accepted. In order to join a federation, the principal may be required to supply a threshold certificate signed by k-of-n members that already belong to the federation, along with a name certificate with which he is requesting the admission. The purpose of joining several federations is to obtain access to a larger amount of resources. By having access to the delegable authorization certificates of all members of each federation joined by him, it becomes easier for a principal to be recognized by other applications spread through the worldwide network.

However, in a large-scale system such as the Internet, it might be necessary for a principal to join several federations in order to be easily recognized. The model avoids this scalability problem by establishing associations among federations, allowing searches and delegations to comprise not only one federation, but a group of associated federations. As a consequence, the amount of federations a principal must join in a large-scale system is reduced sharply.

4.2 The Role of CORBAsec in the Proposed Model

The CORBA architecture and it security service (CORBAsec) were used as the communication infrastructure, guaranteeing the interoperability and the security in the communication in distributed systems.

The proposed authorization model, which is implemented at the application level, employs five CORBAsec objects located at the ORB level: SecurityManager, PrincipalAuthenticator, Credentials, Current and AccessDecision. Other ORB level objects responsible for establishing a secure communication channel – secure invocation interceptor, Vault and SecurityContext – are also used. However, the access control is handled at application level, employing CORBAsec specifications regarding Security Level 2.

The proposed model for integration of SPKI/SDSI with CORBAsec seeks to respect the philosophy of the SPKI/SDSI 2.0 model, where the principal that wants to access a resource is entirely responsible for locating certificate chains that supply it with the required rights to access this resource, leaving to the resource provider the task of verifying the authorization and the authenticity of the request. This verification is executed in the following way: in the attempt to access the resource, the rights supplied together with the request are confronted by the service guard (reference monitor) with the requested rights expressed in an access control list (ACL), determining whether the access to the resource will be allowed for the applicant or not.

The proposed model, therefore, uses CORBAsec Security Level 2 in the sense of maintaining the characteristics of SPKI/SDSI (i.e. keeping security controls at application level). The integration model between SPKI/SDSI and CORBAsec is also a solution for the granularity problems identified in CORBAsec, presented in section 3.

4.3 Dynamics of the Proposed Model

According to the proposed model, a client must be authenticated in order to join the CORBAsec object system. This authentication is accomplished through the PrincipalAuthenticator object, which creates the Credentials object. This object has the privilege attributes of the authenticated client, which will be stored in the SecurityManager session object. All these objects are illustrated by Figure 2.

In order to guarantee the integrity and the confidentiality of messages, SSL version 3 is employed as the underlying security technology. In this way, the authentication process in the CORBA security model is accomplished through the reading of self-signed SSL certificates, which are translated from SPKI/SDSI name certificates according to rules specified in [2]. These SSL certificates are used for mutual authentication and for establishing a secure session (i.e. a cryptographic channel) between a client and a server.

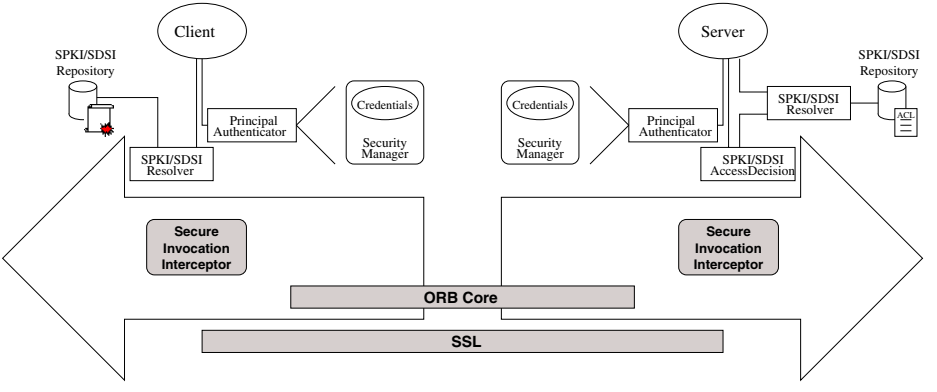


Fig. 2. Overview of the SPKI/SDSI - CORBAsec Integration Model

The ability to work with SPKI/SDSI objects – i.e. to generate certificate chains and to verify the authorization given by a chain – is given to the application by the SPKI/SDSI Resolve object (Figure 2), which encapsulates the SPKI/SDSI 2.0 library.

In the proposed authorization model, the confrontation of the client´s privilege attributes with the control attributes that protect the requested object occurs in a different way, therefore the authentication procedure and the credentials created by this process will not be used to guarantee the access authorization. The access authorization procedure starts with the transfer of SPKI/SDSI authorization certificates from the client to the server using CORBAsec credentials. The server uses the certificates received from the client to activate the service guard, which is responsible for confronting the requested rights with the supplied rights.

On the side of the server, the AccessDecision SPKI/SDSI object (Figure 2) is responsible for deciding if the requests send to the server object will or not to be executed. SPKI/SDSI ACLs play the role of the CORBAsec RequiredRights object, defining the requested rights necessary to invocate each operation supplied by the server, an also the role of the AccessPolicy object, providing groups of principals with rights for executing operations implemented by objects belonging to a domain.

The flexibility offered by CORBAsec level 2 [4] for handling credentials is fully adapted to the use of SPKI/SDSI certificates. Security-aware applications are able to control the security options used during an invocation by choosing the method used for message protection, the privileges contained in their own credentials, the credentials that will be used in the invocation of an object, and if these credentials can only be used by the target object or if they can also be delegated. According to these statements, we have decided to make available in the form of credentials the authorization certificates belonging to the client that are necessary for obtaining of access right to distributed resources. The client's credentials are rebuilt on the server side and can be obtained through the ReceivedCredentials object (Figure 3)

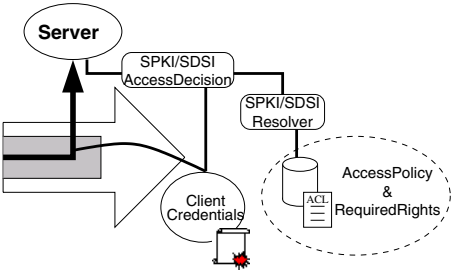


Fig. 3. Objects used for Access Control on the Server Side

4.4 Adding and Recovering Certificates

An object, upon joining the CORBAsec object system, will authenticate itself through the PrincipalAuthenticator object. The authentication is accomplished by presenting its

SSL certificates, since this is the security technology used by the model. However, the credentials created during the authentication process for joining the CORBAsec object system will not be used for access control.

In CORBAsec level 2, the client has the ability to choose which credentials will be used for an invocation. For each authorization chain built by the client in the attempt to accomplish the challenges thrown to him, an object SPKI/SDSI Credentials[4] is created. These authorization chains are converted into sequences of bytes, which will be added as privilege attributes to the credentials defined by CORBAsec.

On the server side, the SPKI/SDSI AccessDecision object is called automatically for each request of a method. This object makes the confrontation of the rights supplied by the client with the necessary rights, specified in an ACL. The client's rights (certificate chain) encapsulated by the SecurityManager session object (Figure 2).

4.5 Access Control List (ACL)

SPKI/SDSI is a practical system that targets the problem of assuring that a user is authorized to execute an action and not just the problem of identifying the user. This characteristic allows larger flexibility in the sharing of resources through delegations, contrasting with authentication systems based on conventional public key infrastructures and authorization through conventional ACLs, which are built starting from the principals´ names.

In conventional public key infrastructures, if a server S wants to verify if principal U really has rights to access the service, it is necessary to consult a Certificate Authority (CA), with which the server has a trust relationship, emitting an authorization certificate that contains the permissions and the name of the principal that is requesting access (U). Besides the server, the only identification of the principal (the client's global name) is contained in the server's ACL.

In SPKI, there is no such entity as a Certificate Authority – instead, each principal is able to issue authorization certificates, so server S can issue an authorization certificate for principal U. Consequently, the use of ACL becomes unnecessary [10], since client U informs which certificates it already has upon accessing the service, and this information is enough for the server to verify the authenticity of the request and if the client has the required rights.

The proposed model uses ACLs in order to allow easier location of valid certificate chains inside SPKI/SDSI Federations. An ACL is composed by several entries, and each entry is composed by a public key and a tag. Tags could be, for example, names of methods provided by a server, allowing him to control access to his methods.

The delegation of authorization certificates is extremely important in the proposed model, since it allows that other entities become authorization issuers. The concept of delegation is used by SPKI/SDSI Federations, since members supply all their authorization certificates that can be delegated and that are important for the federation upon joining it. The federation manager, being a passive entity (i.e. an entity without a public key), does not take part in the trust chain. Instead, the manager acts as a simple repository.

[4] This is a CORBAsec SecurityLevel2 Credentials object, but with specific functionality for controlling access using SPKI/SDSI certificates.

4.6 Mutual Authentication and Authorization in the Proposed Model

One important security principle is to guarantee that protected information will only be revealed to authorized and authentic entities. Cryptographic systems, whether symmetric or based on public keys, provide different means to guarantee these principles. Once guaranteed that the entity involved in the communication is authorized and that the request is authentic (i.e. really originated by the alleged entity), the protected information should be supplied to the requesting entity.

The establishment of secure communication demands that some control messages be exchanged among the communicating parts. The challenge/response protocol, frequently employed by public key infrastructures, is responsible for mutual authentication. This protocol gives assures the client that it is being connected to the right server, and assures the server that it is receiving requests from a valid client.

The communication between client and server in the proposed model uses the challenge/response protocol as a base for the authentication of their respective principals based on SPKI/SDSI authorization certificates and ACLs.

Consider the following scenario: a client wants to invoke a method of a server object. This method is protected by an SPKI/SDSI ACL and, if the request does not cover the necessary requirements, the client must win a challenge in order to have the invocation executed. The steps involved in this protocol between client and server are illustrated by Figure 4 and described below.

In the step 1, the client sends a request to the server together with a nonce[5] (NonceCli) without any authorization chain, because the client ignores the requested rights. Upon receiving the request, the server activates the service guard, which is responsible for confronting the requested rights, contained in the ACL[6] , with the supplied rights. The server generates a challenge, signed by its private key, to the client (step 2). The challenge is composed of the ACL that protects the resource, the nonce sent by the client (NonceCli) and a second nonce (NonceServ) generated by the server.

The client verifies the authenticity of the challenge using the public key of the server[7] and, if the challenge is authentic, activates the necessary devices to generate the certificate chain of that guarantees the access to the resource provided by the server.

Once generated a valid authorization chain, the client sends the response to the server (step 3). The response is composed of the original request, the NonceServ (both signed with the client's private key), followed by the authorization chain.

The server, by holding the client's public key (that can be obtained getting the last key of the certificate chain received from the client), verifies the authenticity of the response. Guaranteed the authenticity, the server activates the service guard in order to verify if the chain really grants the required authorization. In the step 4, the request is granted, returning the wanted information, or refused if the provided chain is not accepted.

It is important to point out that in the protocol described above, the mutual authentication is totally based on SPKI/SDSI authorization certificates. This is a characteristic of the SPKI/SDSI infrastructure.

[5] A random value sent by a server or application requesting user authorization.

[6] Sending the ACL to the client does not imply on security problems, because the ACL entries are composed of public keys, which do not identify their holders.

[7] The server's public keys are known beforehand by the client, possibly through a name certificate.

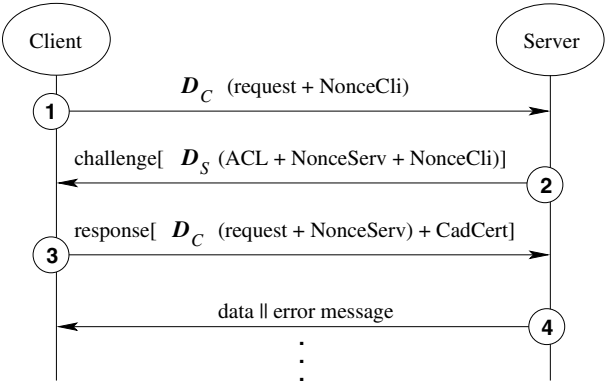


Fig. 4. Access Control Protocol

5 Prototype Implementation

A prototype implementation of the proposed model using SPKI/SDSI certificates is presented and discussed in this section. The architecture of the prototype (Figure 5) employs tools widely used on the Internet, environment in which this work is inserted.

5.1 Architecture

The prototype avails of support for SPKI/SDSI provided by the JSDSI2.0 library, implemented by [11]. Client and server codes are interpreted by a Java Virtual Machine (JVM). The use of Java is currently widespread in distributed applications, mainly because it provides means for the development of platform-independent applications based on Internet application protocols.

CORBA and CORBAsec make possible the interoperability between distributed applications, which are composed of objects running on heterogeneous systems. In the prototype, ORBacus [12], which is an Object Request Broker (ORB) compatible with CORBA version 2.3, was used together with ORBAsec SL2 [13], which despite not being a complete implementation of the CORBAsec level 2 specification, satisfies the needs of the prototype.

The protection of messages sent through the network is guaranteed by TLS/SSL (Transport Layer Security / Secure Sockets Layer). By using this layer, the prototype can guarantee the confidentiality and the integrity of the communication through the network among components running on different machines. The prototype integrates to the ORB the iSaSiLk library [14], which is a fully functional implementation of SSL version 3. Other implementations of SSL, however, can also be used.

5.2 Search for Certificate Chains

The first step of the client when receiving a challenge is to look for a certificate chain linking the public key of the server to his public key in the local repository. In case such

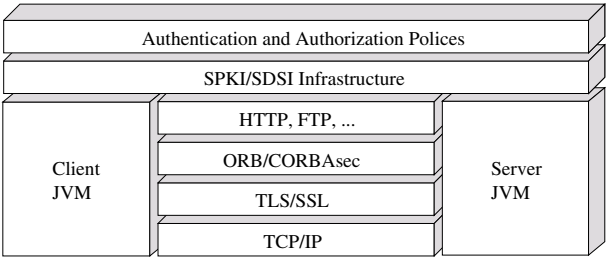


Fig. 5. Prototype Architecture

chain does not exist, the second step is to search for chains linking him to principals listed on the ACL sent by the server.

The search for a chain is performed using the following parameters: PubKeyOfThePrincipalOfTheACL, PubKeyOfTheClient, TAG. At first, the search is performed on the local repository of the client. This search is repeated N times, where N is the number of public keys contained in the ACL.

The third step occurs inside the context of SPKI/SDSI Federations to which the client belongs, whenever the chain of necessary authorization satisfying the challenge does not exist in the local repository. The search is then performed by the certificate managers of each federation. The mechanisms used for searching and generating the certificate chains used by managers are out of the scope of this text; a description of these mechanisms can be found at [8].

5.3 Certificate Repository

SPKI/SDSI certificate repositories can be implemented in several ways. SPKI/SDSI objects are composed of S-expressions [15] that can be stored in ASCII format.

In [16] a standardized way to transform SPKI/SDSI objects represented as S- expressions into XML documents (eXtensible Markup Language) was proposed. Local repositories were built in the prototype based on this proposed, i.e. the local repositories store XML documents converted from SPKI/SDSI objects coded as S-expressions. SPKI/SDSI objects are transformed into XML documents so that they can be stored, but once recovered from the repository they are converted back into S-expressions. This solution allows certificates to be easily exchanged among applications (CORBA objects) involved in the communication and is still compatible with any application that works with SPKI/SDSI objects.

6 Related Work

In [17] is described a complete implementation of the current version of SPKI/SDSI. The implementation was verified building a HTTP server on top of it. The access control protocol adopted defines challenges thrown by the server to the client. Some variations of this protocol have been presented seeking to reduce the amount and the size of messages exchanged between the client and the server.

Clarke also describes a form to protect ACLs against non-authenticated principals, with the protection being done through another ACL. In this way, rights held by each principal on a resource will remain secret for non-authenticated users. Nevertheless, a deeper analysis verifies that is possible that non-authenticated users get the list of principals that have access to the resource, because the key of the principal should be contained in two ACLs. The same is not true for the access rights, because they can be different in each ACL.

The approach described above has the inconvenience of countless message exchanges between the client and the server, because each method request generates two challenges and two responses. The security gain obtained by this approach is not very significant, because SPKI/SDSI public keys already conceal the entities that they represent, imposing to the invader the difficulty of locating the target to be attacked.

In [18] is presented a way of implementing authorization in CORBA distributed applications with SPKI certificates. The resulting implementation is compared with CORBA´s access control, showing the advantages of the proposal. Lampinen employed SPKI version 1.0, which still had not been merged with SDSI – this took place in version 2.0. The client, when joining the CORBA object system, makes available all the authorization certificates belonging to him in a single Credentials object. When receiving an invocation, the server obtains all the authorization certificates belonging to the client and searches for a chain that provides the necessary authorization.

Such approach, besides diverging from the principles used to conceive SPKI/SDSI, also presents scalability problems, because the client can have a large amount of certificates, which is likely to reduce the performance of the application.

In [19] is described a proposal of distributed security model for the Jini technology. The author states that the Jini architecture does not provide any security mechanism in addition to the standard mechanisms provided by Java, such as protection of the client's Java Virtual Machine (JVM) against malicious code executed by proxies.

Eronen and Nikander [19] proposes a solution for the security problems that are not targeted by the Jini architecture, being able to authenticate clients and services and to verify authorization at the level of method calls. However, requires the TAG field of SPKI certificates to be modified, preventing interoperability with SPKI/SDSI certificates provided by other applications.

The model proposed in this paper focused on the current version of SPKI/SDSI. In our proposal the principal, which is the client of the communication, is the only responsible for locating the authorization chain necessary for the access. The server has only the task of verifying whether the chain is valid or not. The proposed model intends to provide a more flexible model for authorization for distributed objects by integrating CORBAsec level 2 with SPKI/SDSI. The granularity problems presented by the former (see section 3) are solved by the integration with SPKI/SDSI.

Access control is implemented in a decentralized way with the necessary granularity. All the characteristics of the current version of SPKI/SDSI are respected, as well as the CORBAsec security level 2 specification. The facilities provided to principals, clients and servers do not impose constraints on their implementations, limiting the effort necessary for using the security service. The authentication protocol guarantees that the entities involved in the communication are authentic and protects the system from replay attacks.

7 Conclusion

This paper presents the proposal of a flexible and scalable authentication and authorization model for large-scale systems such as the Internet. The proposal employs SPKI/SDSI certificates as an alternative for access control in the CORBA security model. CORBAsec is employed as a vehicle for the introduction of the concept of trust chains in order to develop interoperable distributed objects in large-scale distributed systems.

CORBAsec security level 2 provides the necessary facilities for developing this proposal. Such freedom for the implementation of security in applications was extremely important in the model. The use of SPKI/SDSI certificates has shown to be the right choice because it limits the scalability and flexibility problems present in large-scale distributed systems. The implementation of the repositories in XML simplifies the search and generation of certificate chains.

This work is part of a larger project that aims to develop an authentication and authorization support for distributing documents through the Internet.

References

1. Rivest, R.L., Lampson, B.: SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession (1996) http://citeseer.nj.nec.com/rivest96sdsi.html.
2. Ellison, C.M., Frantz, B., Lampson, B., Rivest, R., Thomas, B.M., Ylonen, T.: SPKI Certificate Theory. Internet Engineering Task Force RFC 2693. (1999)
3. Li, N.: Local Names in SPKI/SDSI. In: Proceedings of The 13th Computer Security Foundations Workshop, IEEE Computer Society Press (2000) 2–15
4. OMG: Object Management Group – Security Service v1.7. OMG Document 01-03-08 (2001)
5. Westphall, C.M.: Um esquema de autorização para segurança em sistemas distribuídos de larga escala. PhD thesis, Federal University of Santa Catarina – Brazil (2000)
6. Gollmann, D.: Computer Security. John Wiley & Sons (1999)
7. Hartman, B., Flinn, D.J., Beznosov, K.: Enterprise Security with EJB and CORBA. OMG Press. John Wiley & Sons (2001)
8. Santin, A., Fraga, J., Mello, E., Siqueira, F.: Um modelo de autorização e autenticação baseado em redes de confiança para sistemas distribu'ıdos de larga escala. In: Anais SSI 2002, São José dos Campos, SP – Brazil, ITA, SSI (2002) 101–110
9. Santin, A., Fraga, J., Mello, E., Siqueira, F.: Teias de Federações como extensões ao modelo de autenticação autorização SDSI/SPKI. In: Anais XXI Simpósio Brasileiro de Redes de Computadores, Natal, RN – Brazil, SBRC (2003) 553–568
10. Nikander, P., Viljanen, L.: Storing and Retrieving Internet Certificates. In: Proceedings of the Third Nordic Workshop on Secure IT Systems. (1998)
11. Morcos, A.: A Java implementation of Simple Distributed Security Infrastructure. Master's thesis, MIT (1998)
12. IONA Technologies Inc.: ORBacus User Guide. (2001) Version 3.3.4.
13. Adiron, LLC: ORBAsec SL2 User Guide. (2000) Version 2.1.4.
14. Institute for Applied Information Processing and Communications (IAIK): iSaSiLk 3 – Reference Manual. (2000) Version 3.
15. Rivest, R.L.: SEXP (S-expressions). Internet Engineering Task Force – Internet Draft (1997) http://theory.lcs.mit.edu/ rivest/sexp.html.
16. Orri, X., Mas, J.M.: SPKI-XML Certificate Structure. Internet Engineering Task Force – Internet Draft. (2001) http://xml.coverpages.org/ni2001-12-18-a.html. Visited in 04/02/2003.

17. Clarke, D.E.: SPKI/SDSI HTTP Server / Certificate Chain Discovery in SPKI/SDSI. PhD thesis, MIT (2001)
18. Lampinen, T.: Using SPKI certificates for authorization in CORBA based distributed object-oriented systems. In: Proceedings of the 4th Nordic Workshop on Secure IT Systems (NordSec '99), Kista, Sweden (1999) 61–81
19. Eronen, P., Nikander, P.: Decentralized Jini Security. In: Network and Distributed System Security Symposium – (NDSS 2001), San Diego, California (2001)

An Architecture for On-the-Fly File Integrity Checking

Mauro Borchardt, Carlos Maziero, and Edgard Jamhour

Graduate Program in Applied Computer Science
Pontifical Catholic University of Paraná
80.215-901 Curitiba – Brazil
Phone/fax +55 41 330 1669
{borchardt,maziero,jamhour}@ppgia.pucpr.br

Abstract. There are several ways for an intruder to obtain access to a remote computing system, such as exploiting program vulnerabilities, stealing passwords, and so. The intruder can modify system utilities in order to hide his/her presence and to guarantee an open backdoor to the system. Many techniques have been proposed to detect unauthorized file modifications, but they usually work off-line and thus detect file modifications only when the system is already compromised. This paper presents an architecture to deal with this kind of problem. Through the combined use of digital signature techniques and system call interceptions, it allows for transparent on-the-fly integrity check of files in Unix systems. Its evaluation in real-world situations validates the approach, by showing overheads under 10% for most situations.

1 Introduction

There are several ways for an intruder to obtain access to a remote computing system, such as exploiting program vulnerabilities, stealing passwords, hijacking network connections, and so on. After getting in the system, the intruder can exploit local vulnerabilities to get administrative privileges.

Once having the full control of the system, the intruder can modify some system utilities, in order to hide his/her presence, and to guarantee an open backdoor to that system. System utilities can be substituted by hacked versions that dissimulate the intruder's presence, hiding files, processes, network connections, and other system resources. The set of hacked utilities is known as a *rootkit*; sophisticated *rootkits* are hard to detect and may include changes in the operating system kernel itself [4].

Alongside with the efforts to minimize system bugs allowing root compromise, some work is done in detecting and/or preventing modifications of system files. Most known approaches work off-line, by periodically checking the system files' properties against previously stored values. One of the most known tools using this approach is *Tripwire* [11]. Although it can detect file violations, it is not able to prevent the use of the modified files, and the intruder may work for a while before being detected.

This paper presents an approach allowing to detect modified files and to prevent their usage as soon as possible. This is achieved through the joint usage of digital signatures and system call interceptions. The proposed architecture allows to

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 117–126, 2003.

transparently and quickly verify the integrity of any kind of file at its opening, including executables, libraries, scripts, and data files. Experimentation results validate the approach, by showing overheads under 10% for most situations.

The paper is structured as follows: section 2 introduces system call interception techniques; in section 3 the proposed architecture is detailed; in section 4 the performance results obtained with the prototype are presented and discussed; section 5 discusses the current status of the architecture and outlines future works; section 6 presents some related work, and section 7 concludes the paper.

2 System Calls Interception

To access operating system resources, such as files or sockets, processes make use of *system calls*, which are functions providing a controlled interface to the operating system kernel. The set of syscalls offered by the kernel constitutes its *API* (*Application Programming Interface*). The kernel API defines a clear separation between the specification and the implementation of the kernel services. This separation allows to transparently modifying the implementation of a given service: as long as the kernel API remains the same, all changes in underlying kernel services can be done with no need to modify the user-level code.

Extending basic kernel services can be done by modifying the kernel functions that implement the service to be extended, or by capturing the corresponding syscalls and transferring control to a code that implements the new feature [10,18]. Several architectures have been proposed to facilitate building and using kernel extensions, through plug-in structures and specific APIs. Some projects in this direction are *SPIN* [1], *Exokernel* [7], and *SLIC* [8].

There are several uses for kernel extensions, like process debugging, process migration between hosts, file systems extensions, and host-based intrusion detection systems, among others. A well-known example is the *Virtual File System* (VFS) [9], which allows for applications to transparently use multiple distinct media and file systems.

3 System Architecture

The goal of the system presented here is the dynamic file integrity checking. Before files are opened by processes, their integrity is checked using a digital signature[1] schema, which is activated by a syscall interception. All files to be monitored should have been associated a digital signature, which is verified by the operating system kernel just before opening them. If the signature generated from the actual file

[1] The *Digital Signature Standard* (DSS) [6] is the digital signature schema adopted in this work. The DSS aspect considered here concerns the ability to certify the integrity of a digital document, through an encrypted hash obtained from it. This encrypted hash is called a "digital signature" of the document.

contents matches the corresponding signature stored in the database, the file can be open (i.e. the original system call can proceed), otherwise the access is denied.

Starting from a list of files to monitor, a digital signature database should be built. Database entries are formed by the file paths and their respective digital signatures. At first, only static resources residing in the local file systems should be registered in the database: executable files, libraries and configuration files. The signature database should be built from the system in a known safe state. The best way to guarantee this is using a freshly installed system, avoiding the presence of intruder codes as *rootkits*, *viruses*, *backdoors*, and *trojans*.

Performance is a major concern in this approach. The signature checking procedure can be costly, especially on large files, and imposes a considerable performance penalty. To solve this problem, a *validation cache* is defined (section 3.3). It stores references to files already checked, to avoid repeating integrity checks on the same files. This cache has proven to be essential in this approach (section 4).

For the integrity checking mechanism to be transparent, the file system API should be respected. Also, the checking should be imposed on any file access, with no possibility to be circumvented. Thus, the system should be implemented under the kernel API. Considering the generic *POSIX* API, there are two system calls related to file opening that should be intercepted: `execve`, responsible for reading and starting an executable image from disk, and `open`, to open files for reading and writing.

The proposed architecture is presented in figure 1 and will be detailed in the following sections.

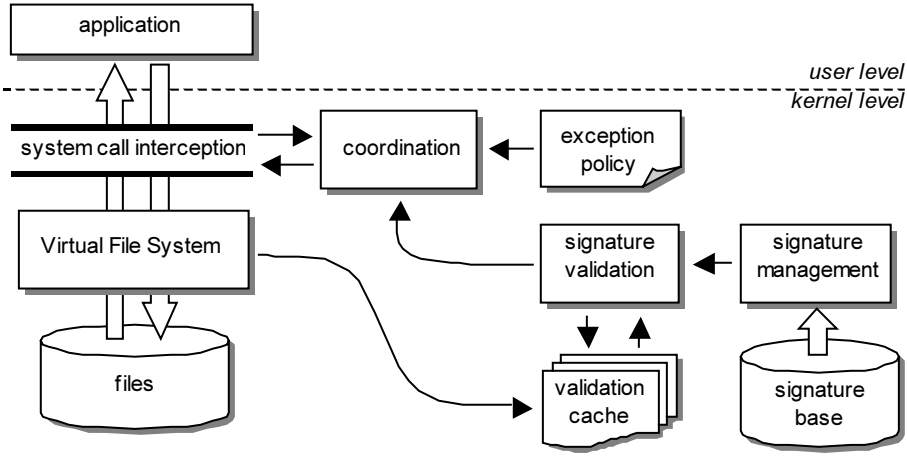


Fig. 1. The proposed architecture

3.1 System Call Interception Module

The **System Call Interception Module** intercepts the `open` syscall, gets the complete file name, and redirects the control flow to the coordination module, for signature checking. If the coordination module returns ok, the original system call is

allowed to continue, otherwise the file access is denied. The `open` system call is also used to open directories and devices, but these are excluded from the integrity checking.

The `execve` syscall is responsible for opening an image file and preparing the kernel structures, in order to start the code execution. Although the file is also opened, this system call doesn't use the `open` syscall code, but makes use of other kernel functions. Thus, the `execve` syscall is also intercepted by this module. Depending on the operating system, other system calls should be intercepted as well.

3.2 Coordination Module

This module receives the name of the file to check and tries to open it using the same privilege level of the process that made the system call. A lookup in the validation cache is performed, using the pair [*i-node; device number*] got from the file as a key. If the file to be open has an entry in the cache, its signature was already validated before and the file contents did not change after that. This result can be returned immediately to the syscall interception module (section 3.1).

In case the file is not referenced in the validation cache, its signature should be verified. The **Signature Verification Module** is then activated, in order to check the integrity of the file using its previously stored digital signature. If the actual file signature agrees with the previously stored one, the file is valid, and a corresponding entry is added to the cache. Otherwise, an error is returned to the syscall interception module. If the file to be open has no associated signature, the **Exception Policy Module** will be activated, to decide whether to allow or to deny that access.

3.3 Validation Cache Module

As the signature verification is time-consuming, it is important to avoid repeating unnecessary verifications. This is achieved using a cache of validated files. The signature verification costs would turn unviable this proposal if there was not such a cache, as demonstrated by the results presented in the section 4. This module implements three operations on the cache: *insert*, *delete*, and *lookup* of files entries.

One main aspect in the cache management concerns entry deletions. An entry should be deleted from the cache if, and only if, the corresponding file is modified or removed. Some mechanisms from the VFS (Virtual File System) abstraction, present in most Unix systems, are used for detecting such events. In the implementation presented here, a hook was put in the VFS function `mark_inode_dirty`, which is called whenever a file is modified or removed.

3.4 Signature Validation Module

This module makes use of a digital signature algorithm to check the integrity of a file. For its application the module uses the digital signature of the file, obtained from the signature database, the hashing value of the current file content and the public key of

the signature database. The access to this public key is a critical issue and will be discussed in section 5.

Before calculating the hash value for the file being open, its digital signature should be requested to the **Signature Management Module**. The hash value for the file is then calculated, using as input the following information: file contents, i-node number, device, size, owner, group, permissions, creation date, and last modification date. The simultaneous application of file attributes in the hash function simplifies the attribute verification procedure.

The digital signature standards adopted were SHA-1 [5] for the hash generation procedure, and FIPS 186-2 ECDSA [6] for the encryption procedure. The FIPS 186-2 ECDSA algorithm was chosen for its smaller execution time, smaller memory footprint, and smaller keys for the same protection level when compared to DSA [12].

3.5 Signature Management Module

This module was included in the architecture to provide transparency on the signature database location. Basically only two parameters are needed to uniquely identify a file: its complete path and the name of the host where it is stored. Using them, this module retrieves the signature value from the database and returns it to the signature validation module. The digital signature database can then be located anywhere, from a file in the local system to a remote LDAP server. Details regarding its implementation issues are discussed in section 5.

3.6 Exception Policy Module

It is not worthwhile to maintain up-to-date signatures for files whose contents are always changing, like log files and mailboxes. The **Exception Policy Module** was included in this proposal to implement policies to deal with these exceptions, allowing to define unsigned directories and files.

The coordination module activates it in order to verify whether a requested operation is allowed or not, according to policies defined by a rule set. The current implementation uses simple rules based on files, directories, users, groups, and operations (`open`/`execve`).

4 Implementation and Evaluation

The prototype implementation was done using a Linux kernel version 2.4.4; all the modules described here were implemented, some of them in a simplified way. The tests were run on a Pentium II 233MHz system, with 128 Mbytes of RAM and an IDE ultra-DMA33 hard disk (seek time around 13 ms).

Three distinct situations were chosen to evaluate the impact of the architecture: compilation, file compression, and user application. They have very different behaviors regarding resource usage (CPU, memory, I/O) and represent frequent situations. Due to the validation cache, subsequent accesses to a file are much faster

than its first access. To measure this effect, each test was run in three situations: a) without the use of signature verification (the *reference system*), b) a first execution after system initialization (the validation cache is empty) and c) a second execution after system initialization (the validation cache has entries generated by the first execution). Each test was run several times, and the results obtained are stable.

To simplify the analysis of results, a "dummy" exception policy was adopted, allowing any file to be accessed, even if is not signed or if it's signature is not valid. For the tests, all files were signed, excepting temporary ones (in `/tmp` and `/var/log` directories).

4.1 Compilation

This test consisted on compiling the Linux kernel source (version 2.4.4, with standard parameters: `make bzImage modules`). The source tree occupies 135 Mbytes on disk and has 8875 files, but only 2051 of them were read in the compilation. The table 1 presents the results obtained.

Table 1. Compilation test results

Measurement	Reference system	1st execution		2nd execution	
#`open` operations	115494	115494		115494	
#`open` cache hit	—	113424	98.2%	114661	99.3%
#`open` cache miss	—	2070	1.8%	833	0.7%
#`execve` operations	2344	2344		2344	
#`execve` cache hit	—	2326	99.2%	2340	99.9%
#`execve` cache miss	—	18	0.8%	4	0.1%
user time (s)	663.12	664.80		662.24	
system time (s)	41.42	134.70		90.01	
total time (s)	704.54	799.50		752.25	
overhead (%)	—	13.47%		6.77%	

The meaning of the table fields are:

- **#`open`** (**#`execve`**): total number of syscalls invoked by the process(es).
- **#`open` cache hit** (**#`execve` cache hit**): number of syscalls in which the file was found in the cache.
- **#`open` cache miss** (**#`execve` cache miss**): number of syscalls in which the file was not found in the cache, forcing a complete signature verification.
- **User time** (s): time spent by the process(es) at user level.
- **System time** (s): time spent by the process(es) at kernel level.
- **Total time** (s): total time spent by the process(es).
- **Overhead** (%): the execution time compared to the reference system time.

4.2 File Compression

This test consisted on applying a `tar -czf` command on the Linux kernel source tree. This command applies both data compression (using the `gzip` tool) and agglutination of the files to generate an archive containing all the source files compressed. The obtained results are presented in table 2.

Table 2. Compression test results

Measurement	Reference system	1st execution		2nd execution	
#`open` operations	8876	8876		8876	
#`open` cache hit	—	12	0.14%	8876	100%
#`open` cache miss	—	8864	99.86%	0	0%
#`execve` operations	2	2		2	
#`execve` cache hit	—	0	0%	2	100%
#`execve` cache miss	—	2	100%	0	0%
user time (s)	55.28	55.75		55.27	
system time (s)	2.56	293.33		3.31	
total time (s)	57.84	349.08		58.58	
overhead (%)	—	503.5%		1.28%	

4.3 Execution

This experiment used the **`gv`** application, a popular PostScript file viewer. It consisted on opening a 40-pages PostScript file, quickly viewing all pages and closing it. The obtained results are shown on table 3.

Table 3. Execution test results

Measurement	Reference system	1st execution		2nd execution	
#`open` operations	117	117		117	
#`open` cache hit	—	46	39.3%	117	100%
#`open` cache miss	—	71	60.7%	0	0%
#`execve` operations	2	2		2	
#`execve` cache hit	—	0	0%	2	100%
#`execve` cache miss	—	2	100%	0	0%
user time (s)	20.56	20.83		20.59	
system time (s)	1.52	11.31		1.57	
total time (s)	22.08	32.14		22.16	
overhead (%)	—	45.56%		0,36%	

The results got from this test should not be considered as-is. Their goal is just to give a feeling about the performance impact of the system, from an ordinary user perspective. For a more precise numeric evaluation of this test case, one would need to measure the time spent by the graphical interface processes, the I/O devices, and the user latency on controlling the application.

4.4 Result Analysis

The signature verification procedure imposes a significant computing overhead, as shown by the differences between the first execution and the reference system execution. The first execution can be very time-consuming, because the validation cache is barely used. This is perceptible in the compression experiment: the first execution takes 503% more time than the reference system. This bad performance occurs because each file being compressed is accessed only once (cache miss ratio near to 100%). The validation cache influence is visible on the second execution of this experiment, which states an overhead of only 1.28% and cache hit ratios of 100%. This allows concluding that the overhead is around 1% if the cache is fully used.

On the other hand, the compilation experiment shows a much better picture for its first execution: only 13% overhead. This is due to the way compilation is done: header files are often included on several different files, generating lots of accesses on them. This can be seen on table 1: although 115494 `open` syscalls were performed, they generated only 2070 cache misses. The same can be concluded on the `execve` side: few tools (assembler, compiler, linker, etc) are extensively called, generating high cache hit ratios. Here, the second execution shows results not so good as in the compression experiment. This is due to the fact that the compilation modifies configuration scripts and object files that should be checked again.

The execution experiment shows that the performance impact for an average interactive user is acceptable. Although the first execution presents an overhead around 45%, the second execution overhead is near zero. In a multi-user environment, only the first user of an application will suffer from the first execution overhead. In this case, subsequent users will notice no overhead.

5 Current Status

Presently, the system is able to check file signatures and to allow/deny access to files depending on their signatures. This makes it useable in servers for common services like e-mail, web, proxy, and file sharing. Considerable work remains to be done in the signature generation and management aspects, before the system is ready for production multi-user systems. Some points for future work are outlined here.

As results show, the first access to a file is time-consuming. For large applications, this means an uncomfortable starting delay for its first user. For network service daemons, such response delay can leads a network client to time-out. This problem can be minimized by “proactively” populating the validation cache (by running a routine to check all files registered in the signature base, in moments of low system activity).

The system public key needs to be accessible to the kernel during the system initialization. It can be passed as a kernel boot parameter, made available from a read-only external media, or even be fetched from a remote key server. The public key should not be available for write access during system operation. If the public key is modifiable, the signature validation mechanism can be bypassed by tampering the signature base and re-signing all registered files.

Presuming that the public key is kept safe from tampering, the signature base file can be maintained on the local file system, protected from access by normal users. It does no need to be encrypted, as it is also signed and will be verified like other files. This is also valid for the exception rules file. In the case one of these control files is corrupted or removed, access to the signed files will be denied. As a consequence, the system will be useless, but preserving its integrity. A better solution would be to store these files in a remote server, to be fetch during system initialization.

User files are frequently modified. The file signature base would need to be frequently adjusted to reflect changes in those files, but this can be unfeasible if the signature base is a static and protected resource. One solution to this problem is to create exception rules for user directories, as stated in section 3.6. However, allowing file execution in user directories can open security breaches. A safer approach would be to allow trusted users to sign their executable files, using personal keys. This can be achieved by creating a local Public Key Infrastructure (PKI). The trusted users' keys should be certified by the system public key, and would be used to generate signatures for their files, which are stored in personal signature bases.

6 Related Work

The most known work in the domain on file integrity check is the *Tripwire* system [11], which inspired this work. It is a user-level application operating in batch mode (usually when the system load is low). Tripwire is strong in detecting file modifications, but is unable to prevent them. The *Bsign* project [17] is similar to Tripwire, but applies only to executable files in ELF format. It stores each file signature in the executable file itself (ELF files support "sections" in which particular data can be stored). The SOFFIC system [16] and the system presented in [2] for the NetBSD operating system have a similar approach to the system presented here, but at this moment there are no concrete results or analysis widely available about them.

The idea of signing files and checking their signatures on demand is not new and can be applied to different domains. The work presented in [15] uses this approach to validate web pages by the web server, as they are requested by remote clients. Their goal is to prevent delivering hacked pages to the Internet.

7 Conclusion

In this paper we present a software architecture to improve the security of a computing system using file integrity checking. It does not prevent a system to be intruded, but denies access to tampered files or to files dropped in protected areas. The proposed approach makes use of digital signature and system call interception techniques. Although it was implemented on an UNIX-like operating system, it can be adapted to other operating systems.

The performance results presented show that the proposed approach is worthwhile in real situations. Once the validation cache has enough information, the overhead imposed on the applications considered is under 10%.

The proposal is not complete, as it lacks features to improve user key management. Some possibilities to improve it were enumerated, but there are other approaches to consider. We are currently studying aspects related to user key management using a distributed PKI, and defining strategies in the case of network outage.

References

1. B. Bershad, C. Chambers, S. Eggers et al. SPIN: An Extensible Microkernel for Application-specific Operating System Services. In *Proc. 6th ACM SIGOPS Workshop on Matching Operating Systems to Application's Needs*, Warden, Germany, Sep 1994.
2. B. Lymn. *Verified Executables for NetBSD*. `http://members.optusnet. com.au/~blymn`, Nov 2002.
3. CERT Coordination Center. `http://www.cert.org`, Nov 2002.
4. ChkRootKit.org. `http://www.chkrootkit.org`, Nov 2002.
5. FIPS 180-1. *Secure Hash Standard*. NIST, Apr 1995.
6. FIPS 186-2. *Digital Signature Standard*. NIST, Jan 2000.
7. D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles*, 1995.
8. D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proc. USENIX 1998 Annual Technical Conference*, pages 15-19, Berkeley, USA, Jun 1998.
9. J. Heidemann and G. Popek. *A Layered Approach to File System Development*. University of California, Los Angeles, Technical Report, CSD-910007, 1991.
10. M. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. 14th ACM Symposium on Operating Systems Principles*, pages 80–93, Dec 1993.
11. G. H. Kim and E. H. Spafford. *The design and implementation of Tripwire: a file system integrity checker*. Tech. Report CSD-TR-93-071, Computer Science, Purdue Univ, 1993.
12. Menezes, P. Van Oorschot, S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1999.
13. Ronald L. Rivest. *The MD5 message-digest algorithm*. IETF RFC 1321, Apr 1991.
14. R. Rivest, A. Shamir, and L. Adleman. *A method for obtaining digital signatures and public key cryptosystems*. Communications of the ACM, 21(2): 120–126, 1978.
15. S. Sedaghat, J. Pieprzyk, and E. Vossough. *On-the-fly web content integrity checker boots user's confidence*. Communications of the ACM, 45(11):33–37, Nov 2002.
16. V. Serafim, A. Reguly. *SOFFIC – Secure On-the-Fly File Integrity Checker*. URL `http://www.inf.ufrgs.br/~gseg/projetos/soffic.shtml`, Nov 2002.
17. M. Singer. *The BSign Project*. `ftp://ftp.buici.com/pub/bsign`, Feb 2002.
18. C. Wright, C. Cowan, J. Morris, S. Smalley, G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. 11th USENIX Security Symposium*, San Francisco, CA, Aug 2002.

Fault Injection Tool for Network Security Evaluation

Paulo César Herrmann Wanner and Raul Fernando Weber

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
PO Box 15.064 - 91.501-970 - Porto Alegre - RS - Brazil
{herrmann,weber}@inf.ufrgs.br

Abstract. Lately, a great number of new security tools have been developed to solve the problems of IP networks, but some doubts were raised about the efficiency of those systems. There are some techniques used to evaluate the vulnerabilities of a network, but their results are limited and few tools aim to facilitate and automate the evaluation of security network mechanisms. This article presents a network fault injection system especially created to test network security equipment, aiming to solve the inexistence of this type of tool and to fill the gap between packet injectors and vulnerability scanners.

1 Introduction

In the last years, with the increasing use of the computer networks based on the IP protocol, small networks and metropolitan networks were joined together and became part of what today it is known as the world-wide computer network. Despite the benefits of communication and information exchange through the Internet, this global phenomenon also brought some security problems, mainly because the origin and structure of the protocols limit the possibilities to prevent, to identify or even to detect possible attacks or intruders.

Thus, some tools were developed to prevent attacks and to assist in the task of identifying security problems in networks such as firewalls, sniffers and intrusion detection systems (IDS) [1]. Despite the benefits brought by these new technologies, some doubts were raised about the real security that these resources provide. After all, the development and validation of these systems are sufficiently complex and, frequently, these systems become the primary target of an intruder. As a consequence we have, not rarely, a false feeling of security due to the inadequate use of these mechanisms, which is, usually, more harmful than the simple lack of security in an organization whose unreliability is known by its administrators.

Several techniques exist to verify the security of a network, but we still do not have a simple and efficient methodology that is able to evaluate with precision the vulnerability level of an organization. Currently, it is possible to identify vulnerabilities in a system and to verify the attack tolerance that a network supports, but, unfortunately, we can not state that a network is safe and what is the level of its security.

Besides the security evaluation techniques, it is very important to have tools for carrying out these security tests. Thus, it is possible to automate the tests execution and to easily verify the existence of problems in a network. The existence of tools that test systems security is extremely important and necessary, since the security of the whole network can be strongly dependent on some of these systems.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 127–136, 2003.

2 Network Security Evaluation Tools

A survey was carried out on the Internet to find out available free tools for fault injection in network security systems. The majority of the tools found are inadequate for the accomplishment of these tests, because they were developed for other purposes [5]. There are packet injector tools and vulnerability scanners programs.

The packet injector tools, also known as packet generator programs, are tools that allow creating and sending TCP/IP packets to a certain station. So, it is possible to create any type of packet, forging communications and data origins with the purpose of testing systems as firewalls and intrusion detection systems. Some of the existing tools on the Internet for generating and sending TCP/IP packets have been analyzed, such as hping [7], nemesis [3] and sendip [6].

The other tools found are the vulnerability scanners. These tools are developed with the objective of automating the search for faults in services and stations in a certain network. Although they are not programs that carry out tests in a network, they are normally used to test the detection mechanisms of an IDS, because many intruders use them to scan vulnerabilities in a potential target. Some programs found on the Internet for this purpose include Nessus [2], SAINT [9] and SARA [8].

The current problem in using these type of software to inject fault in network security systems is that they do not supply the necessary features for such goals. The packet injection tools in a general way are in text mode, do not send sequences of packets, have few resources to monitor the destination computer answers and do not allow the implementation of sophisticated tests. The vulnerability scanners do not have a monitor mechanism of the target answers accessible to the end user, the user can not create a test to be executed, except for Nessus [2], the most of the injected faults are in the application level and they test vulnerabilities in services and stations of a network and not in security systems such as firewalls or intrusion detection systems.

3 Proposed Model

As could be seen in the previous section, there are two types of tools that can be used for fault injection in a network. The ideal tool to inject faults in network security mechanisms is in a middle way between these two types of program. It is a fault injector in network level similar to a packet generator, but with a greater flexibility of use as vulnerability scanners. In this in case, it must be able to generate and send protocol packets in the network, transport and application levels. Moreover, it must also have a programming language that allows sending sequences of packets based on the monitored answers of the target computer.

According to Hsueh [4], a fault injection system must have the following elements: fault injector, fault library, workload generator, workload library, controller, monitor, data collector and data analyzer.

The proposed system, as a fault injector, should execute all the tasks of a fault injector system. However, considering that it carries out tests in network security equipments, some of these attributions can be simplified. The proposed model implements these elements through three modules, fig. 1 : management module, injection and monitoring module and test module.

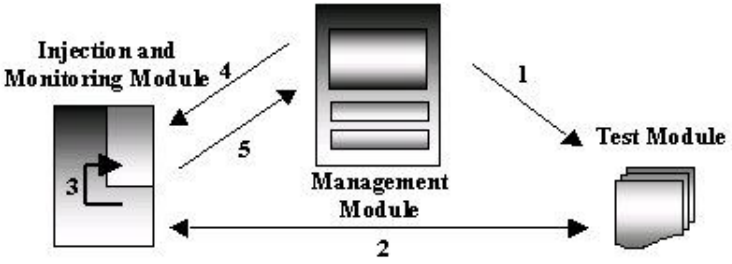


Fig. 1. Proposed Architecture

The management module is responsible for the start up, finish up, fig. 1 (1), and monitoring, fig. 1 (4, 5), of the execution tests and for the graphical interface with the end user. It implements, thus, some of the functions designated to the controller and the monitor of a fault injection system. Another important functionality played by the management module is the packet edition. The management module, through its graphical interface, allows the creation and edition of packets to be used in the evaluation of network security systems.

The injection and monitoring module, as the name suggests, is responsible for the fault injection and the data collection. The fault injector is represented by the generation and sending of TCP/IP packets, while the data collector is represented by the monitoring of answers generated by the tested station. These two functions are provided to the system end users, so that they can develop their own tests (plug-ins).

The injection and monitoring module must supply the basic functions to fault injection as Ethernet, ARP, IP, UDP, TCP and ICMP sending functions, functions for network monitoring and advanced security test functions such as dnsspoof, tcpkill, arpspoof and scanning. This way, the end user can accomplish simple tests through the TCP/IP sending functions and has a starting base to develop more sophisticated tests through the attack functions such as dnsspoof, tcpkill, etc.

Beyond the test functions, this module implements some specific functions for the management module: communication, management and statistics, fig. 1 (3, 4, 5). These functions are relevant for the exchange of information between the management module and the test modules. They also allow the user to gather test information during the execution and to get a summary of each test at its end.

An important feature in a tool for fault injection in network security system is that the end users can develop their own tests, not being obliged to depend on the tests supplied with the tool. Therefore, a programming language for test creation is necessary. The desirable characteristics in such a language are simplicity, efficiency and functionality. It must be efficient so that the translation or compilation of the tests do not represent an overhead of the execution.

Thus, the fault library, the workload generator and some of the controller and the monitor functions of the fault injection system are responsible for the test modules in execution. The user of the tool, based on the fault type or traffic load that he desires to inject, must create and determine the logic of the test or workload in execution. A set

of attack models, workloads and packets may be defined and supplied together with the tool, however the user is also able to create its own attacks and test packets.

Through the test module and its programming language, the proposed model reaches a greater flexibility and portability than the usual packet injection tools, having features similar to a vulnerability scanner. The whole communication between the test modules and the tool is made through the injection and monitoring module functions supplied to the user, fig. 1 (2, 3, 5). The only interaction between the tests and the management module is during the plug-in start up and finish up, fig. 1 (1).

The data collector and the data analyzer have their functions shared between the fault injector and the security system tested. This is due to the fact that one of the fault injection goals is to verify the detection capacity, the correct logging and the correct alarm and countermeasures activation of the network security system being tested. Thus, the data collection of the fault injector is restricted to the answers of the tested equipment and to the existing traffic in the network, while the analysis is made by the user of the fault injector through the functions provided by the injection and monitoring module. Any other data must be collected and analyzed by the tested equipment, and its correct functionality depends on the executed tests, its security policies and its correct configuration.

4 Implementation

In this section, a network fault injection prototype is described. The main implementation's goal is to validate the previously proposed model and to evaluate its potentialities. The characteristics and the implemented functionality of the management module, injection and monitoring module and test module are also presented.

4.1 Management Module

The management module was developed in C language for the GNU/Linux platform and presents a graphical interface developed using the GTK (GIMP Toolkit) library. Through this module it is possible to manage the TCP/IP packets that will be injected in a network and the tests modules that will be executed by the tool.

This module has an interface to edit TCP/IP packets. Through this interface it is possible to open, create, edit and send sequences of TCP/IP packets, fig. 2 (1). Currently, the prototype sends Ethernet, ARP, IP, UDP, TCP and ICMP packets.

The fault injector was developed in order to allow a great degree of flexibility and usefulness to the end user. All the implemented headers are available in a tag structure that facilitates the movement between the diverse protocols, fig. 2 (2). Most of the protocol fields must be filled in decimal, except for some fields such as checksum and payload that must be filled in hexadecimal.

No data consistency is made, so the program tries to send any entered data. This characteristic is essential, because it allows the user to create the most diverse types of packets, even if they are incorrect, and to explore potential vulnerabilities in systems and protocols. All the packets are saved in the Libpcap format, allowing a greater compatibility with other systems and a greater ease of use.

As no test of data consistency is carried out, some errors can be generated, intentionally or not. For example, an ICMP packet can be created and saved as a TCP packet.

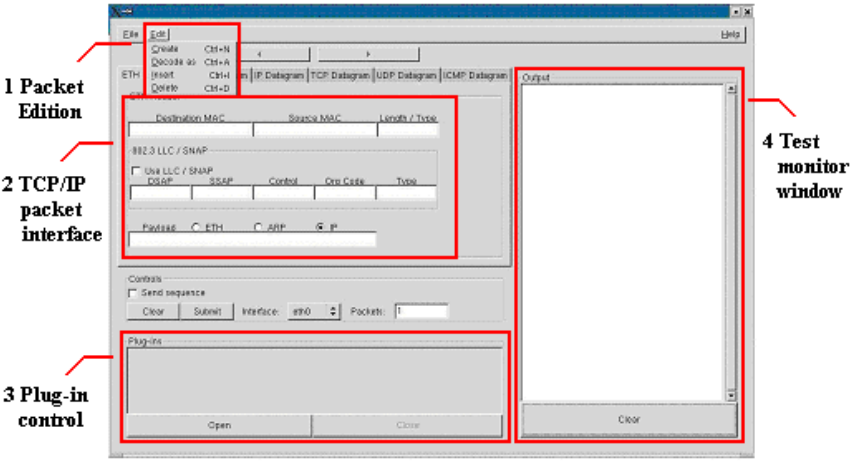


Fig. 2. Management Module

The problem occurs at the moment that we try to open this packet again. Any program that supports the Libpcap format will decode the packet as a TCP packet, despite of its inconsistencies. Thus, the user could not edit his ICMP packet again, because, now, it became a TCP packet. The solution was to supply a function so that the user can specify the type of decoding that must be made in a packet, regardless of the packet's data.

Moreover, a series of functionalities to edit sequences of packets is also supplied, such as creation, insertion and exclusion of packets. This tool allows new packets or sequences of packets to be created through the graphical interface or through previously saved packets, fig. 2 (1).

Another functionality of the management module is the control and execution of plug-ins, fig. 2 (3). After opening a test module (plug-in), it is possible to carry out three types of operations: to initiate the plug-in, to stop the plug-in and to ask for information about the plug-in.

When initiating a plug-in, a specific thread is created for the accomplishment of its tests. The thread control is passed to the user, receiving the parameters in the argument field. After the plug-in starts up, we can call the stop function, which tries to finish the test in execution. However, as the tests are implemented by the user, we do not have guarantees that they will finish correctly. Thus, in case it is not possible to stop a test execution in a certain period of time, the plug-in thread is unconditionally finished. The third function shows information on the plug-in usage and functions.

The management module can also control and monitor the tests in execution. These tests can be carried out either by the management module itself, or by the use of plug-ins, as it can be seen in fig. 2 (4). The monitoring consists basically of opened, saved or sent packet information, and any other type of information sent by the plug-in to the management module.

4.2 Injection and Monitoring Module

The injection and monitoring module, as previously specified, supplies the basic functions for packet injection and network monitoring, as well as more sophisticated functions for security equipment tests. To achieve this objective, this module was implemented as a library in C language, in such a way that it can be easily used by the management module and by the developed test modules.

The injection and monitoring module supplies an API that currently allows carrying out four types of operations: create TCP/IP packets, send packets, monitor packets in the network and send messages for the management module.

The supplied API allows the creation of any type of packets, however it is aimed to create common packets such as Ethernet, ARP, IP, TCP, UDP and ICMP. The functions for packet creation have been developed in such a way that the user can create protocol frames independently and latter add them to other protocols as their payloads. The advantage of this approach is the great freedom that the user has for creation of packets without any type of verification. Thus, Ethernet frames can be created and then added as payload to a TCP packet, for example. This functionality allows the user to explore possible protocol weaknesses and to generate specific tests for specialized network equipments.

For packet sending, the supplied API consists of four functions: sendeth, sendarp, sendip and sendfile. The first three functions allow specifying a packet in different levels of details, thus their selection depends on the desired detail level. However, it is important to point out that the operating system can carry out some validation tests in the packets it sends, depending on the level of detail being used. For example, in certain Linux releases, if a packet is specified at the network level, the operating system verifies the IP header, and checksum errors are corrected. Thus, in case that an IP packet incorrect checksum must be sent, this packet must be created in the link level. The fourth function sends packets or sequences of previously saved packets. This function simply opens an archive in Libpcap formats and sends all the existing packets. All sending functions are blocking, thus in case a great packet sequence must be sent through the function sendfile, the test module that called it will be blocked until all the existing packets in the archive are sent.

Besides the creation and sending functions there are two other classes of functions supplied by the injection and monitoring module API. These are a function for network monitoring and a function for communication between the management module and the plug-ins.

The first one, the waitpacket function, is implemented using the Libpcap library capture functions, and provides a higher abstraction level for the end user. This abstraction allows the user not to worry about implementation details. He only needs to call one specific function, instead of making several function calls to the Libpcap. This function is also blocking and monitors the network in search of packets that satisfy the filter specified by the user. When a packet matches the defined filter, it is returned to the user for further processing.

The writemsg function, in turn, allows the user to send messages to the management module. These messages can be displayed, either to inform over the plug-in execution or over the occurrence of important events. When it is called, this function writes the received message in a communication channel created between the management module and the injection and monitoring module. The injection and monitoring module, as

previously specified, supplies the basic functions for packet injection and network monitoring, as well as more sophisticated functions for security equipment tests. To achieve this objective, this module was implemented as a library in C language, in such a way that it can be easily used by the management module and by the developed test modules.

4.3 Test Module

The test module, as seen previously, is responsible for the fault library implementation and for some functions of the controller and monitor of the fault injection system. All the logic of the test execution is determined by the user through this module. The test execution depends on the type of faults to be injected, therefore it is important that the tool provides a functional, simple and efficient programming language that allows a great degree of flexibility to define and control the test execution.

There are two basic solutions for this problem. A new language can be defined, trying to optimize the execution of security equipment tests, through the definition of specific structures and functions. As an alternative, an existing language can be used, together with its known characteristics, disadvantages and all existing infrastructures.

Between these alternatives we have chosen to use an existing programming language instead of creating a script language for the tool. The creation of such a language would cause the necessity of also providing mechanisms for the control, compilation and execution of this language, besides the definition of structures, control functions and statements for this language, all tailored to test execution. Moreover, the creation of a new language would not guarantee a better test definition, or a better performance of test execution.

Thus, the plug-ins are implemented using C language and supplied in the form of a dynamic library to be loaded by the tool. For a rational use of this library, some standard functions must be defined in the test module. The management module shall call these functions at the moment of the test execution: _start, _stop and _info.

The _start function is called by the management module when initiating the execution of a plug-in. The parameters passed are a plug-in identification and a sequence of values assigned by the user. One thread is especially created at the beginning of this function, so that the plug-in can execute concurrently to the management module. This also allows more than one test to be executed simultaneously.

The _stop function, in turn, informs to the plug-in in execution that it must be finished up. In this case, this plug-in must take the necessary measures for the correct closing of the tests in progress. The _info function is used to retrieve a summary of the test implemented by the plug-in.

It is important to notice that the test modules are independent from the management module. The management module does not control the tests in execution neither their correct implementation. The only existing form of control is over the thread created for the test execution. In this way, the only possible action is to cancel a thread, and, consequently, to abruptly finish the test under execution. An example of plug-in is described in the next section.

4.4 Test Example: TCP Kill

The test example described bellow aims to verify the detection capacity of an IDS to a specific attack, in this case the TCP kill attack. This attack is classified as a denial

of service attack that consists of finishing all existing TCP connections in a local network, except for the intruder's connections. The TCP kill is a very simple attack to be implemented and it uses as advantage some fragilities of the TCP protocol: weak authentication, weak integrity and connection closing.

This attack can be divided in two phases. The first one consists of an intruder monitoring the local network waiting for TCP connections and storing some important data for future use such as source IP address, destination IP address and the next sequence number expected. The second phase consists of forging TCP RST packets with the information obtained in the first phase.

As seen previously, the plug-in must have three functions: _start, _stop and _info. The existence of these functions is essential, because they are called by the management module in the plug-in start up, finish up and information retrieval, respectively. Moreover, it is necessary to include the injection and monitoring module header so that we can use its functions in a plug-in.

Then, we will first declare the _info function, because it is very simple. The function copies one string that explains the test to a global variable that is returned when the function is called.

```
char *_info (unsigned char id)
{
strcpy (info, "This plug-in kills all tcp connections in the local network.\n");
return info;
}
```

The _start function is the core of the plug-in and needs to carry out a series of tests and to keep some information on the existing connections, as seen previously, therefore some auxiliary variable are necessary. The first action of the _start function, after called, is to inform the management module that it has initiated its execution. After this, it initiates a loop waiting for TCP packets in the local network. The functions writemsg and waitpacket provided by the API are called in order to perform theses operations. Through the waitpacket function parameters we can notice that the monitored interface is eth0 and the packets that must be returned are only the TCP ones.

```
void _start (unsigned char id, char *input)
{
writemsg (id, "Starting tcp kill...\n");
do
{
writemsg (id, "Waiting for a tcp packet...\n");
/* Wait for a TCP packet */
packet = waitpacket (id, "eth0", "tcp", &length);
```

The waitpacket function is blocking and only returns when a TCP packet is captured. After getting a TCP packet, we must search for important information for the attack. Thus, the source IP addresses, the destination IP address, the sequence number, the source port and the destination port are copied to auxiliary variables.

```
if (packet != NULL)
{
...
/* copy ip source and destination */
ip_addr.s_addr = ip->saddr;
sprintf (ip_source, "%s", inet_ntoa (ip_addr));
ip_addr.s_addr = ip->daddr;
sprintf (ip_dest, "%s", inet_ntoa (ip_addr));
...
```

```
    /* copy tcp ack, source port and destination port */
    sprintf (tcp_ack, "%lu", ntohl (tcp->ack_seq));
    sprintf (tcp_sport, "%u", ntohs (tcp->source));
    sprintf (tcp_dport, "%u", ntohs (tcp->dest));
```

After getting all the necessary data from the captured packet, the second phase of the attack initiates. This phase consists of forging a TCP RST packet to finish the existing connection. We need to use the collected information of the TCP packet and the packet construction functions of the API, such as frame_tcp and frame_ip. The code that follows shows the use of these functions as well as the previously copied data. First we create the TCP header, because it is necessary to construct the IP packet.

The source and destination IP addresses and the other data to create the TCP segment are informed to the frame_tcp function so that it can calculate the TCP packet checksum. The length_tcp variable address is also passed as a parameter so that the functions can calculate the final size of the packet being created. This same variable is passed as parameter for the frame_ip function together with the TCP header, now the payload variable.

After sending the TCP RST packet, a message is sent to the management module informing the identification data of the killed connection. And, at the end of the main loop, the memory used by these functions is released to prevent memory shortage problems; additionally the variable that keeps the packet size must also be zeroed.

```
    /* create tcp frame RST, ACK */
    payload = frame_tcp (id, ip_dest, ip_source, tcp_dport, tcp_sport, tcp_ack, "0", "0", "",
FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, "0", "", "0", "", "", &length_tcp);
    /* create ip frame */
    packet_to_send = frame_ip (id, "4", "5", "0", "", "0", TRUE, FALSE, "0", "64", "6", "",
ip_dest, ip_source, "", payload, &length_tcp);
    /* send packet */
    size = send_ip (id, packet_to_send, length_tcp, ip_source);
    ...
    /* free memory */
    free (packet);
    free (payload);
    free (packet_to_send);
    }
    length_tcp = 0;
    } while (i == 0);
    }
```

As can be observed, the plug-in developed is very simple. The last function necessary for this plug-in is the _stop function. This function consists of modifying a variable of the plug-in that finishes the main loop and sends a message to the management module informing the end of the test execution. Moreover, nothing hinders that other auxiliary functions are implemented in a plug-in, not being necessary to restrict the plug-ins' functions to the explained ones.

```
    void _stop (unsigned char id)
    {
    i = 1;
    writemsg (id, "Leaving tcp kill...\n");
    }
```

5 Conclusion and Future Work

Currently, the Internet is being used with goals that are different from the initial ones. The enormous expansion that the Internet has had since its creation and the purposes

for which it is being used, such as electronic business and electronic banking, were not foreseen at the beginning. Thus, the Internet had to adapt to the new requirements and necessities.

The security concern increased, since innumerable sensible data began to be sent through the Internet. New technologies have been created to solve these unexpected difficulties faced by the Internet, trying to prevent possible invasions and attacks. The correct design, development and functioning of these new systems are essential for the security of a network. Therefore, specific techniques and tools for the evaluation of security equipment in a network are extremely important.

This work presented a specific tool to inject faults in network security systems in order to test their effectiveness, efficiency and correct functioning. The fault injector was projected to have a small granularity of attack, injecting packets on the network level, thus being able to test problems in protocols and systems used in the network and at the same time, carry out more sophisticated attacks through the use of plug-ins.

References

1. Bace, R. (1999) "An Introduction to Intrusion Detection & Assessment", Scotts Valley: Infidel, 1999.
2. Deraison, R. (2000) "Nessus Security Scanner", http://www.nessus.org/, December 2002.
3. Grimes, M (2000) "Nemesis Packet Injection" , http:// www.packetfactory.net/Projects/nemesis/, December 2002.
4. Hsueh, M., Tsai, T. K., Iyer, R. K. (1997) "Fault Injection Techniques and Tools", Computer, New York, v. 30, n. 04, p.75–82, April 1997.
5. McHugh, J., Christie, A., Allen, J. (2000) "Defending Yourself: The Role of Intrusion Detection Systems", IEEE Software Computer, New York, v. 17, n. 5, p. 42–51.
6. Ricketts M. (2000) "SendIP", http://www.earth.li/projectpurple/progs/sendip.html, December 2002.
7. Sanfilippo, S. (2001) "Hping", http://www.hping.org, December 2002.
8. Todd, B. (1999) "Security Auditor's Research Assistant", http://www-arc.com/sara/index.shtml, December 2002.
9. World Wide Digital Security, Inc. (2000) "Security Administrator's Integrated Network Tool", http://www.wwdsi.com/saint/, December 2002.

Emulation of Software Faults: Representativeness and Usefulness

Henrique Madeira[1], João Durães[2], and Marco Vieira[2]

[1] University of Coimbra, DEI-CISUC,
3030-199 Coimbra, Portugal
henrique@dei.uc.pt
[2] Polytechnic Institute of Coimbra, ISEC; CISUC,
3030-199 Coimbra, Portugal
{jduraes, mvieira}@mail.isec.pt

Abstract. This paper discusses the problem of the emulation of software faults by fault injection. The first part of the paper investigates the possibilities of accurate emulation of real software faults by a SWIFI tool (Xception). Results revealed the limitations of Xception (and other SWIFI tools) in the emulation of several classes of software faults. Based on this first conclusion, the paper presents G-SWFIT which is, to the best of our knowledge, the first technique specifically proposed for the injection of software faults. The feasibility and accuracy of G-SWFIT have been experimentally evaluated, and the results show that this technique provides a very good accuracy. The technique is also quite portable, as all the details related to the emulation of software faults in different environments are encapsulated in fault emulation operator libraries.

1 Introduction

Fault injection has been extensively used in the last two decades to evaluate specific fault tolerance mechanisms and to assess the impact of faults in systems. A major issue in fault injection is to assure that the injected faults are representative of actual faults, as this is a necessary condition to obtain meaningful results. It is widely accepted that existing fault injection technologies can emulate hardware faults, either transient or permanent. However, and in spite of the progress made in recent years, the emulation of software faults (i.e., program defects or bugs) by fault injection is still a rather obscure step.

The interest of extending the fault injection technologies to the emulation of software faults, in order to assess the impact of residual bugs or validate software fault tolerance mechanisms, is potentially very large. In fact, several studies show a clear predominance of software faults [1, 2, 3, 4] as the root cause of computer failures and, given the huge complexity of today's software, the weight of software faults on overall system dependability will tend to increase.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 137–159, 2003.

The complete elimination of software defects during software development process is very difficult to attain in practice. In addition to well-known technical difficulties of the software development and testing process [5, 6], practical constraints such as the intense pressure to shrink time-to-market and cost of software contribute to the difficulties in assuring 100% defect free software. Therefore, the current scenario in the computer industry is having systems in which software defects do exist but no one knows exactly where they are, when they will reveal themselves, and, above all, the possible consequences of the activation of the software faults. In a world where components of the shelf (COTS) are used more and more to build larger systems, the residual software faults represent a growing risk.

Many software reliability models and measurement procedures have been proposed for the prediction and estimation of measures of quality such as the number of faults remaining in a given software package. However, in addition to the difficulties in handling the extreme complexity of today's software and the limitations of the necessarily simple models with few parameters actually related to the project or software package at hand, the problem of these software quality measures is that they are mainly developer-oriented and do not give a measure of the possible impact of residual faults on the operation and on the end-user.

The use of fault injection to emulate the effects of real software faults has been recognized as potentially very useful. We identify three major goals for the emulation of software faults:

- **Prediction of worst case scenarios and risk assessment.** The emulation of software faults can be used as a way to quantify the impact of software faults from the user point of view and get a quantitative idea of the potential risk represented by residual faults [7].
- **Validation of fault tolerant mechanisms.** As typical software faults left in deployed systems are rarely triggered, the validation of fault tolerant mechanisms can be achieved by accelerating the software fault activation through the emulation of software faults [8].
- **Dependability benchmarking.** One recent research effort is centered on the definition of dependability benchmarks [9, 10]. Techniques to emulate software faults are essential for the benchmark faultload.

The key issues surrounding the injection of software faults are the accuracy of the emulation of the faults (i.e., the fault representativeness) and the definition and validation of proper operating scenarios/tools that allow the use of this technology to attain the three major goals mentioned above. Ideally, the injected faults should precisely emulate the hidden defects of a program, which is obviously not possible because these defects are not known in advance (if we knew the bugs we would fix them beforehand). Given this impossibility, the emulation of software faults by fault injection can be achieved in practice by attaining the following goals:

- The first goal is to inject faults that generate errors or induce erroneous program behavior similar to the ones caused by real software faults. That is, what is im-

portant is to avoid the injection of faults that cause errors that cannot be generated by a software fault (for example, errors that could only be generated by hardware faults), as in this way the injected fault do not emulate a software fault.

- A second, and more refined objective, is to inject faults that emulate classes of software faults.

It is worth noting that the emulation of physical faults, as presented in numerous research works in the literature, also suffers a similar limitation. In fact, the ideal situation concerning physical faults would also be to inject the faults that emulate the real physical faults that the system will eventually have. As this is not possible, researchers do inject faults that emulate physical faults in general. In certain circumstances, it is possible to know that some kinds of physical faults will be prevalent. For example, in a space born computer single event upsets (SEU) is a quite probable type of faults. In this case, the fault injection tool should be chosen or tuned to inject this kind of faults.

In the case of software fault emulation, once the process of generating and injecting software faults is dominated (first bullet above), it will be possible to tune the process to generate specific classes of software faults most likely to be experienced in the field by a particular program. This identification of the most likely classes of software faults could be achieved by using information about the development process, the team that developed the software, specific metrics of the code (code size, number of variables, complexity of data structures, etc), or field data about previous faults discovered in the program.

In spite of the above similarities between the physical and software fault emulation, the problem of emulating software faults is far more difficult and it is still an open question. In fact, while the error models corresponding to physical faults in an integrated circuit are relatively easy to understand (bit flip, stuck-at-one, stuck-at-zero, etc), software faults are by nature human made faults (at the design, implementation, etc) and it is extremely difficult to model human errors.

This paper addresses the two key issues already mentioned (fault **representativeness** and the definition/validation of **operating scenarios/tools** for the injection of software faults) in the following way:

1. Investigates the possibility of accurate emulation of real software faults by state-of-the-art SWIFI (Software Implemented Fault Injection) tools. To do this, we use the Orthogonal Defect Classification (ODC) [11] to classify a set of real faults found in programs and then compare the effects of the activation of a given fault with the effects of the emulation of the same fault by the Xception tool [12]. The results are discussed and generalized in order to identify:
    a) the types of software faults that can be emulated by a typical SWIFI tool;
    b) the new features that would have to be added to a typical SWIFI tool to extend it to the emulation of additional types of software faults;

c) the types of software faults that cannot be emulated by any SWIFI tool.

This discussion highlights, as a first conclusion, that the use of SWIFI tools, even with extended features, is probably not the best way to emulate software faults, and a different approach is required to cope with this problem.

2. Presents the technique G-SWFIT (Generic Software Fault Injection Technique), which is our proposal for a practical approach to inject software faults. This technique consists of finding key programming structures at the machine code-level where high-level software faults can be emulated. In this way, it is possible to have a library of machine-code level structures and possible software faults (a low-level mutation, in practice) that, once introduced in such programming structures, can emulate specific classes of high-level software faults. The fact that G-SWFIT works at the machine code-level is essential in a fault injection context, as the source code of the system under evaluation is not generally available.
3. Evaluates the accuracy of G-SWFIT by comparing the effects of the faults introduced by this technique with the corresponding high-level faults that are intended to be emulated. In general, we have observed a pretty good match between the injected faults and the actual faults that are supposed to be emulated.
4. Discusses the generalization and portability of the proposed method according to multiple aspects, such as the high-level language used in the construction of the target system, the compilers, the compilers optimization options, and the processor architecture. A very important conclusion is that G-SWFIT is practically independent from the high-level language and compiler details, which is a necessary condition to use the technique in systems where the source code is not available.
5. Finally, the paper presents the proposed operating scenario for the use of G-SWFIT in fault injection experiments.

The structure of the paper is as follows: next section presents the related research; section 3 investigates the possibilities of the emulation of software faults by SWIFI tools. Section 4 presents the G-SWFIT technique and discussed its accuracy and portability. Section 5 presents some final conclusions.

2 Related Work

The study of software faults has been mainly related to the software development phase, as the software faults are originated during the different steps of this phase (requirements, specification, design, coding, testing, etc). This is an important area of software engineering and many studies have contributed to the improvement of the software development methodologies, with particular emphasis on software testing, software reliability modeling and software reliability risk analysis [5, 6].

The fault history during the development phase, the operational profile, and other process measures have also been used in software reliability models to estimate the reliability of software and to predict software faults for risk assessment [5, 13, 14, 15, 16].

The study of the software reliability during the operational phase (i.e., after the product deployment) is substantially different from the software under development. The operational environment and the software maturity are different during the operational phase, and the software reliability should be studied in the context of the whole system (and not just in the context of a given application). The difficulties are not only in the instrumentation required to collect data on software faults but also in the fact that the software faults must be analyzed taking into account the system architecture (hardware and software) and not only software modules. Maybe these difficulties account for the fact that the number of works on software faults during the operational phase is lower than the studies available for the development phase. Nevertheless, the study of the effects of actual software faults in the field is of utmost importance for our work, as this is just the kind of faults we want to emulate by fault injection.

Two significant studies of software dependability in Tandem systems are presented in [4, 2]. The impact of software defects on the availability of a large IBM system is presented in [17]. An early study [18] investigated the effect of the workload on the reliability of an IBM operating system based on data collected from field.

An important contribution to promote the collection and study of observed faults is the Orthogonal Defect Classification (ODC) [11]. ODC is a classification schema for software faults (i.e., defects) in which defects are classified into non-overlapping attributes and used as a source of information to understand and improve the software product and the software development process. Although ODC is intended to provide feedback on to the development process, it also provides a useful defect classification concerning the problem of the emulation of software faults by fault injection.

Concerning the area of fault injection, many different approaches have been proposed in the literature. Generally they can either be based on specific hardware, system simulation, or software. Hardware techniques inject physical faults in the target system hardware. Simulation techniques make use of a simulation model of the target system. Finally, a third solution is to emulate hardware faults and errors through software (Software Implemented Fault Injection, or SWIFI for short).

The latter technique (SWIFI) has become very popular as it has low complexity and requires low development effort when compared to the other fault injection approaches. The basic idea of SWIFI consists of interrupting the application under execution in some way (usually by inserting a software trap or executing the application in trace mode) and executing specific fault injection code that emulates (hardware) faults by inserting errors in different parts of the system such as the processor registers, the memory, or the application code. Some examples of SWIFI tools are FERRARI [19], FTAPE [20], Xception [12], and Goofi [21]. The Xception approach for SWIFI becomes the standard way to inject faults and most of the SWIFI tools

available today use the processor breakpoint register and low-level exception mechanisms to inject faults.

Only very few studies have addressed the problem of the injection of software faults [22, 23]. In spite of the fact that the issue of how accurately the injected faults emulate real-world software faults remains largely unknown, fault injection has been used with success in several research works where software faults are the most relevant class of faults. Several examples of software weaknesses revealed by faults injected at random can be found in [7].

Mutation testing is a specific form of fault injection that consists of creating different versions of a program (mutants) by making small syntactic changes [24, 25]. Mutation can be considered as a static fault injection technique, as the source code is changed instead of the program/system's state, as happens in classical fault injection (considered dynamic fault injection in this view). Mutation has been largely used for software testing to identify the best sets of test cases or to study the error propagation process from fault activation to an eventual wrong program output.

An experimental comparison of the errors and failure modes generated by actual software faults and mutations is presented in [26]. The results favor the idea that mutations produce error patterns and erroneous program behaviors similar to actual software faults. In the experiments presented in that paper, 85% of the errors generated by mutations have also been produced by real faults. Clearly, this doesn't prove that the injection of faults (by a SWIFI tool) causing the same error patterns as mutations also represent real software faults. Nevertheless, it suggests that the injection of faults that cause the same classes of errors as mutations must be investigated as a potentially good fault model for the emulation of software faults.

In [7] fault injection is proposed for the quantification of the risks created by the software component of a system (experimental software risk assessment). Given the difficulties of knowing what kinds of software faults are most likely to be hidden in the code and their probability of future manifestation, the results of fault injection cannot be used as an absolute measure of risk. Instead, the authors suggest the use of fault injection for the prediction of worst case scenarios (in terms of software risks).

In [8] random code corruption and faults meant to emulate specific programming errors were used to improve the design of a reliable write-back file cache. This paper is a quite convincing example of how SWIFI tools can be used to improve fault tolerance techniques during the design phase.

The problem of the accurate emulation of software faults by fault injection was first addressed by Christmansson and Chilarregue [27, 28]. These studies propose the same basic set of rules for the generation of errors that emulate software faults. These rules are obtained from the analysis of field data about discovered software faults that have been classified using ODC. In the first paper [27] the rules for error generation for fault injection are proposed for fault forecast and the second paper [28] addresses the problem of error generation for fault removal.

Christmansson and Chilarregue papers are seminal papers in this topic, as they give a first contribution to the solution of the problem of emulation of software faults by fault injection. Nevertheless, they also raised the new questions we are addressing in the present paper, particularly the questions on the accuracy of the emulation and on the adequacy of existing SWIFI tools to perform the injection of errors according to the rules proposed by Christmansson and Chilarregue.

The work presented in this invited paper covers several results of our research at the University of Coimbra, which have been presented in successive papers in recent years [29, 30, 31, 32].

3 Software Fault Emulation Using SWIFI Tools

3.1 Problem Statement

The definition of software faults requires the notion of correctness of software. In a broad view, the correctness of software should be measured taking into account the end user/customer needs. However, the needs and degree of satisfaction of the user are too vague to be useful for our purposes. For the purpose of this work, it is assumed that the requirements and specification are correct. Thus, a software fault means that the code is not correct somehow (i.e., it does not implement the specification in some particular aspect). More specifically, the software faults that we would like to emulate by fault injection are the faults originated during the coding phase that have not been detected by the testing procedures and, consequently, go with the deployed product.

A software fault can be characterized by the change in the code that is necessary to introduce to correct it (i.e., to put the code consistent with the specification, which is assumed to be correct in our case). This is just the notion of defect proposed in ODC [33]. In ODC a trigger and a type characterize a defect (i.e., software fault). The trigger describes the general conditions that make the fault to be exposed and the type represents the fault in the source code. The following fault types in ODC are directly related to the code (thus describe software faults as defined in the context of the present work):

Assignment – value(s) assigned incorrectly or not assigned at all;

Checking – missing or incorrect validation of data or incorrect loop or conditional statements;

Interface – errors in the interaction among components, modules, device drivers, call statements, or parameter lists;

Timing/serialization – missing or incorrect serialization of shared resources;

Algorithm – incorrect or missing implementation that can be fixed by (re)implementing an algorithm or data structure without the need for requesting a design change;

Function – incorrect or missing implementation of a capability that affects a substantial amount of code and requires a formal design change to be corrected.

The classes of triggers defined in ODC are associated to common activities of the development process: review/inspection, function test, and system test. Only the system test class of triggers is relevant in our case, as it represents the broad environmental conditions when the faults are exposed during the operational use in the field. These general conditions (triggers) are startup/restart, workload volume/stress, recovery/exception, hardware/software configuration, and normal mode. The normal mode category means that the software fault has been exposed when everything was supposed to work normally (nothing unusual has occurred in the system). This is the trigger category relevant for our study, as all the experiments have been done with the target system working in normal conditions.

The fault trigger classes defined in ODC reflect the main purpose of this technology, which is to help the improvement of the software development process. In fact, ODC trigger classes are the very general environmental conditions that work as a catalyst in the process of activating a dormant software fault in such a way that it causes a failure. These trigger classes (startup, recovery, workload volume, etc) are useful when the objective is to improve the software development process but are of little use in our case, as the emulation of specific software faults requires the definition of much more precise triggers.

As mentioned above, a software fault is characterized by the necessary change in the code to correct it. It means that the fault type and fault trigger are defined by the programmer according to the change he/she made in the source code. As the source code is normally written in a high-level language and the typical SWIFI tools inject faults at the machine code-level of the target processor, it means that the fault classification and the fault emulation by fault injection are done at different abstraction levels (see Fig. 1).

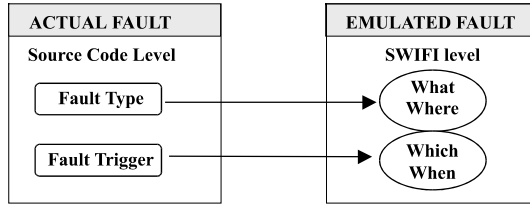


Fig. 1. Fault classification at source code level and fault emulation at SWIFI level.

In a typical SWIFI tool injected faults are defined according to three main classes of parameters: *what* (what should be changed/corrupted), *where* (where, in the code, should the change be applied), *when* (when, during the program execution, should the change be inserted). The traditional *when* parameter should, in our opinion, be de-

composed in *which* (which instruction or event acts as fault trigger) and *when* (when, during the various executions of the trigger instruction or trigger event is the fault injected).

While ODC fault types have a relatively clear translation into the *what* and *where* fault injection parameters (it is a matter of translating high-level changes into machine level), the ODC fault triggers cannot be used to define the fault triggers (the *which* and *when* parameters) because ODC triggers such as startup, recovery, normal mode, etc are too vague to define the actual circumstances in which the fault should be injected.

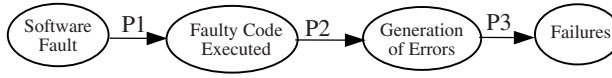


Fig. 2. Probability of a software fault exposure. The probability of a software fault resulting into a failure (i.e., the probability of a software fault being exposed) is heavily dependent on the operational profile. Assuming a fault exists (is dormant), the probability of the faulty code to be executed is p1 (see Figure 2). If the faulty code is executed, the probability of error generation is p2. If errors are generated, the probability of these errors resulting into a failure is p3. Thus, the probability of a software fault resulting into a failure is the product of p1, p2, and p3.

Ideally, the fault trigger should reproduce the chain reaction described in Fig. 2. However, the need of accelerating the process suggests that errors should be injected instead of faults (i.e., making p1 and p2 equal to one) which leads us to the paramount question of the representativeness of the injected errors.

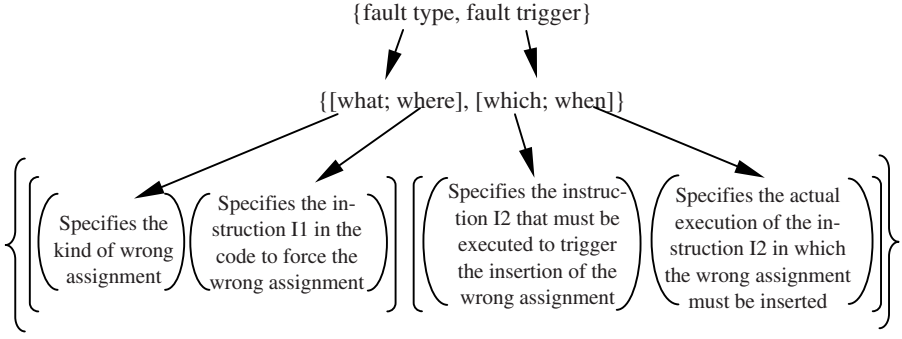


Fig. 3. Assignment fault described in terms of the parameters what, where, which and when. The instructions I1 and I2 can be the same and these instructions are defined at the abstraction level of the SWIFI tool, which is normally the machine code-level of the target processor.

The error representativeness can be regarded in two different perspectives:

- *Representativeness concerning fault type:* The injected errors are considered software errors if they could have been caused by a real software fault of any type (ideally, it should be possible to generate errors that emulate specific fault

types such as assignment, checking, etc). That is, what is required is to avoid the injection of errors specific of other kinds of faults such as hardware faults.

- *Representativeness concerning fault trigger:* The injected errors should emulate the errors that would have been caused if the faulty code (corresponding to a software fault) had been executed with the input data required to generate errors. As the software faults are program defects that escaped the testing procedures the probability of one of these faults being triggered is normally very low and is heavily dependent on the input data. That is why it is important to accelerate software error generation process (making p1 and p2 equal to one in Fig. 2), but that should be done in such a way that the injected errors are representative of the errors generated by real software triggers.

The idea is then to check if it is possible to emulate software faults by injecting errors (using SWIFI tools) defined in terms of the parameters that describe the *what*, *where*, *which*, and *when* attributes. The exact parameters depend on the SWIFI tool and the target system. As a general indication, the fault type is described by the attributes *what* and *where* and the fault trigger is described by *which* and *when*. Fig. 3 shows an example of these parameters for a wrong assignment fault.

3.2 Experimental Evaluation

The Xception fault injection tool. The Xception is described in detail in [12]. However, a very brief description is provided here to facilitate the discussion on the software fault emulation in the next sections. Xception uses the debugging and performance monitoring features existing in most of the modern processors to inject faults by software and to monitor the activation of the faults and their impact on the target system behavior. Faults are injected with minimum interference for the target application. The target application is not modified, no software traps are inserted, and it is not necessary to execute the target application in trace mode (the application is executed at full speed).

Xception provides a comprehensive set of fault triggers, including spatial and temporal fault triggers, and triggers related to the manipulation of data in memory. Faults injected by Xception can affect any process running on the target system (including the kernel), and it is possible to inject faults in applications for which the source code is not available. Table 1 shows the target fault locations and basic fault triggers and fault/error types of Xception.

The version of Xception used is the XceptionPPC, targeted for the PowerPC 601 processor. The target system is a Parsytec PowerXplorer box with four PowerPC 601 processors running under the Parix operating system (a Unix like operating system for Parsytec parallel machines) and the host is a Sun/Solaris machine.

The real software faults used. In order to evaluate the possibilities of a SWIFI tool (the Xception) to emulate real software faults we need to have access to programs with known real software faults. The programs resulting from the International Olympiads in Informatics (IOI) from ACM International Collegiate Programming Contest [34] and a program actually used in real life provide an easy source of "real" software faults. It is obvious that the programming contest surroundings are different from a typical software development environment. However, a programming contest like IOI has several factors that make it an interesting source of programs for the present research. In short, this idea is supported by the following arguments:

- All the programs are written according to a formal and correct specification;
- The programs were written by very skilled programmers;
- The contest gives us rapid access to several implementations from the same specification (the number of teams presented in the IOI 98 contest was 237, which results in a large number of alternative implementations of each program). These alternative programs normally use different design strategies and algorithms, which makes it possible to compare the influence of different program control and data structures in the failure modes caused by the software faults;
- There is a test case associated to each problem specification, which works as an acceptance criteria for correct programs from the contest judges' point of view. Only bugs found in these "correct" programs (i.e., programs passed in the test cases) are considered as representative of real software faults.

The following programs were used as source of bugs:

Camelot – This program computes the minimum number of moves required to gather all the pieces of a chessboard in the same square. Only two kinds of pieces are considered: one king and a variable number of knights ranging from 0 to 63 knights. The size of the different available versions of this program (made by the different teams) ranges from 200 to 360 lines of code.

JamesB – This program codifies strings according to a specific algorithm. A seed received as a parameter with each string determines the actual codification. The result is the coded version of the original string. The size of the available alternative versions of this program is about 100 lines of code.

The search for software faults in these programs was done in the following way:

1. The programs approved by the board of judges test case were selected (these programs were considered totally correct for the contest purposes);
2. Next, these programs were intensively tested by using a very thorough test case. This was achieved in practice by running the programs a huge number of times with random input data sets. These tests may run for hours. Programs failing this intensive test have software faults;
3. Programs with software faults were then analyzed in detail to identify the fault. The input data that exposed the fault was used to help the bug identification.

Seven software faults have been identified in seven different programs (i.e., each program contains only one detected software fault). Five software faults were in the implementations of Camelot and two faults in the JamesB. The reason why the number of software faults found in the JamesB is smaller than the number of faults found in the Camelot is due to the fact that the JamesB problem is simpler that the Camelot, which makes it easier to write correct implementations of JamesB.

The software faults have been analyzed and classified according to the following fault types (to simplify the identification each program is named as Camelot.team# and JamesB.team#):

- Assignment: 2 faults (JamesB.team6, Camelot.team4);
- Checking: 1 fault (Camelot.team1);
- Algorithm: 4 faults (JamesB.team7, Camelot.team2, Camelot.team3, Camelot.team5).

Emulation of the real software faults by the Xception tool. To evaluate the possibility of accurate emulation of the actual software faults by using the Xception tool each fault was analyzed in order to determine the adequate Xception fault trigger and fault models. For the algorithm faults we quickly concluded (as expected) that the accurate emulation by the Xception (or any other machine code-level SWIFI tool) is simply not possible (this will be discussed further on). However, assignment and checking faults could in general be emulated. In general, one fault can be emulated in several ways, using the different possibilities of fault trigger and fault models provided by the Xception.

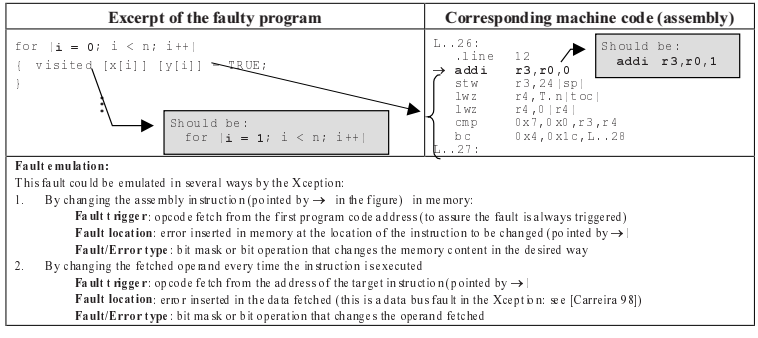


Fig. 4. Example of an assignment fault in the program Camelot.team4.

To compare the fault emulation done by Xception with the actual faults, the correct version of each program (i.e. the version obtained after removing the bug) was executed with the same test case (i.e., same set of data inputs) used in the faulty version. However, in each run the fault was injected by Xception to emulate the effects of the bug. If the results (i.e., impact of the fault in the form of failure modes) are the same in both runs it means Xception do emulate the fault accurately.

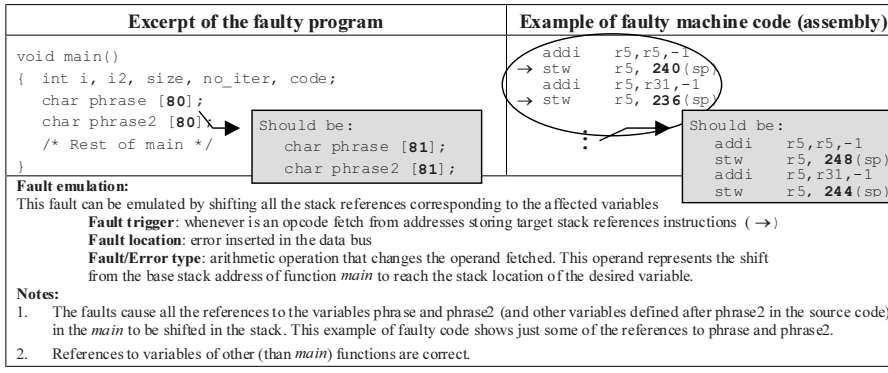


Fig. 5. Example of an assignment fault in the program Camelot.team6.

For a better understanding of the way the software faults have (or have not) been emulated let us examine some of these faults in detail (see figures 4, 5, 6, and 7).

One important aspect is that the emulation of specific faults by the Xception requires manual intervention for determining specific locations in memory to set fault triggers or to insert errors. In our experimental environment, the loader provides this information. However, in a real system this could be difficult to obtain (but it is generally not necessary as the fault injection campaigns are typically done in a statistical way and not with the goal of reproducing specific faults). Another manual task is the definition of the right mask and bit level operation to insert the desired error.

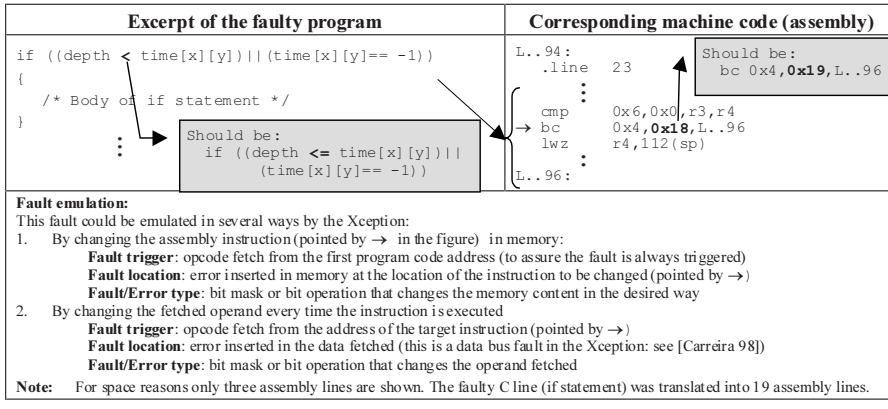


Fig. 6. Example of a checking fault in the program C.team1.

The Xception could not entirely emulate the assignment fault shown in Fig. 5. The reason is in the fact that the fault trigger used (opcode fetch from a specified address) is implemented by using the processor breakpoint registers, which are only two in the PowerPC. Using the traditional SWIFI approach of inserting trap instructions to trigger the faults could solve this, but this technique is much more intrusive. Another relevant aspect concerning the emulation of this fault is that it requires large manual

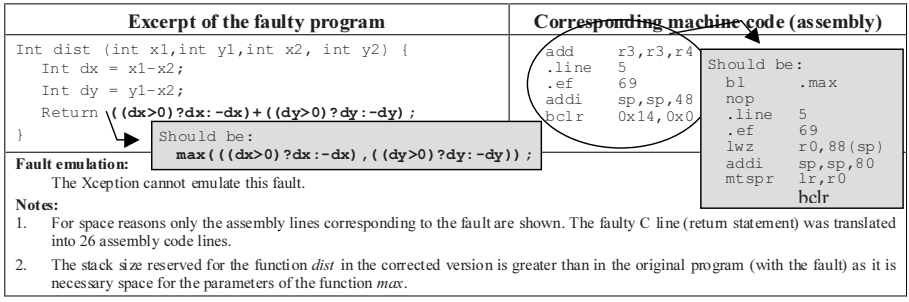


Fig. 7. Example of an algorithm fault in the program C.team5

intervention. Definitely, extra software tools to assist the definition of assignment faults that cause shifts in the stack are required to make the process usable.

Fig. 7 shows an example of an algorithm fault. In general algorithm faults cause considerable changes at the machine code-level and cannot be emulated by SWIFI tools (at least using low intrusive ways). In some cases, algorithm faults can be decomposed in assignment and/or checking faults. For example, the fault in Fig. 7 corresponds to an incorrect assignment to the return parameter of function dist (it is assigned the sum of two values instead of the larger value). However, the emulation of algorithm faults by "equivalent" assignment/checking faults is very doubtful.

The ODC types function, interface, and timing/serialization were not analyzed, as we didn't find software faults of these types in the used programs. However, considering the definitions of these types of faults we would say that function faults suffer the same problems as algorithm faults and cannot be accurately emulated by SWIFI tools. Interface faults are somehow similar to assignment faults (the wrong assignments are in the modules and function interface) and some of them can be emulated. The accurate emulation of timing/serialization faults is heavily dependent on the specific fault.

The results of this experiment can be summarized as follows:

1. Most assignment and checking faults can be accurately emulated by the Xception.
2. The present version of Xception cannot emulate some assignment faults or the emulation requires high manual intervention. However, it is possible to define new Xception features to facilitate the accurate emulating of this set of faults.
3. A third set of faults cannot be emulated by the Xception and their emulation by SWIFI tools does not seem feasible. Algorithm faults and function faults fall in this category. The ODC types interface and timing/serialization are most probably also in this category. This set of faults represents the limits of the Xception model (and SWIFI tools models) in emulating real software faults.

Considering the field data results published in [27, 32] algorithm and function faults account for more than 40% of the software faults. If we add to this group the ODC types interface, and timing/serialization the obvious conclusion is that a SWIFI tool like Xception can only emulate accurately less than 50% of software faults. Additionally, the tools would have to be extended with new features to automate the definition of the fault triggers and fault types for software faults (the task that has been done in a manual way in these experiments). A possible solution to this problem is proposed in [30].

4 Software Fault Emulation by Low-Level Mutations

The logical conclusion of the previous section is that SWIFI tools are not the best choice to emulate software faults, which has driven our quest for alternative approaches. G-SWFIT (Generic Software Fault Injection Technique) is, to the best of our knowledge, the first technique proposed for the injection of software faults.

When compared to SWIFI tools, G-SWFIT follows a completely different approach. It works at the object code level (i.e., executable code) and it consists of finding key programming structures (code patterns) at the machine code-level where high-level software faults can be emulated. As soon as these programming structures are identified the injection of the faults consists of applying mutated pattern directly in the object code. To make the technique practical, G-SWFIT is heavily based on a library of machine-code level structures (or patterns) and possible software faults that, once introduced in such programming structures, can emulate specific classes of high-level software faults.

This section describes G-SWFIT and shows that the accuracy of the injected faults (when compared to the high level faults) is very high. Additionally, the (apparent) dependencies of this technique on high-level language used to develop the target code and the compiler, assembler, etc is also discussed and we conclude that G-SWFIT is pretty general and can be easily ported to practically all types of systems.

One important aspect is that as G-SWFIT works at the machine-code level it does not require the source code of the target program, which makes it possible to apply G-SWFIT to virtually any program.

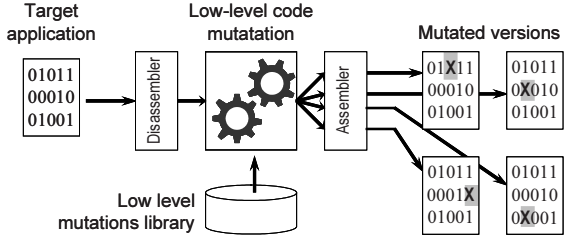


Fig. 8. The automated low-level fault emulation in G-SWFIT consists of modifying the ready-to-run binary code of software modules by introducing specific changes that correspond to the code that would be generated by the compiler if the software fault were in the high-level source code. A library of mutations previously defined for the target platform guides the injection of code changes: the target application code is scanned for specific low-level instruction patterns and mutations are performed on those patterns to emulate related high-level faults.

The key idea of G-SWFIT is represented in Fig. 8. The mutation tool includes the functionality of a disassembler to translate the executable file into assembly code where the machine-code patterns are scanned. The mutations are inserted directly in the executable file (i.e., disassembler is just to facilitate the patterns scanning).

The definition of the low-level mutation library is based on two principles: the existence of a set of simple high-level programming errors that occur frequently, and the knowledge on how high-level languages are translated into low-level code.

Many of the defects that remain in the software after deployment are usually simple programming errors [32] (at least if analyzed independently from their context) or wrong usage of language constructs: their complexity arises from the code that contains them [26, 29, 32]. Such defects are classifiable into a few high-level fault classes, allowing for a definition of a set of common errors. In order to accurately inject faults at machine language level that represent real high-level software defects, it is necessary to get detailed knowledge about the high-level constructs they correspond to.

Fig. 9 describes the steps needed to build the G-SWFIT library. One important aspect related to the representativeness of the faults injected this way is the choice of the high-level software faults that are described in the G-SWFIT library. We analyzed several sources (detailed further on) and conducted and extensive field study [32] in order to build the list of bugs that can reasonably be expected to occur frequently. We call these faults "educated mutations" to emphasize that they include inputs from experience and field data on real software faults.

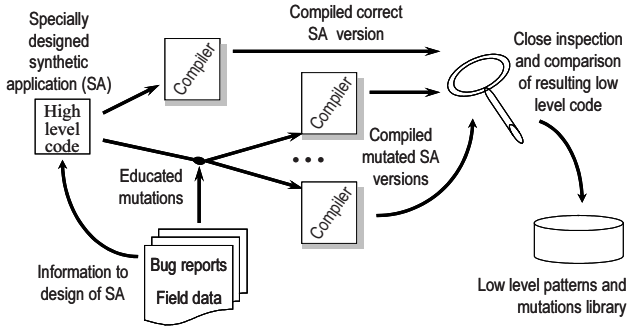


Fig. 9. To assist definition of the fault library we used a synthetic application containing all the pertinent key constructs and structures, both with and without the considered high-level faults. The observation of the generated code for both cases (with and without faults) allows us to identify the specific machine code patterns where a given class of faults should be emulated by low-level mutations. Using this application with several compilers and different optimization settings we could understand how such variations affect low-level code generation and identify patterns for a given class of faults.

4.1 Feasibility and Accuracy

The C programming language was selected for the case study to evaluate both the feasibility and accuracy of the emulated faults. However, the methodology described

here is in no manner tied to the language itself, and can be easily ported to other languages (this issue is discussed in section 4.2).

The application of G-SWFIT was carried out in three steps:

1. The set of high-level language software faults are identified. This step also comprises the elaboration of high-level faulty versions of the target applications through educated mutations.
2. The low-level instruction patterns where mutations can be inserted for emulation of the high-level faults are defined. The related mutations are also defined in this step.
3. Fault emulation is performed through the injection of mutations defined in the prior step into the low-level code. A given number (from tens to thousands) of mutants are generated from the original executable version. The execution of each mutant with a representative input profile is a fault injection experiment in our context.

The characteristics of the C programming language were analyzed to obtain a set of errors that are prone to be made in this language, obtaining a high-level fault characterization related to ODC but having a more detailed characterization in what concerns its relationship with the syntactic rules of the language behind the faults. An extensive research on most common C programming bugs has been carried out, using various information sources ranging from programming manuals and best practice tutorials [35] to field data [32].

The identification of such faults was additionally guided by the following rules:

1. Faults must be relatively common (i.e., likely to appear in available software products).
2. Faults must not generate syntactic errors.
3. Faults must not be too obvious.

The considered high-level faults were characterized according to the following items:

- Applicable ODC class: the characteristic relates the mentioned errors with the ODC classes. As it was explained above, some errors fit in more than one OCD class, depending on the way that those errors would be corrected.
- Example of possible cause: this helps to understand how the error may appear in the source code. This is a fundamental characteristic of the considered errors, as this work analyses mainly how and why faults appear rather than how they can be corrected.
- Compiler ability to detect the error: the ability of available (common) compilers to detect and warn the programmer of the possible existence of that error. For this evaluation, three different compilers were observed (VC++, BorlandC and GCC). This characteristic is important as it is directly related to the probability of the programmer leaving that error uncorrected in the program.
- Language specific degree: this characteristic relates each particular error with the possibility of that error appearing in all common languages, or instead, if it is something specific to one or few languages. This gives some insight for future expansion of this work to encompass other programming languages.

The resulting set of high-level common errors in C programming language is presented in Table 1. The detailed description of the fault emulation operators is discussed in [31].

Table 1. Xception fault locations, fault triggers, and fault/error types.

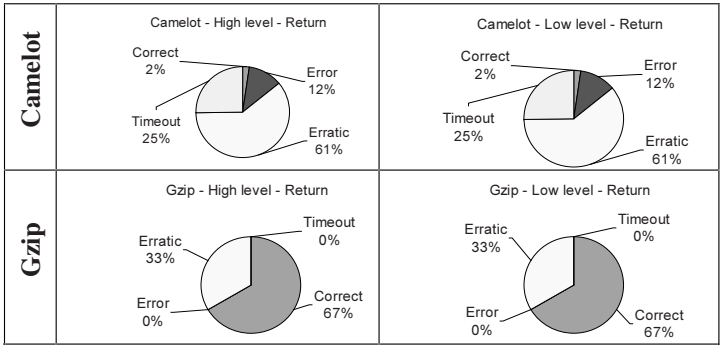
ODC	Fault type description	Example of possible causes	Educated mutations examples
Assign.	Missing or wrong local variable initialization	Initialization dependent of execution path	Omission of first assignment to local variable
Check.	Assign. instead of comparison	*If (a = b)* instead of *if (a = = b)*	Change "==" to assignment
	Miss by one in decisions	Lack of attention;	If (a < b) instead of if (a <= b)
	One iter. too few or too many	Misunderstanding of variable meaning	while (a<=b) instead while (a<b)
Interf.	Missing return statement	Non-trivial execution paths inside functions	Omit return statement were syntactically possible
	Wrong return statement	*return var* with *var* not properly set	Omit previous assignment to variable used with return.
	Wrong usage of parameters in function call	Func(x,y) instead of func(y,x) with x and y of compatible data types	Change order of values in the func. call (compatible data type)
Algor.	Miss aligned else	Bad indenting plus single statements mixed with multiple statement	Omission of curled brackets inside complex if-else chains
	Binary operator used instead logical operator	Typo; Lack of attention	Changing *a && b* to *a & b*
	Wrong logical expr. by wrong usage of op. precedence	Usage of complex expressions such as (a == b && c == d || d == e)	Insertion, omission or change of parenthesis in complex express.
	Missing statement	Comment ends one line too late.	Omit statement after a comment
	Missing function call	Lack of attention	Removal of the function call if it is not part of a bigger statement

To evaluate the accuracy of the emulation of the high-level faults we compared the effects (program failure modes) of high-level educated mutants with the ones obtained with low-level mutations inserted using G-SWFIT.

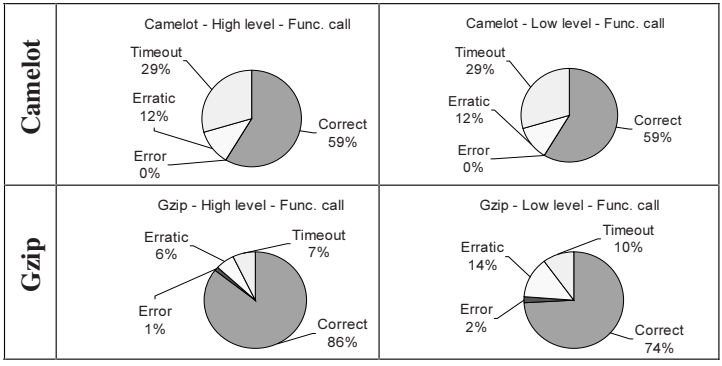
The correctness of the outputs generated in each run was evaluated using a database of correct results (obtained by running each application with no faults for all the input vectors). In addition to the correctness of the results, other aspects of the application behavior have been used, leading to the following failure modes:

- Correct behavior ("***correct***"): the application produced the expected result in the allotted time and did not cause any abnormal event during its execution. Either the injected fault was not activated or it was tolerated by the inherent program redundancy. To minimize the possibility of the injected fault not being activated, a large number of different inputs were used with each mutated version.
- Uneventful execution but with incorrect results ("***error***"): the application terminated but the results were incorrect.
- Erratic behavior ("***erratic***"): the application behaved in an unpredicted manner (for instance, the application produced an incoherent error message) and without valid feedback to enable the determination of the success of the requested task.
- The application hanged ("***timeout***"): the situation is detected by allowing a *more-than-enough* time interval for its completion. If the application does not terminate within that interval, then it is assumed that it hanged and is terminated by an automated tool.

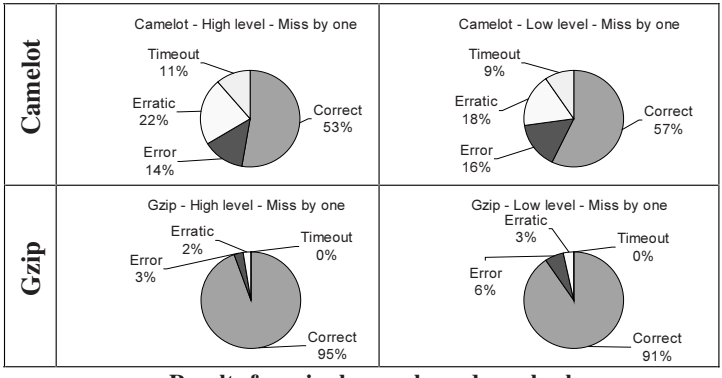
The applications used in these experiments are the Camelot (already described) and the widely used Gzip application (compression tool). The experiments have been performed in a Intel Pentium III machine running Windows2000.



Results for missing or wrong return statement.



Results for missing function call.



Results for miss-by-one boundary check.

Fig. 10. Comparison of the effects of high-level faults and corresponding G-SWFIT faults.

4.2 Generalization of G-SWFIT

In this section we investigate how the factors such as the high-level language, the compiler, the compiler optimization settings, and the target processor architecture may or may not influence the library of low-level instruction patterns and the corresponding faults. Obviously, once we have a library of faults, the actual use of G-SWFIT is straightforward, as what is needed is to find the pattern and insert the mutations, which is largely independent from the setup details.

The method consists of compiling the synthetic application (the same mentioned in Fig. 9) using different optimization settings, different compilers, target architectures, and a different language and analyze the resulting low-level code to check if the instruction patterns initially identified were still present or if new patterns have to be added to the library. The following paragraphs discuss these results.

Different compilers and optimization settings. Many different settings for a variety of compilers (Visual C++, Borland C++, GNU C++, Turbo C++) were evaluated. It has been observed that different patterns are generated for a few high level constructs, depending on the optimizing settings used. However, the only consequence of this is that we have to add some new patterns to the library in order to assure that the library covers the code generated from different compilers and with different settings.

Different languages. Both C++ and Pascal languages have been used. The C++ generates essentially the same low-level code as C, when the same kind of high-level constructs is used. Although object-oriented constructs were not directly compared, no ambiguity with the existing patterns was detected. To encompass the new high-level faults resulting from the expanded syntax of C++, it is only necessary to augment the library of patterns and mutations. In the case of Pascal, the resulting low-level code is essentially the same as the resulting from C language. Some minor differences do exist, for instance in the way that parameters are passed to functions. However, no interference was noted concerning the patterns used in the case-study.

Different target processor architectures. The following compiler/platforms were used: GCC/Linux on an IA32 machine and GCC/OSF Unix for Alpha AXP. In the GCC for Linux over IA32 case, the code output is essentially the same as compilers for Windows over IA32, including the variations caused by different optimization settings. This was somewhat expected since the type of the underlying machine is the same. Concerning the GCC for OSF Unix over Alpha AXP, the code is totally different. This was expected since there are very few similarities between an Intel 80x86 and an Alpha AXP processor. Therefore, low-level instructions are also different, which means that each processor requires a specific library of patterns and corresponding mutations.

Generalization discussion. As a conclusion, the generation of the library of faults (and G-SWFIT itself) is mainly dependent on the target processor architecture. This means that we need as many libraries of faults as architectures we want to cover. All

the other factors evaluated have also some influence on the libraries of faults, but this influence consists of having a more or less complete library. That is, the more compilers and optimization settings used, the more complete the library is.

5 Conclusions

This paper discusses the problem of the emulation of software faults by fault injection. A set of real software faults found in different programs has been compared with faults injected by the Xception to evaluate the accuracy of the injected faults. The results of this experiment revealed three different kinds of software faults: i) faults that can be accurately emulated by the Xception; ii) faults that could be emulated if Xception was improved with extra fault triggers, fault models, and tools to avoid the need of manual fault definition iii) faults that could never be emulated by Xception (or any other SWIFI tool). The logical conclusion of this first study is that SWIFI tools are not the best choice to emulate software faults, which has driven our quest for alternative approaches.

G-SWFIT is a fault injection technique specifically conceived for the emulation of software faults. The basic idea of G-SWFIT is to find suitable locations for the accurate emulation of specific high-level languages programming errors that are usually responsible for common software faults. The key aspects of G-SWFIT are a library of low-level instructions patterns and mutations that relate to specific high-level faults in specific constructs and structures, and a pre-processing step of the target application to generate a (large) number of mutants. The execution of each mutant represents the injection of a fault.

Experimental results show that most of the patterns provide good accuracy, while some provide a not so good but still acceptable accuracy. This suggests that G-SWFIT is in fact a good way to emulate software faults when no source code is available.

The generalization and portability of G-SWFIT is mainly dependent on the target architecture, while aspects such as the compiler optimization settings, different compilers, and language used to program the target application only influence the size of the library of faults and the effort needed to generate this library.

References

1. M. Kalyanakrishnam, Z. Kalbarczyk, R. Iyer, ”Failure Data Analysis of a LAN of Windows NT Based Computers”, Symposium on Reliable Distributed Database Systems, SRDS18, October, Switzerland, pp. 178–187, 1999.
2. I. Lee and R. K. Iyer, “Software Dependability in the Tandem GUARDIAN System”, IEEE Transactions on Software Engineering, vol. 21, no. 5, pp. 455–467, May 1995.

3. M. Sullivan and R. Chillarege, “Comparison of Software Defects in Database Management Systems and Operating Systems”, Proceedings of the 22nd IEEE Fault Tolerant Computing Symposium, FTCS-22, pp. 475–484, July 1992.
4. J. Gray, “A Census of Tandem Systems Availability Between 1985 and 1990”, IEEE Transactions on Reliability, vol. 39, no. 4, pp. 409–418, October 1990.
5. J. Musa, “Software Reliability Engineering”, McGraw-Hill, 1996.
6. M. R. Lyu, “Handbook of Software Reliability Engineering”, IEEE Computer Society Press, McGraw-Hill, 1996.
7. J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, “Predicting how Badly ‘Good’ Software can Behave”, IEEE Software, 1997.
8. Wee T. Ng and Peter M. Chen, “Systematic improvement of fault tolerance in the RIO file cache”, Proc. of the 30th IEEE Fault Tolerant Computing Symposium, FTCS-29, Madison, WI, USA, June 1999.
9. Aaron Brown and David Patterson, "Towards availability benchmark: a case study of software RAID systems", Proceedings of 2000 USENIX Annual Technical Conference, San Diego, California, USA, June 18–23, 2000, pp 263–276.
10. M. Vieira, DSN ou VLDB
11. R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. Moebus, B. Ray, M. Wong, “Orthogonal Defect Classification – A Concept for In-Process Measurement”, IEEE Transactions on Software Engineering, vol. 18, no. 11, pp. 943–956, November 1992.
12. J. Carreira, H. Madeira, and J. G. Silva, “Xception: Software Fault Injection and Monitoring in Processor Functional Units", IEEE Transactions on Software Engineering, vol. 24, no. 2, February 1998.
13. S. Yamada, M. Ohba, and S. Osaki, “S-shaped software reliability growth models and their application”, IEEE Transactions on Reliability, vol. 33, no. 4, pp. 289–292, October 1984.
14. J. K. Chaar, M. J. Halliday, I. S. Bhandari, R. Chillarege, “In-Process Evaluation for Software Inspection and Test”, IEEE Transactions on Software Engineering, vol. 19, no. 11, pp. 1055–1070, November 1993.
15. T. Khoshgoftaar et. al., “Process Measures for Predicting Software Quality”, High Assurance Systems Engineering Workshop, HASE’97, Washington D.C., USA, IEEE CS Press, HASE’97, 1997.
16. J. P. Hudepohl et. al., “EMERALD: A Case Study in Enhancing Software Reliability"” Proc. of IEEE Eight Int. Symposium on Software Reliability Engineering, ISSRE’98, pp. 85–91, November 1998.
17. M. Sullivan and R. Chillarege, “Software defects and their impact on systems availability – A study of field failures on operating systems”, Proceedings of the 21st IEEE Fault Tolerant Computing Symposium, FTCS-21, pp. 2–9, June 1991.
18. R. K. Iyer, “Experimental Evaluation”, Special Issue FTCS-25 Silver Jubilee, 25th IEEE Symposium on Fault Tolerant Computing, FTCS-25, pp. 115–132, June 1995.
19. G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: A Tool for the Validation of System Dependability Properties”, Proceedings of the 22th IEEE Fault Tolerant Computing Symposium, FTCS-22, pp. 336–344, June 1992.
20. T. Tsai and R, K. Iyer, “An Approach to Benchmarking of Fault-Tolerant Commercial Systems”, Proceedings of the 26th IEEE Fault Tolerant Computing Symposium, FTCS-26, Sendai, Japan, pp. 314–323, June 1996.
21. J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Goofi: Generic object-oriented fault injection tool", Int. Conference on Dependable Systems and Networks, DSN-2001, Gothenburg, Sweden, July 1–4, 2001,

22. J. Hudak, B. Suth, D. Siewiorek, and Z. Segall, “Evaluation and Comparison of Fault-Tolerant Software Techniques”, IEEE Transactions on Reliability, vol. 42, no. 2, pp. 190–204, June 1993.
23. W. Kao, “Experimental Study of Software Dependability”, Ph.D. Thesis, Technical Report CRHC-94-16, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, USA, 1994.
24. T. Budd, “Mutation Analysis: Ideas, Examples, Problems, and Prospects”, in B. Chandrasekaran and S. Radicchi editors, Computer Program Testing, pp. 129–134, North Holland, 1981.
25. R. DeMillo, D. Guindi, W. McCracken, A. Offut, and K. King, “An Extended Overview of the Mothra Software Testing Environment”, Proceedings of ACM SIGSOFT/IEEE Second Workshop on Software Testing, Verification, and Analysis, pp. 142–151, July 1988.
26. M. Daran and P. Thévenod-Fosse, “Software Error Analysis: A Real Case Study Involving Real Faults and Mutations”, Proceedings of 3rd Symposium on Software Testing and Analysis, ISSTA-3, San Diego, USA, pp. 158–171, January 1996.
27. J. Christmansson and R. Chillarege, “Generation of an Error Set that Emulates Software Faults”, Proceedings of the 26th IEEE Fault Tolerant Computing Symposium, FTCS-26, Sendai, Japan, pp. 304–313, June 1996.
28. J. Christmansson and P. Santhanam, “Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms – Criteria for Error Selection using Field Data on Software Faults”, Proceeding of the Seventh IEEE International Symposium on Software Reliability Engineering, ISSRE’96, October 30 to November 2, 1996, New York, USA, 1996.
29. H. Madeira, M. Vieira, and D. Costa, "On the Emulation of Software Faults by Software Fault Injection", Proceedings of the International Conference on Dependable Systems and Networks, DSN 2000, New York, USA, pp. 417–426, 2000.
30. D. Costa, T. Rilho, M. Vieira, and H. Madeira, “ESFFI – A Novel Technique for the Emulation of Software Faults in COTS Components“, Proceedings of the 8th Ann. IEEE International Conference on the Engineering of Computer-Based Systems, ECBS2001, Washington D.C., April 2001.
31. J. Durães, and H. Madeira, “Emulation of Software Faults by Selective Mutations at Machine-code Level”, Proceedings of the 13th IEEE International Symposium on Software Reliability Engineering, ISSRE 2002, Annapolis, USA, November 2002.
32. João Durães and Henrique Madeira, "Definition of Software Fault Emulation Operators: a Field Data Study", IEEE/IFIP International Conference on Dependable Systems and Networks, Dependable Computing and Communications, DSN-DCC 2003, San Francisco, CA, USA, June 22–25, 2003.
33. R. Chillarege, “Orthogonal Defect Classification”, Chapter 9 of “Handbook of Software Reliability Engineering”, Michael R. Lyu Ed., IEEE Computer Society Press, McGrow-Hill, 1995.
34. International Olympiads in Informatics, http://olympiads.win.tue.nl/ioi/
35. A. Koenig, “C Traps and Pitfalls”, Addison-Wesley, 1989.

Managing Adaptive Fault Tolerant CORBA Applications*

Marcos A.M. de Moura[1]** and Markus Endler[2]

[1] Universidade de São Paulo - Instituto de Matemática e Estatística
Rua do Matão, 1010, 05508-900, São Paulo, Brazil
marcosammoura@ig.com.br
[2] PUC-Rio - Departamento de Informática
Marquês de São Vicente, 225 - Gávea, 22453-900, Rio de Janeiro, Brazil
endler@inf.puc-rio.br

Abstract. Only very recently CORBA specification, with the adoption of the Fault Tolerant CORBA standard (FT-CORBA) [10], addressed issues concerning object replication and reliable communication between objects, which are key components to provide fault tolerance for distributed object systems. This work presents the design and implementation of Juggler/OGS+, a full featured FT-CORBA-like fault tolerance infrastructure. We have created OGS+, an extension of a CORBA Group Communication Service called OGS [6], and on top of it we developed Juggler, a distributed service that provides means for flexible and automatic management of fault tolerant CORBA applications through the deployment of a set of high-level mechanisms and interfaces based on the ones standardized in FT-CORBA.

1 Introduction

In April 2000 OMG adopted the first version of Fault Tolerant CORBA specification (FT-CORBA) [10], which defines a set of interfaces, policies and services that allow the development of highly reliable CORBA [8] applications that require fault-detection and failure-recovery mechanisms. The standard is based on the use of replicated objects (object groups) and describes a set of fault tolerance strategies and mechanisms for the detection, notification and analysis of faults on object replicas.

In this paper we describe Juggler/OGS+ [5], an FT-CORBA-like fault tolerance infrastructure that allows the development and management of CORBA applications requiring high degree of reliability and availability. It is composed of two main services, namely Juggler and OGS+. OGS+ is an extension of OGS [6], a programming and execution environment that provides a flexible *object group* abstraction for the development of reliable CORBA applications. Juggler serves as the management tool for the applications developed with OGS+. It provides a

* Supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) - Proc. No. 98/06138-2.

** The author was supported in part by CNPq, Brazil.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 160–180, 2003.

set of high-level group management services and implements mechanisms similar to the ones proposed in FT-CORBA.

Juggler supports the use of several fault tolerance strategies defining the behaviour of object groups. It allows the use of different replication styles and fault tolerance properties on a per-group basis. Moreover, it has mechanisms that support a dynamic change of the fault tolerance behavior and the replication style of CORBA object groups. Thus, Juggler provides the system administrator with a fine grained control over the execution of fault tolerant CORBA applications, and define criteria and policies that allow these applications to dynamically adapt to the computational environment in which they execute.

One contribution of Juggler/OGS+ is the deployment of a complete execution and management environment suited for distributed applications with fault tolerance requirements, especially those with variable demands for service availability and performance. A second major contribution of this work is the design and implementation of a state-of-the-art approach to support dynamic (on-the-fly) modification of the degree of synchronization between object replicas. Finaly, another contribution of our work is the discussion of some issues concerning the implementation of an FT-CORBA-like infrastructure, where we share our practical experience of how we avoided some limitations of the FT-CORBA standard.

The remainder of this paper is organized as follows. In the next section we present an overview of the FT-CORBA standard. Section 3 describes some related work. Section 4 gives an overview of Juggler/OGS+'s architecture, services, and interfaces. In Section 5 some implementation issues, tests and results are discussed. Finally, Section 6 summarizes and concludes the paper.

2 Fault Tolerant CORBA

FT-CORBA [10] provides support for CORBA applications that require a high degree of reliability and availability, which is achieved through entity redundancy, fault detection, and recovery. The specification defines CORBA objects as the basic replication entities and object groups as the main abstraction to provide fault tolerance to applications. FT-CORBA supports several fault tolerance strategies, such as active and passive replication, request redirection and transparent method reinvocation, when passive replication is used. The standard defines mechanisms that allow the detection, notification and analysis of faults on object replicas. Moreover, FT-CORBA supports automatic failure recovery, and mechanisms for checkpointing and logging, which allow the fault tolerance infrastructure to automatically control the level of consistency between the object replicas that comprise an object group. The FT-CORBA architecture basically consists of the replication management, fault management, logging and recovery management services. In the following we describe some of its basic mechanisms and discuss some majorlimitations of the FT-CORBA specification.

2.1 Basic Mechanisms

FT-CORBA defines mechanisms that provide replication and failure transparency and that allow the deployment of arbitrarily large fault tolerant ap-

plications. FT-CORBA defines a special IOR, called *Interoperable Object Group Reference* (IOGR), which is used to address an object group. An IOGR contains multiple IIOP profiles, which are used to access the object replicas within a group. Each profile must contain a group profile stores an object group ID and the Fault Tolerance Domain in which the group is defined. Additionally, each IIOP profile may contain a component to denote the primary replica of the group, if it adopts a passive replication style.

The *Transparent Reinvocation* mechanism provides reinvocation of methods. It handles the failure of the primary member of an object group that uses a passive replication style, redirecting the client's request to a backup replica. In order to provide scalability and ease of management for fault tolerant applications, FT-CORBA defines *Fault Tolerance Domains* (FTDs). A domain typically contains several hosts and object groups. A host can be part of more than one FTD and all groups of objects within a domain are created and managed by a unique logical *ReplicationManager*.

2.2 Limitations of the Specification

Despite standardizing a rich set of interfaces, services and mechanisms to manage fault tolerant CORBA applications, FT-CORBA has some limitations, which are described as follows.

Legacy ORBs – The standard allows a client developed in a non-FT-CORBA ORB to invoke operations on replicated objects. However, such client cannot take full advantage from the fault tolerance properties provided through the use of object groups, since the client ORB does not understand IOGR references.

Lack of interoperability – FT-CORBA requires that within each Fault Tolerance Domain all the ORBs and fault tolerance infrastructures used must be from the same vendor, in order to guarantee interoperability and that all the fault tolerance mechanisms within the domain can be used.

Deterministic behaviour – When using a group consistency style controlled by the infrastructure, application objects ORBs must have deterministic behaviour. Multi-threading in the application or in the ORB must be restricted or should be avoided, unless transactional mechanisms are used.

Restricted failure model – The standard only supports crash (e.g. fail-stop) failures. Neither network partitions nor other types of object faults, such as commission or correlated faults, are considered in FT-CORBA. If one of such faults occur, fault tolerance is no longer guaranteed.

3 Related Works

Along with the emergence of the first environments for the development of fault tolerant CORBA applications some high level management infrastructures were developed. In the remainder of this section we give a brief overview of several research projects that are closely related to the fault tolerance infrastructure we have developed.

Piranha [14] was the first service for the management of CORBA replicated objects. It is built on top of Electra [13] and is composed of a collection of objects that implement notification, failure detection and restart services for CORBA applications. Piranha provides support for automatic restart of failed objects, dynamic replication of stateful objects, object migration and enforcement of a user-defined replication degree for each managed CORBA object group. Piranha is based on a very robust and fault tolerant architecture. However, it lacks adequate mechanisms for adaptation and dynamic modification of application-specific availability policies.

Proteus [18] provides fault tolerance in the AQuA architecture [3] by supporting dynamic management of replicated distributed objects, based on availability requirements specified by the application's programmer. In order to configure the application according to the desired availability, the user may choose the replication style, type of voting, degree of replication and the type of faults to tolerate. The other function of this dependability manager is to support dynamic modification of the configuration of an application based on the decision about the most adequate fault tolerance style to support.

DOORS [17] is an infrastructure that supports fault tolerance in CORBA by implementing a large subset of the functionalities, services and mechanisms standardized by FT-CORBA. In fact, the development of DOORS began before the adoption of FT-CORBA and many concepts and components defined in this service influenced the standard adopted by OMG.

GroupPac [11] is a set of services that allow the development of FT-CORBA compliant applications. GroupPac fully implements the mechanisms and services proposed in the FT-CORBA standard. The main divergence from the standard is that GroupPac also provides extensions in order to support fault tolerant applications in large scale asynchronous networks, which include modifications of the replication management function and supported failure model, aiming at the provision of a scalable infrastructure.

IRL [2], or Interoperable Replication Logic, is an FT-CORBA-compliant platform that provides transparent client-server interactions and server failover to application clients by using a set of replicated CORBA objects. IRL components cooperate through standard CORBA invocations, which guarantees inter-ORB interoperability for IRL components and fault tolerant applications.

Yet another related work is **ARM** [15], a replication management framework for partition-aware applications based on Jgroup [16]. ARM provides means to exchange replica distribution schemes and application specific recovery strategies, being also extensible to several replication and recovery strategies. This framework also enables autonomous replication management, requiring system administrator intervention only during creation and removal of object replicas that compose a group.

4 Juggler/OGS+

Our work aimed at providing a complete infrastructure for the development and management of fault tolerant CORBA applications based on groups of objects, and was materialized through Juggler/OGS+, a combination of two services:

Juggler and OGS+. Juggler is mainly concerned with offering a high-level interface to manage replicated objects (object groups) and OGS+ is responsible for controlling the configuration and the style of replication of object groups. Both these services assume that processor failures are of type *fail-stop* and that communication links are reliable, i.e. that the underlying communication layers either successfully deliver a message or detect that it has been lost. Network partitions and other types of failures, such as commission or correlated faults, are also not supported.

Juggler/OGS+ is best suited for fault tolerant applications demanding high service availability and adaptive consistency enforcement policies for the object replicas, so as to be able to adapt to varying patterns of network load and connectivity, e.g. as experienced in wireless networks. Similar requirements have been identified in the context of database applications for mobile networks [4] and cluster-based network services [20].

A concrete example of an application with such an adaptability requirement could be a highly available web site based on replicated servers, each of them with the capability of storing data provided by site visitors, e.g. a blog, a discussion-forum, etc. Since most accesses, e.g. web browsing, are read-only and non-critical with respect to the currency of the stored data, most requests could be redirected to any of the replicas, and for the same reasons, updates need not be immediately propagated to all the replicas but can be grouped into periodic, bulk updates in order to save network resources. However, some operations which modify/affect the functionality of the service, such as an update of an interface component (e.g. a cgi script for database access) should be made in a consistent way at all replicas, since otherwise some accesses may retrieve or store data in the wrong format. Hence, for such "critical update operations", the group of servers should temporarily switch to a replication style with a stronger consistency enforcement than during normal operation. This requires that the replica group be able to switch back and forth between active replication and cold passive replication styles, for example.

Dynamic adaptation of the replication style may also be necessary for fault tolerant applications that operate in networks with dynamically changing failure and performance patterns. In this context, it is important to compare the cost of maintaining synchronized replicas with the cost of recovering from a failure. It is known that passive replication suffers from the delays incurred by the election of the new primary replica, and hence this replication style may not be advantageous if the probability of host failures is high. However, updates are faster with this replication style than with active replication, since the replicas do not need to run a consensus protocol in order to guarantee atomicity and total order of their updates. Now, if some network links present high latency (or temporary disconnections) then consensus may become impossible (or unacceptably slow), which will probably have harmful consequences on service performance and availability. Hence, depending on the current network conditions, the probability of host failures, and priority of the application (high availability versus resilience) the application may require to dynamically switch between replication styles. These are only some examples which show the benefit of on-the-fly change of the replication style of a replica-based distributed service.

Figure 1 presents Juggler/OGS+'s architecture. It shows that application objects can access both Juggler and OGS+. Through Juggler, applications have access to a high-level service for managing object groups which takes care of tasks such as replica monitoring and transparent failure recovery. Through OGS+ application programmers can also control groups of object replicas, but since OGS+ provides only lower-level services, many of the management functions will have to be explicitly implemented as part of the application.

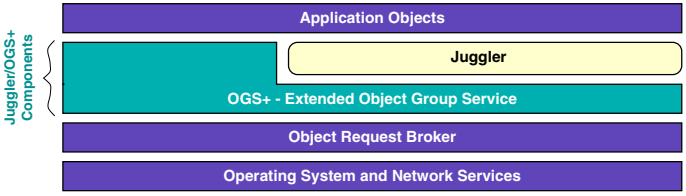


Fig. 1. Juggler/OGS+ architecture

Since OGS+ and Juggler are based on OGS's service approach for integrating group communication within CORBA [7], our fault tolerance infrastructure supports legacy ORBs and is able to interoperate across ORBs from different vendors, which are some of the main limitations of the FT-CORBA specification.

Although Juggler and OGS+ are clearly separated at the architectural level and have their own set of functionalities and interfaces these services work very closely at the implementation level, as discussed in the following sections.

4.1 Juggler

Juggler is a service that provides a robust infrastructure for managing fault tolerant CORBA applications. It provides interfaces for managing the configuration of groups of objects, as well as mechanisms for automatic detection and recovery of failed objects.

Through Juggler, it is possible to define a specific replication style and fault tolerance properties, and to assign them to each object group individually. These properties define, for instance, the location and the minimum number of replicas required by a group, and also allow to choose the most suitable failure recovery strategy for a group. Juggler supports the creation and management of object groups with different replication styles, such as active replication [19], primary-backup replication [1] and a variant of the virtual-primary-copy [12] replication. It also supports dynamic modification of the fault tolerance properties and replication style associated with each object group.

Juggler is built upon OGS+, an extension of OGS [6]. OGS provides basic support for creating and destroying CORBA object groups, adding and removing group members, notifying view changes and performing state transfer among group members. We have chosen OGS as the basis of our infrastructure mainly because it has mechanisms for specifying different communication and replication

styles for each object group and also due to its support for monitoring and failure notification of individual objects within object groups. Furthermore, OGS provides group communication using only CORBA standardized constructs, which guarantees portability and interoperability.

Juggler makes extensive use of two main components of OGS+, namely the *Monitoring Service* and the *Object Group Service*. It uses the *Monitoring Service* to receive notifications about failure suspicions of monitored objects (i.e. group members), and the *Object Group Service* for managing the object group's life cycle and for reliable group communication. Juggler is composed of three main components: `GroupManager`, `GroupActor` and `GroupObserver`, which are specified as collections of OMG IDL interfaces and are implemented as groups of replicated objects. As shown in Fig. 2, Juggler can be seen as a unique logical entity that is composed of these three components working tightly together.

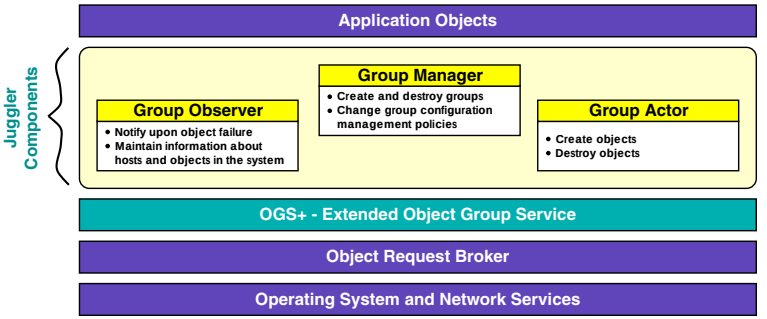


Fig. 2. Juggler architecture

`GroupManager` mainly deals with group creation and destruction, as well as dynamic modification of group configuration. It does this by invoking methods on `GroupActor`, which is responsible for creation and destruction of individual object replicas. `GroupObserver` is responsible for suspecting failures of object replicas and sending the corresponding notifications to `GroupManager`, which in turn automatically executes some actions to maintain a required degree of resilience of the corresponding group. Its behaviour is mainly determined by the group-specific management parameters provided by the service administrator.

Each object group created through Juggler has its fault tolerance management parameters stored in a structure called `Semantics`, whose attributes are discussed as follows. Attribute `type_id` contains the identification of the object replicas within CORBA Interface Repository [8], while `the_locations` stores a list of the locations where the group members are executing. A location is a physical host, or cluster, where object replicas can be created. One restriction imposed by Juggler is that for each group, only one replica can exist at a given location, which makes sense when considering only crash faults.

Other attributes are `initial_number_replicas` and `minimum_number_replicas`, which hold, respectively, the initial and minimum number of replicas

that should exist in the group. `Minimum_number_replicas` only makes sense if the `automatic_recovery` attribute is set, which indicates that Juggler must automatically recover failed object replicas. Otherwise an external entity must control the minimum number of object replicas within the group. Finally, attribute `replication_style` stores the replication style associated with the group and the specific properties related to the style. Juggler supports stateless, cold passive, warm passive, active and virtual-primary-copy replication styles, which are discussed in Sect. 4.2.

Architectural Components

GroupManager. The `GroupManager` component basically provides means for the creation and destruction of object groups, and operations to modify and to obtain their associated semantics. It implements the main interface of Juggler (`GroupManagementOperations`) and other interfaces that allow for it to be replicated and to receive notifications of failed object replicas. The `GroupManagementOperations` interface, partially shown in Fig. 3, provides functionalities similar to the ones defined by FT-CORBA in the `ObjectGroupManager` interface.

```
interface GroupManagementOperations {
    GroupInfo create_group(in ObjectGroup object_group,
              in Semantics the_semantics,
              in FT::Criteria the_criteria)
    GroupInfo set_replication_style(in ObjectGroup object_group,
              in ReplicationStyle the_replication_style)
    GroupInfo set_minimum_number_replicas(
              in ObjectGroup object_group,
              in FT::MinimumNumberReplicasValue minimum_number_replicas)
    GroupInfo set_recovery_style(in ObjectGroup object_group,
              in AutomaticRecoveryValue automatic_recovery)
};
```

Fig. 3. `GroupManagementOperations` interface

Operation **create_group** is used to create a new object group and **set_replication_style** allows the modification of the replication style associated with a group, which can be done while the group is being used. The protocol used for this dynamic switch operation is discussed in Sect. 4.2. Operation **set_minimum_number_replicas** is used to specify the minimum number of replicas that should exist in a group and **set_recovery_style** determines whether Juggler is responsible for automatic recovery of failed group members. Other operations, not shown in Fig. 3, are used to obtain the location and a reference of individual group members and to retrieve information concerning the object groups managed by Juggler.

GroupActor. The `GroupActor` component is responsible for the creation and destruction of individual object replicas. These operations are inherited from the `GenericFactory` interface, which is exactly the same as defined in the FT-CORBA standard. `GenericFactory` interface must also be implemented by the application programmer in order to allow the creation of objects of specific types at several locations. In fact, `GroupActor` does not create or destroy objects, but delegates these operations to the Object Factories (developed by the application programmer) registered with Juggler. Thus, `GroupActor` behaves as a Generic Object Factory, since it can handle the creation of any type of object replica.

`GenericFactory` interface defines **create_object** and **delete_object** operations, which create and delete object replicas, respectively. In addition to the `GenericFactory` interface, `GroupActor` also implements the `FactoryOperations` interface, which allows adding and removing the registries of application-specific Object Factories, used to execute **create_object** and **delete_object** operations on behalf of `GroupActor`.

GroupObserver. In Juggler the tasks concerning monitoring and suspecting failure of replicas are defined in two interfaces inherited by the `GroupObserver` component, namely `ReplicaMonitoring` and `FaultNotifier`, shown in Fig. 4. `FaultNotifier` interface offers a subset of the functionalities provided by the CORBA Notification Service (`CosNotification` [9]), and is identical with the interface specified in FT-CORBA [10].

```
interface FaultNotifier {
  void push_structured_fault(
            in CosNotification::StructuredEvent event);
  void push_sequence_fault(in CosNotification::EventBatch events);
  ConsumerId connect_structured_fault_consumer(
            in CosNotifyComm::StructuredPushConsumer push_consumer);
  ConsumerId connect_sequence_fault_consumer(
            in CosNotifyComm::SequencePushConsumer push_consumer);
  void disconnect_consumer(in ConsumerId connection)
  void replace_constraint(
            in ConsumerId connection,
            in CosNotification::EventTypeSeq event_types,
            in string constr_expr);
};
```

Fig. 4. `ReplicaMonitoring` and `FaultNotifier` interfaces

`ReplicaMonitoring` interface defines the **start_monitoring_replica** and **stop_monitoring_replica** operations, which allow the `GroupObserver` to start and stop monitoring object replicas created by Juggler through the `GroupActor`. `FaultNotifier` interface allows the `GroupObserver` to forward notifications concerning failed object replicas to the `GroupManager`, which is responsible for recov-

ering them. Objects that wish to receive these notifications must register with `FaultNotifier` through either the **connect_structured_fault_consumer** or **connect_sequence_fault_consumer** operations.

Push_structured_fault and **push_sequence_fault** operations are used to send notifications, as structured events (`CosNotification::StructuredEvent`) or event batches (`CosNotification::EventBatch`), respectively, to the event consumers registered with `FaultNotifier`. Operation **disconnect_consumer** is used to disconnect an event consumer from the `FaultNotifier` and the **replace_constraint** operation allows a consumer to specify which types of events it wants to receive from the `FaultNotifier`. `GroupObserver` also implements the `LocationOperations` interface, which allows updating/retrieving information concerning the hosts where Juggler is running.

ReplicationManager. In addition to the `GroupManager`, `GroupActor` and `GroupObserver` components, Juggler also defines the `ReplicationManager` component, whose sole purpose is to aggregate all the operations made available to Juggler clients. Although logically there is only one such component in Juggler, in fact it is a replicated component, ensuring fault tolerance of the management infrastructure itself. The interface provided by this component may also serve as the basis for the implementation of other Juggler clients, e.g. one with a GUI.

`ReplicationManager` inherits interfaces `GroupManagementOperations`, `FactoryOperations` and `LocationOperations` and defines **register_fault_notifier** and **get_fault_notifier** operations. The first one is used to register `FaultNotifiers` with Juggler and the latter allows to retrieve an IOR for a `FaultNotifier` previously registered with the service. A Juggler client may, optionally, obtain such an IOR so that it can register itself with the `FaultNotifier` in order to receive notifications of events generated by Juggler. This is the approach we are using to implement JugglerGUI, a graphical administration tool for Juggler/OGS+.

Group Management

In this section we present the protocols that Juggler uses for group creation, failure recovery of object replicas and modification of an object group configuration. In order to implement these protocols as fault tolerant services Juggler defines three OGS groups, namely *MANAGERS*, *ACTORS* and *OBSERVERS*. These groups comprise, respectively, all *GroupManager*, *GroupActor* and *GroupObserver* objects running in the system. In a Juggler/OGS+ environment one *GroupActor* and one *GroupObserver* must be instantiated on each host of the system, and there should be at least two *GroupManager* instances executing on two different hosts, so as to guarantee redundancy of the fault tolerance management infrastructure.

Group *MANAGERS* maintains a replicated *Group Information Table* (GIT), which contains information about all the object groups, such as their names, semantics and lists of members. Since group *ACTORS* is responsible for the creation and destruction of object replicas, it stores the *Factory Information*

Table (FIT), which contains information on all Object Factories registered with Juggler. Another replicated structure stored in group *ACTORS* is the *Object FactoryCreationId Map Table* (OFMT), which contains a mapping between the IOR of each object replica and its corresponding `FactoryCreationId`. Group *OBSERVERS* also maintains two replicated tables, namely *Location Information Table* (LIT) and *Object Information Table* (OIT), which hold information about all the hosts where Juggler is running and the monitoring of object replicas created at these hosts, respectively.

The tables maintained by groups *MANAGERS*, *ACTORS* and *OBSERVERS* are updated through operations defined at the *GroupManager*, *GroupActor* and *GroupObserver* interfaces, respectively. The names of these operations are very intuitive and will be mentioned during the explanation of the group creation and failure recovery protocols, discussed as follows.

Group Creation. In order to facilitate the understanding of the group creation protocol we describe the creation of an object group named *Test*, with object replicas located at hosts *A* and *C*, using active replication style (see Fig. 5).

Initially, a client application invokes the **create_group** operation on any *ReplicationManager* object made available through the CORBA Naming Service. This object delegates this operation to one of the *GroupManager* objects. Then, *GroupManager* invokes the **create_object** operation on group *ACTORS* and waits for the replies from all of its members. Each member of group *ACTORS* checks whether an object replica should be locally created (in our example, in hosts *A* and *C*) and then the registered Object Factories create the replicas. *GroupActor* objects that created a replica then invoke the **add_object_factoryId** operation on group *ACTORS*, which is used to update the OFMT.

Next, each *GroupActor* invokes the **start_monitoring_object** operation on the local *GroupObserver* object passing the **factory_creation_id** of the newly created replica as an argument. Then, the *GroupObservers* asynchronously invoke **add_monitoring_info** operation on group *OBSERVERS*, thus updating the OIT. Each *GroupActor* waits for the completion of the **start_monitoring_object** operation and then returns a reference to the *GroupManager* that started the protocol. After receiving all the replies, the *GroupManager* stores the information concerning the new group and disseminates this information by asynchronously invoking the **add_group_info** operation on group *MANAGERS*, which updates the GIT. Finally, the *GroupManager* object that started the protocol returns the information concerning the new object group to the client that requested the operation.

Failure Recovery. Juggler implements a protocol for automatic failure recovery of object replicas. To facilitate the description of this protocol we will use group *Test* again, assuming that it has been created as described in the previous section. Now let us suppose that a failure occurs in the object replica located on host *C* and that the semantics of the group specifies that Juggler should automatically recover failed objects.

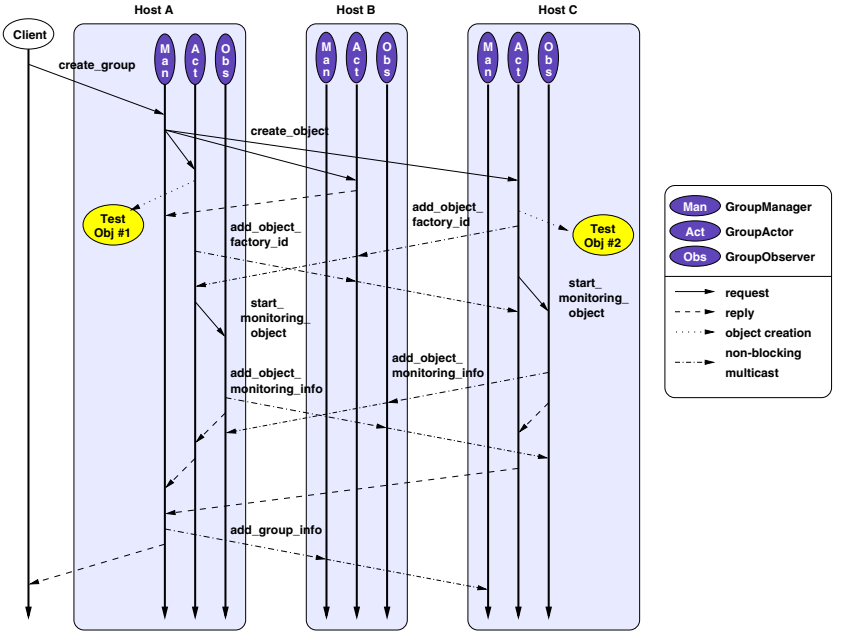


Fig. 5. Group creation protocol

When an object replica fails, *GroupObserver* receives a notification failure suspicion through the Monitoring Service provided by OGS+. Next, *GroupObserver* invokes the **remove_object_monitoring_info** operation on group *OBSERVERS* (updating the OIT), and sends a notification about the failure to the group *MANAGERS*. Then, one deterministically chosen *GroupManager* executes the failure recovery procedure: it creates a new replica on the same host where the failed object was executing by invoking the **create_object** operation on the *ACTORS* group. From this point on, the failure recovery proceeds similarly to the creation of a single object replica. It only differs from the protocol described in the previous section in that the last operation invoked on group *MANAGERS* is **update_group_info**, which updates the *Group Information Table* (GIT).

Group Semantics Modification. As mentioned before, Juggler provides means for dynamic modification of the semantics associated to object groups. This allows Juggler to deal with applications whose fault-tolerance requirements, such as the number of object replicas or the replication style to be used, change over time. Moreover, this feature supports adaptive fault tolerance in the sense that applications managed by Juggler may be automatically reconfigured in response to variations of the system performance, such as changes in the computational load at hosts or the performance of network links.

Changes of the group configuration (semantics) which affect the number or location of the object replicas are performed automatically by Juggler through

the creation and/or removal of object replicas on the hosts specified in the `the_locations` attribute of the group semantics. Changes of the replication style of a group are also performed automatically, and will be discussed in Sect. 4.2.

4.2 OGS+

OGS+ is an extension of OGS (Object Group Service) [6], a Group Communication service for CORBA based on the service approach [7] which provides means of creating and maintaining CORBA object groups using only CORBA standardized constructs. OGS+ is implemented in Java and is built upon OGS version 0.8b1 using VisiBroker 4.0CORBA ORB.

OGS is composed of four main components: *Group Service*, *Consensus Service*, *Monitoring Service* and *Messaging Service*. The *Group Service* allows the management of object groups supporting state synchronization (i.e. consistent states) among group members. It also provides means of sending multicast invocations to object groups with different degrees of ordering and reliability. The *Consensus Service* solves some consensus-related problems in OGS, such as group membership and total ordering of messages. The *Monitoring Service* provides mechanisms to detect crashed objects and to disseminate fault event notifications. Finally, the *Messaging Service* supports multicast communication with different reliability, ordering and synchronization guarantees.

On the server side OGS manages object groups through *Group Administrator* objects, where each such object is associated with one (application) object replica in the group. Completely transparent to the application objects, *Group Administrator* objects interact with each other to execute consensus and voting protocols, and to implement active replication style, which is the single replication style supported in OGS. From the perspective of the application object, each *Group Administrator* object is responsible for delivering multicast messages and notifying group view changes to the group member attached to it.

On the client side, OGS provides *Group Accessor* objects (i.e. proxies), which enable clients to issue invocations to the object replicas' methods, without knowledge of the number or location of the replicas, and specifying if one, some or all replies are required. *Group Accessor* is actually the element in charge of sending multicasts to *Group Administrator* objects and waiting for the replies.

When active replication is used, OGS' protocol to maintain replica states synchronized has huge message complexity which is proportional to the number of replicas. Therefore, this replication style causes considerable delay when processing method invocations on a group of replicated objects. This happens because all group members must agree on the order in which the request will be delivered to and processed by all group replicas.

However, for several applications it makes sense to trade the resilience degree against service responsiveness. In such cases, it may be sufficient that only a core (sub)set of the object replicas in the group keep their states perfectly synchronized, while the other replicas' states may be updated periodically or on demand. In order to support also these weaker forms of replica consistencies, OGS+ provides several replication styles other than active replication, and also supports dynamic switching between them.

Replication Styles. In addition to the active replication style originally implemented in OGS, OGS+ also supports three replication styles specified in FT-CORBA, namely stateless, cold passive and warm passive (see Sect. **??**). Additionally, OGS+ also implements a variant of the virtual-primary-copy replication style [12], which can be seen as a generalization of the warm passive replication style with more than one primary copy.

In order to implement these replication styles we extended OGS's *Group Administrator* and *Group Accessor* classes. The *Extended Group Administrator* objects implement the new replication styles and a protocol to dynamically switch between them. The *Extended Group Accessor* object stores the information about the current replication style being used by the object group to which they are connected, and interact with the group's *Extended Group Administrator* objects to execute the corresponding multicast protocol.

In the virtual-primary-copy replication style one or more active replicas behave as primary replicas, thus forming a *virtual primary replica* or *consistency island* (or simply, *island*). The remaining replicas within the group are backups of the virtual primary replica. Interaction among the two sorts of replicas is done similarly to warm passive replication. The main advantage of this replication style when compared to warm passive replication is the guarantee of a higher availability of the (virtual) primary replica. When compared with active replication on the entire group, virtual-primary-copy incurs in less overhead for the synchronization of the active replica.

Dynamic Modification of the Replication Style. In order to enhance flexibility of applications based on replicated objects, OGS+ provides means to dynamically modify the replication style associated with an object group. Such flexibility is especially required for applications in dynamic, i.e. mobile networks, where object replicas may be distributed both at mobile and fixed hosts, and where mobile hosts may become temporarily disconnected or may experience strong variations in the quality (e.g. bandwidth, latency) of their links to the remaining network.

When the communication link to a host (e.g. mobile host) with a replica starts do degrade, eventually it may become necessary to change the replication style of the object group, e.g. from active to cold passive, or to virtual-primary-copy. Therefore, the protocol used for enforcing the consistency of the remaining replicas will be less affected by the slow (or broken) connection, allowing the replicated service to keep a minimum degree of resilience and still maintain a reasonable service response time. In addition, the replica at the weakly connected (or disconnected) host may still be accessible locally, however, with the obvious lesser guarantee of its state freshness. Hence, according to the current quality of the connectivity among the replicas, a replicated service can trade replica consistency for service performance in a dynamic and automatic way.

Several other works suggest that replication styles and consistency requirements ought to be adaptive and flexible when used in networks with highly dynamic connectivity and load patterns. In the specific context of database applications for mobile networks, Barbara and Molina [4] propose a *weak-transaction model* with a flexible approach to data replication. But also for services in wired

networks, project Neptune [20], which deals with cluster-based services, is an example where a service-specific replication scheme to cope with varying performance trade-offs was adopted.

Figure 6 shows an example depicting how the protocol for dynamic change of the replication style works. It shows the steps performed during the change of the replication style of a group composed of two replicas (located at hosts *A* and *B*) from active to warm passive replication style. In this scenario we assume that after the style change the primary replica of the group will be the one executing at host *A*. We also assume that during the modification of the replication style the object group is accessed by an usual client.

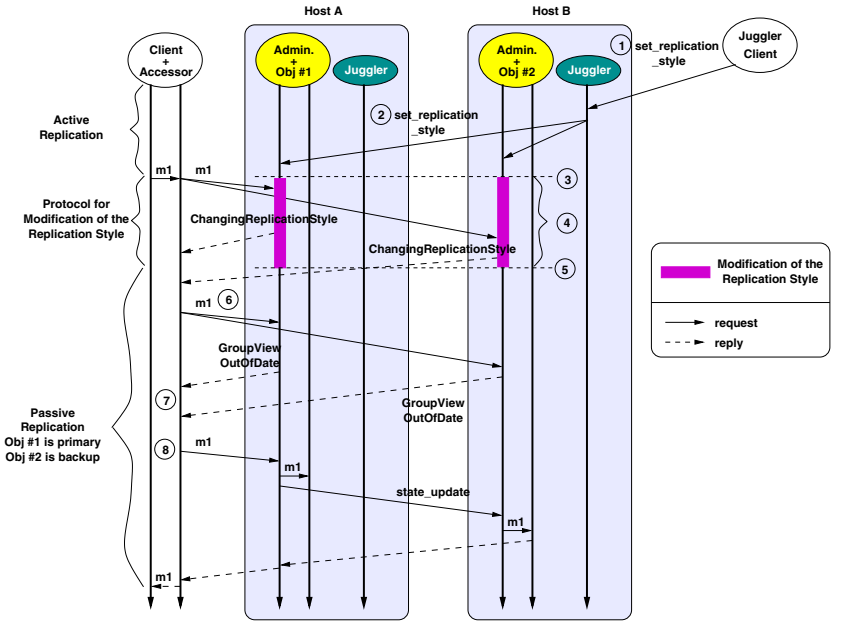


Fig. 6. Dynamic modification of the replication style of an object group

In order to change the replication style of a group a client of Juggler/OGS+ obtains a reference to a *ReplicationManager* object and invokes **set_replication_style** operation (step 1), which is sent via multicast to the *Extended Group Administrators* associated with the group members (2). The *Extended Group Administrators* then block the processing of new requests (3). While the replication style is being changed any request sent by clients (i.e by the *Extended Group Accessors*) of the group is replied with a *ChangingReplicationStyle* exception (4). After receiving such a reply, an *Extended Group Accessor* enters a loop where it periodically reissues the request in predefined time intervals.

After updating their internal structures with the new replication style of the group, the *Extended Group Administrators* unblock the group so that it can receive and process new requests (5). At this point, the replica located at host *A* becomes the primary replica and the one located at host *B* becomes the backup replica. On the client side, the *Extended Group Accessor* object reissues the multicast operation to the group (6), which is promptly replied with a *GroupViewOutOfDate* exception (7). This exception informs the *Extended Group Accessor* that its group view must be updated. This exception also carries information concerning the new replication style associated with the group. Finally, the *Extended Group Accessor* updates its view of the group and reissues the multicast again (8). Because of the new replication style this time, it only sends the request to the primary replica of the group. The request is then processed, the state of the backup replica is updated and the reply is returned to the client.

5 Tests and Results

Currently, Juggler/OGS+ is a fully fuctional service. It was developed using VisiBroker for Java 4.0 and, for test purposes, we have implemented a simple banking application composed of replicated objects. These objects implement the `Account` interface, which specifies operations to deposit and withdraw money from a hypothetic bank account, as well as an operation to get the current account balance, implemented as the state of the object replicas. Tests were performed on a local 10Mbit Ethernet network interconnecting 9 Sun SPARCstations 4 shared by other users. Tests were run with the TCP_NODELAY option set, which disables buffering and forces the immediate sending of requests. All tests described throughout this section assume that object replicas have been previously added to the object groups and that the groups are ready to accept and to process client requests. One important test performed aimed at evaluating the time needed by Juggler/OGS+ to recover failed object replicas. In order to run this test, we created groups with 1 to 9 members and simulated the failure of one of these members. During the execution of this test the time necessary to detect the failure was not computed, since it essentially depends on monitoring parameteres defined in the OGS+ Monitoring Service and may be adjusted to arbitrary values on each host where Juggler/OGS+ is running. The values shown in the lines of the graph are accumulated, i.e. they represent the sum of the time needed to accomplish the step(s) that precede the step in discussion plus the time required to perform the step in study, and the results represent the average values obtained from 25 runs of the test for each group size.

The failure recovery process in Juggler/OGS+ basically consists of three steps: (1) a notification concerning the failure is sent to the group *MANAGERS*; (2) selection of the *GroupManager* responsible for carrying on the failure recovery; and (3) creation of a new group member. Figure 7 illustrates the time spent to perform each of these steps.

For groups composed of 3 replicas, the time necessary to send a notification concerning the failure (step 1) was about 234ms. Steps (2) and (3) consumed, on the average, 115ms and 1.29s respectively, which makes up a total recovery

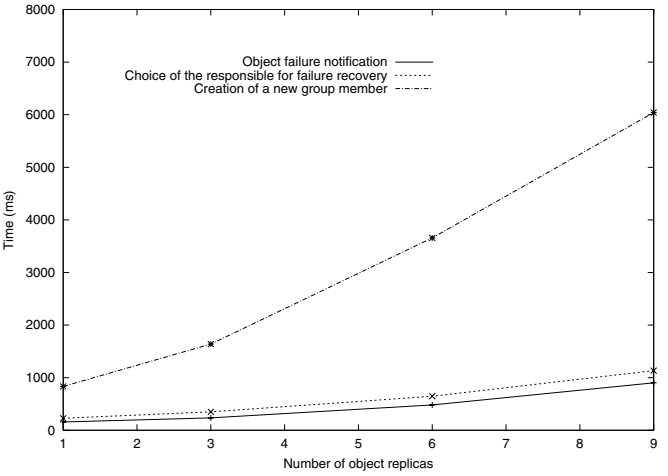


Fig. 7. Time spent during the recovery of a failed object replica

time of about 1.64s. For groups with 6 and 9 replicas the total time spent to recover a failed object was aproximately 3.65s and 6.04s, respectively. From the graph one can see that the creation of a single object replica grows in a non-linear pattern. This is mainly due to the fact that the reliable multicast protocol implemented in OGS spends an exponential time to perform its task, as described in [6]. Since the creation of an object replica through Juggler/OGS+ involves the exchange of several multicast messages, the poor performance of OGS's multicasts is the determinant factor of the time spent to recover a single failed replica. The obvious solution to overcome this performance problem is to replace OGS's atomic multicast protocol with a more efficient one.

Another test performed with Juggler/OGS+ measured the time spent by an application client to issue and receive a response for an atomic multicast sent to an object group. This test was done for groups composed of 3 members being concurrently accessed by 1, 5, 10 and 15 clients with both active and warm passive replication styles. The results of this test are shown in Fig. 8, where the values shown in the graph represent average values for sending a few hundred multicasts by each client set.

As can be seen in Fig. 8, the average time spent to send an atomic multicast to groups that use active replication was aproximately 50% to 60% higher than when warm passive replication style was used. The difference between the times for active and warm passive replication can be much higher if a small improvement is made in the protocol that implements the warm passive replication style. This improvement consists of sending a reply to the client as soon as the operation has been processed by the primary replica, instead of waiting for the acknowledgements of the state updates from the backup replicas before sending the reply to the client. With this optimization, however, there is no guarantee for the client that the primary's state has been successfully backed-up.

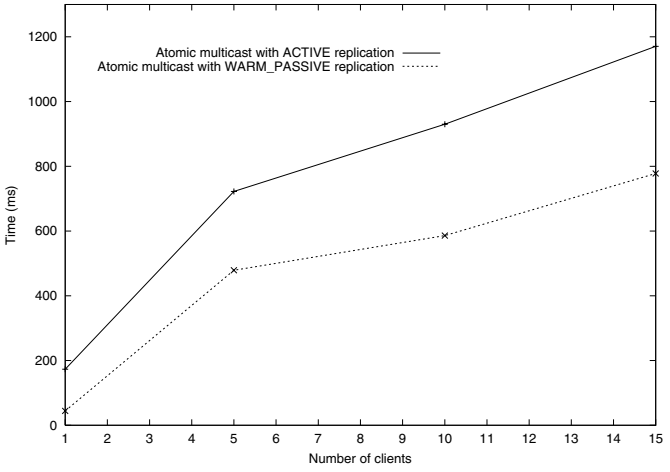


Fig. 8. Time spent to send an atomic multicast to an object group

A third test evaluated the time to issue (and receive the reply of) an atomic multicast sent by a client while the replication style of a group is dynamically changed. In this test we measured the time to switch from active to warm passive replication styles, and vice-versa. To perform this test a group with 3 members was created and its replication style was alternately switched between active and warm passive replication.

During the change from active to warm passive replication style, while the group was accessed by 1 and 5 clients, the average time spent was 163ms and 2.02s, respectively. The opposite operation consumed 268ms and 3.19s, respectively, for 1 and 5 clients. Thus, our measurements indicate that the change from warm passive to active took about 50% more time than when changing from active to warm passive replication style.

For groups accessed by 10 and 15 clients, the change from active to warm passive replication took, respectively, 2.16s and 6.71s. However, when we performed the opposite operation the information concerning the replication style, which is stored in the *Extended Group Administrators*, became inconsistent due to the large number of concurrent requests sent to group, thus blocking the group for the reception of new requests. This problem occurs due to a design decision made in the OGS Consensus Service. In OGS a consensus is reached even if there is only the single majority of the propositions of the participants in the consensus protocol. Thus, if the primary replica of the group becomes flooded with client requests, a request concerning the replication style modification may be delivered first to the backup replicas and later to the primary replica. So, it can happen that some *Extended Group Administrators* store stale information concerning the replication style of the group, hence preventing the group from processing new requests.

6 Conclusion

Support for fault tolerance is a usual requirement of many dependable distributed applications. Through our work we aimed at providing high-level abstractions that ease the management of fault tolerant CORBA applications. Juggler/OGS+ supports monitoring and automatic failure recovery of object replicas, availability management and specification of several group management policies for fault tolerant applications. In addition, it is portable and interoperable with different CORBA ORBs. In our work we have adopted some ideas presented in related works. Like Piranha, Juggler/OGS+ replicates the information concerning object groups managed by the service. The flexibility of specifying different management policies for groups of object replicas came from Proteus. In addition, Juggler also implements a full set of services, such as a generic Object Factory, a fault notification service and mechanisms for check-pointing and logging, which follow the guidelines specified in FT-CORBA.

One of the main differences between Juggler/OGS+ and FT-CORBA is that our fault tolerance infrastructure implements a variant of the virtual-primary-copy replication and provides means to dynamically switch between different forms of replication, which provides applications with an extra degree of flexibility for adapting to varying network conditions.

We have implemented and tested a fully functional version of Juggler/OGS+. We have evaluated its performance and identified the major overheads associated with the implemented protocols. Juggler/OGS+ is clearly not at the stage to be used for real-world applications, mainly due to the lack of scalability and other smaller problems that may occasionally occur. Some of the problems were inherited from OGS, which implements non-optimal algorithms for reliable multicast and consensus, as well as the use of CORBA Naming Service as the repository of the IORs associated with group members. Another problem is that OGS requires deterministic behavior of the application objects, i.e. single-threading, which restricts the programmability of applications that may use Juggler/OGS+. Yet another drawback is that Juggler/OGS+'s internal protocols are still consensus-based, and do not support well disconnections and network partitions.

Thus, Juggler/OGS+ seems to be suitable for applications with small-scale replication of objects e.g. up to 5-7 replicas per group, and which do not require very small or predictable service response times.

However, we think that Juggler's capability of performing automatic and group-specific failure recovery is a very handy and important feature for long-running applications, since especially for CORBA applications this task is very error prone when performed by a human. Moreover, Juggler introduces only very little overhead, since it only has any effects during the initial creation of object groups and when the semantics of the groups is changed, while during most of the time groups are accessed directly through OGS+. It is also important to note that the overhead of the group management service offered by Juggler will be noticed only by the applications that use it, without affecting the performance of other applications.

Currently, we are implementing features that allow the safe use of Juggler/OGS+ when accessed by many clients concurrently and are developing a

Graphical User's Interface with a more user-friendly management interface. The complete Juggler/OGS+ software is free, open source and can be downloaded from the URL *http://www.ime.usp.br/*sin*sidam/components/juggler/*.

As part of future work we plan to substitute OGS's protocol for atomic multicast, modify the implementation of OGS Consensus Service in order to support reliable switching of replication styles under heavy load, and optimize some of Juggler's protocols by making them less dependent on all replicas. At a later stage, we also plan to incorporate FT-CORBA's concept of Fault Tolerance Domains into Juggler/OGS+, since this will enable the management of large-scale systems.

References

1. P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. *2nd International Conference on Software Engineering*, pages 562–570, 1976.
2. R. Baldoni, C.Marchetti, and A.Termini. Active Software Replication through a Three-tier Approach. In *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS'02)*, pages 109–118, Osaka, Japan, October 2002.
3. M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, Indiana, USA, October 1998. IEEE.
4. Barbará D. and Garcia-Molina H. Replicated Data Management in Mobile Enviornments: Anything New Under the Sun? In *Proc. IFIP Conf. on Applications in Parallel and Distributed Computing, Caracas, Venezuela, April*, April 1994.
5. M. A. M. de Moura. Uma Infra-Estrutura para o Gerenciamento de Aplicações CORBA Tolerantes a Falhas. Master's thesis, Universidade de São Paulo, Brasil, Fevereiro 2001. In Portuguese.
6. P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.
7. P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS '96)*, pages 150–161, Los Alamitos, CA, USA, October 1996. IEEE Computer Society Press.
8. Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG, June 2002. v3.0.
9. Object Management Group. CORBAservices: Notification Service Specification. OMG, August 2002. v1.0.1.
10. Object Management Group. Fault Tolerant CORBA Specification. OMG, June 2002.
11. Lau C. L., J. Fraga, L. Souza, and R. Padilha. GroupPac: Um Framework para Implementação de Aplicações Tolerantes a Falhas. In *Anais do XXVI Conferencia LatinoAmericana de Informatica - CLEI'2000*, Estado de México, México, Setembro 2000.

12. R. Lenz. The Virtual-Primary-Copy Approach Compared to other Approaches with Weak Consistent Data Replication. In *II Workshop on Mobility and Replication, ECOOP'96*, Linz, Austria, July 1996.
13. S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, Switzerland, 1995.
14. S. Maffeis. Piranha: A CORBA Tool for High Availability. *IEEE Computer*, 30(4):59–66, April 1997.
15. H. Meling and B. E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS)*, Bertinoro, Italy, May 2001.
16. A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, University of Bologna, July 2000.
17. B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards High-performance Fault Tolerant CORBA. In *Proceedings of the 2nd Distributed Applications and Objects (DOA) Conferece*, Antwerp, Belgium, September 2000.
18. B. S. Sabnis. Proteus: A Software Infrastructure Providing Dependability for CORBA Applications. Master's thesis, University of Illinois, USA, 1998.
19. F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
20. K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, , and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, March 2001.

Adaptable Objects for Dependability

Jose Lino Contreras[1] and Jean Louis Sourrouille[2]

[1]UTFSM, Departamento de Informática, Valparaíso, Chile
Jose.Contreras@inf.utfsm.cl
[2]INSA de Lyon, PRISMa, Bat. B. Pascal,F69621 Villeurbanne Cedex
Jean-Louis.Sourrouille@insa-lyon.fr

Abstract. Computer applications are meant to satisfy functional and non functional constraints derived from their design specifications. Nevertheless, in current highly dynamic execution environments, unpredictability of execution contexts makes it difficult to ensure that during execution some system properties such as times of response, precision, security and so for, will behave as expected. This uncertainty affects system dependability negatively. Static approaches are effective if all events affecting the system can be predicted or bounded, which is not the case for most of current systems. For more general cases, our approach is to provide dynamic adaptability capabilities to applications in order for them to deal themselves with unexpected situations. Indeed, we bring these capabilities as close as possible to where real action takes place, that means, at object level in object oriented systems. Being provided with adaptation capabilities and guided by appropriate decision criteria, our objects adapt their behavior dynamically in function of their execution contexts, improving in this way system dependability.

Keywords: Adaptation, Dependability, QoS, Active objects, Soft Real-Time, Metaobject-based architecture

1 Introduction

Current computer systems are characterized by very complex, dynamic and less predictable execution contexts, making it difficult for software designers to ensure dependability, often expressed as non functional properties of applications. Static approaches provide good solutions when all events affecting the system are known or bounded, but they are not so capable of dealing with unexpected situations. On the other side, dynamic approaches can't ensure the total respect of all constraints, but they can deal with unknown situations.

The ARTO project (*Adaptable Real-Time Object*) aims to define a model for autonomous objects that modify their behavior dynamically according to changes in their execution context. It deals with ordinary systems receiving service requests at unknown instants. Although the original intention of the project was to apply ARTO on the *soft* real-time domain, the underlying principles of the model aim to improve system dependability in unpredictable execution contexts. Given its open nature, the model may be easily adapted to deal with non functional constraints in other domains.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 181–196, 2003.

In ARTO all non functional aspects, such as time related constraints, are treated as part of the quality of the services that objects provide and adaptation decisions are driven by QoS criteria.

There are many views of adaptation in the software development community. From *Gluons* that supply dynamic object connection [20] to applications that provide a tunability interface for run-time adaptation [3], including middlewares managing the quality of service [1]. Adaptable elements can be objects (as in this project) or actors, but also applications [3] or components. Such a multiplicity and diversity does not exclude the use of common mechanisms.

In its initial version, the project has been restricted for execution on a single processor, but we plan to extend it to distributed systems in the near future. Having this in mind, a special attention has been put on the design in order to facilitate the transition to a distributed version. For that future scenario special object brokers middleware such as CORBA and J2EE could be of great utility, specially the versions for real-time systems [23].

In the rest of the paper, the adaptable aspect of the project is first explained describing the aim of adaptation and the strategy to achieve it. The next section describes the adaptation mechanisms and the associated active object model. Then the mode of description and the needed information are shown, and finally the prototype is outlined and some results and conclusions are given.

2 Adaptation Principles

Any system is assumed to apply a best effort strategy in order to maintain dependability, satisfying expected user's needs. Nevertheless, the point of view of this project assumes that adaptation starts further, when some needs can not be satisfied. Therefore some decisions should be taken to keep the global satisfaction as high as possible, i.e., to maximize the global satisfaction according to various criteria. In this work adaptation means that the system modifies its behavior, i.e. the way services are provided, according to its context in order to reach a more suitable and satisfactory working point. The satisfaction is assessed using criteria that are gathered together to form the Quality of Service (QoS) provided by the system. In this sense, adaptation improves dependability by increasing the global QoS of the system.

Implementing adaptation requires a major choice of architecture to be done first. With a *non-intrusive* type of QoS management, adaptations are driven by the application execution context following two main ways:

- The operating system (OS) masters adaptation. Indeed, any OS manages the QoS, for instance by distributing CPU resources among applications, therefore only some additional functions are needed. A main advantage of this solution is that applications don't need to be modified and all of them running in the same context (OS, computer, etc.) benefit from the same services.
- A middleware layer between the application and the operating system manages the QoS. Via a privileged relationship with the OS, the middleware manages resources according to applications needs. Compared with the previous solution, this one has the additional advantage of greater portability because the middleware can be implemented over several OS.

By the other way, using an *intrusive* type of QoS management, applications themselves are involved in the adaptation process. The obvious drawback of this approach is that applications must be specifically developed to that purpose. Nevertheless, a great interest for this approach comes from the fact that only the application has the knowledge required to manage resources at best. For instance a video application displaying 12 frames per second instead of 25 to reduce resource use has still an acceptable QoS, while an OS that would divide by two the available resources of the application would be a worst solution. So, in order to use resources in the best possible way, the knowledge objects have about themselves is a crucial and essential factor. This is one of the reasons supporting the choice of an intrusive architecture. Moreover, this solution deals with another major design concern: preserving the autonomous and independent nature of objects. In this sense, adaptation decisions are taken by service providers (servers) and hidden to clients requesting the services.

In simple terms, the project strategy to improve dependability is to optimize a function measuring the provided QoS level. This usual problem is NP-Hard [16] and can not be optimized dynamically in a reasonable time. So we have chosen a heuristic approach to find a solution. Indeed, this approach is acceptable because, due to unknown event arrival distribution, any optimum is ephemeral and should be re-assessed for every new event arrival. Obviously, the result of partial optimal decisions is not optimal; therefore in this context the global policy can not be optimal. Conversely, real-time systems that ensure the respect of all time constraints require all events and their arrival distributions be known, thus a static analysis may produce an optimal predefined execution strategy. In the ARTO project, a partial static analysis of the application is done to ensure that critical events will meet their deadline. The number of critical events should remain as small as possible to allow a lightweight static analysis for some worst cases only.

3 Adaptation Mechanisms

In order to have adaptation capabilities objects in ARTO have degrees of freedom. They make decisions according to context and within the limits defined by their degrees of freedom. Adaptation capabilities are obtained by using very general and simple mechanisms that may be also parameterized.

3.1 General Structure of an Application

Applications are composed of active objects that communicate by *asynchronous* or *explicit wait for reply* messages only (Fig. 1). Each object owns at least two *light processes* (threads). One of them, called the *controller,* controls the object behavior and makes all *local* decisions. It manages the incoming messages queue, filters messages to avoid access conflicts, process executable messages according to object state, selects from the executable messages the one that must be executed, etc. The other threads execute object methods according to *controller* decisions. Intra object concurrency is necessary because high priority requests must preempt lower priority thread executions.

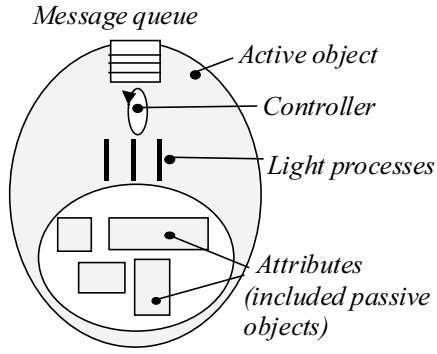


Fig. 1. *Active object*

The application's behavior shouldn't be left to the results of independent local decisions at object level because optimal sharing of resources such as processor time or memory requires decisions be made in a centralized fashion. So, a two levels decisions strategy was conceived to deal with this issue [7]. Instead of having a specific object in charge of centralized decisions, our solution let objects themselves make global decisions on shared resources collectively, based on global information. At *global* level all objects necessarily apply the same strategy, but at local level each one applies its own particular one. In practice, each application owns a root object that gives access to a shared data zone, as for example the scheduling of tasks, that is common to all applications. Objects wanting to execute a global procedure must reserve the access to the shared zone. Although the current version of ARTO prototype is mono application, this common zone is managed as if it were shared by many applications, therefore no reference but only operating systems identifiers such as *process ids* are stored. Our two levels approach for adaptation decisions was also a good way to preserve the distributed nature of the object approach [7]. Objects are not allowed to know the internals of other objects.

In short, ARTO active objects are atoms each one having a portion of autonomy and responsibility in the resulting application's behavior, driven by adaptation needs. The QoS management is intrusive at application level but it is also finer at object level. This allows to profit of all possible adaptations. Concerning the relationship with the operating system, objects fully manage processes and replace the standard manager.

3.2 Service Request

3.2.1 Message

Messages in ARTO are requests for services that carry functional and non-functional specifications. A server that receives a request decides how to provide the service. A basic service provision is to execute a method and to return a response if needed: in this case the message is associated with a method execution. In the general case, for each requested service (i.e., message) the server has a set of possible operations: it can refuse the message immediately, put it in a waiting state, select from the available methods the most suited one in the current context, delegate the request to another object, etc. Some operations are executed automatically according to the context, for instance to leave in the queue a message that is not ready, while others are specified using *actions*, for instance to *refuse* or to *delegate* a message. Therefore, at object level, a message is associated to a set of methods and actions:

$$\text{Message} \rightarrow \{ \ldots, \text{method}_i, \ldots, \text{action}_j, \ldots \}$$

At run time, a method or an action in the set will be selected according to the execution context and the decision policy. This correspondence is not rigid and may change due to context or object history. Each method has its own properties in relation to the resources it needs and/or its resulting qualities. If time is scarce, a lower time consuming method with a lower result quality will be executed, but with more available time an algorithm producing higher quality may be executed.

In an emergency situation, lower priority requests are eliminated to leave place to more important ones. Therefore, message *importance* is one of the first QoS parameters to take into account when making decisions. The values we defined for importance are: *low*, *normal*, *high*, *guarantee* and *critical*. They are used to decide which methods will be scheduled and which will not, when system is overloaded. Critical messages are executed at the highest priority to the detriment of all other messages, even those that have been guaranteed.

3.2.2 Negotiation

In a general QoS model, service's QoS levels are set by negotiations between clients and servers that result in *contracts* that are meant to be respected during all their period of validity. Nevertheless, in order to reduce negotiation overheads, messages with explicit request for *guarantee* of execution completion before deadline are the only ones that are negotiated. The server must give or deny its guarantee agreement, and the client must decide knowing the facts: when the guarantee is refused, the client must take an alternative action. For all other messages, the client is supposed to accept the server's policy and is only aware of the service results.

3.3 Application Modes

Application *modes* correspond to particular system or application situations, such as *Initialization*, *Normal*, *Take off*, *Panic*, *Stopped*, … In fact the mode is a kind of a global state but in order to avoid confusion, in the model the word *state* is reserved to active objects. At any given time, an application and all its objects are in the same mode. According to the mode every object has a mapping table linking messages to methods and actions: *message* → { *methods/actions* }. This table is automatically commuted when the application mode changes. This way a message that is executed in the *Normal* mode may be not executable in the mode *Panic*. The change of mode can be realized by any of the objects according to application-specific criteria. For instance, a lack of response may lead to some *Panic* mode, an anomaly may lead to a *Stop* mode, or just a change of operational phase may conduct to a *Take off* mode. This mechanism is a type of "hard" adaptation because once defined, no flexibility is possible.

3.4 Global Level Decisions

The global level is common to all objects, and its degrees of freedom for adaptation are a) the choice of a message to execute and b) the tasks execution order. The problem to face is to schedule the execution of a task list where each task is

characterized by a deadline, an importance, a precision, etc. For simplicity we call *task* the association of a message (and its method set) to a thread. Tasks are defined at local level by objects and submitted to execution at global level where all of them are treated in a common way.

3.4.1 Task Planning Policy

For the soft real-time domain, the selected heuristic favors the timing criteria when optimizing the function that evaluates the QoS. In this document it is only presented the general scheduling principle (more details in [7] [8]). The EDF algorithm [17] selects the task with the closest deadline for the next execution. EDF is a simple and efficient algorithm that produces an optimal scheduling if all the tasks can be executed. Nevertheless, when the system is overloaded EDF is not determinist, i.e., the tasks that will not meet their deadline are unknown. To deal with this problem, tasks with higher importance are privileged (this policy doesn't minimize deadline faults). This modified EDF algorithm guarantees that a message will not be eliminated to favor a lower important one, therefore non-determinism is reduced to messages with the same importance.

Each method is characterized by its *worst case execution time*, WCET, and the schedule is built with the smallest WCET of the methods associated with the messages, i.e., with the methods that produce the lowest quality[1]. The main objective is to reduce number of faults, i.e. to execute the maximum number of tasks. When a task doesn't fit in the planning, an overload situation is assumed and messages with lower importance are deferred or eliminated from the planning, allowing the scheduling of a more important one. Nevertheless, when CPU time is freed (at the end of a method's execution) deferred messages are rescheduled if possible. To accept a new task in a schedule of n task, the maximal complexity of the heuristic is $\boldsymbol{O}(n^2)$ while the general problem of temporal faults minimization is NP-complete. It means that the number of faults in our scheduling strategy is not minimal but in practice when the system is overloaded several requests are pending and the CPU is always busy, so the lost is minimal.

To avoid execution beyond the scheduled duration, an event is triggered at the expected end of each task. Then the scheduler either continues the method when there is free time or schedules the next task in the planning. This is the way how CPU load monitoring takes place and since everything is scheduled by ARTO controllers and since metaobjects catch all events, the load is always known.

3.4.2 Selection Policy for Task Execution

Let us consider two tasks in the planning, t_1 and t_2, each one with three possible qualities, low_i, $medium_i$ and $high_i$. Let us also consider that only three combinations are possible due to timing reasons: ($medium_1$, $medium_2$), (low_1 , $high_2$), and ($high_1$, low_2). Even without adding quantitative aspects, it is clear that the search for an optimal combination may be very time consuming. This is the reason for the use of an "optimistic" heuristic: task t_1 is executed with its highest possible quality ($high_1$) expecting that task t_2 , which is planned to be executed with the quality low_2 , be finally executed with a higher quality. This approach is reasonable, because the

[1] We assume that quality is proportional to method's execution time.

schedule is built with the pessimistic worst case time (WCET) and many executions will not use all of their reserved time. In fact, at the time when a task is to be executed, the effective available CPU time is assessed and the longest possible method is selected (among those associated with the message). This basic approach has the advantage of being a predictable base for the setting of local policies.

3.5 Local Level Decisions

Local decisions are made at object level. The first local decision is the composition of the methods and actions set { ..., $method_i$, ..., $action_j$, ... } to be associated with the message, where any element can provide the requested service. The second local decision is the selection of the messages to execute (the choice of a method to execute is made at global level).

3.5.1 Initial Methods and Actions Set

A message is a request for a service, and each object holds, for each mode, the different ways it may respond to the request, in the form of *Message* → { *methods* | *actions* } associations. { *methods* | *actions* } represents the set of possible responses and the associations are stored into a table that is initialized in function of the mode and then modified according to needs. Currently, only one action (among 20) can be associated with the message, for example to *refuse*, to *change of mode*, to *change of global policy*, … Nevertheless, the methods set can be built in many ways, some of them are presented below.

3.5.2 Composition of the Method Set

3.5.2.1 Methods with Different Qos Characteristics

The set is composed of methods with different duration, precision and in general different QoS (*time polymorphic invocation* [29]). At the start of the service execution, the method that best provides the service within the context will be selected. It is obvious that the more CPU time is available, the better the result quality is.

It is possible that more than one method provide the same service, for instance several algorithms that find the location of an object in a camera image, so in this case the methods set is built directly. But the power of this mechanism comes out from the possibility of adding extra artificial methods. Let us consider an example with two methods m_i and m_j with execution times of d_i and d_j respectively, which return a value. In case of overload the one with the smallest execution time will be executed. Let us add into the set a new method m_k with duration d_k, such as $d_k << d_i$ and $d_k << d_j$, which returns the average value of the last n requests (or any other fast solution). If the system is under a hard overload, the very fast method m_k will be executed, returning an estimated value. This value is supposed to be better than no result at all. In the example of object location in a camera image, the previous position or an estimated one may be returned.

3.5.2.2 Set with Decision Rules

Execution rules such as "m_1 and m_2 must be executed alternately" [9] are worked out easily by appropriate set compositions:

- "To execute m_1 or m_2 with a maximum of n executions of m_2": both methods are placed in the set while the constraint is respected. But only m_1 is in the set when violated.
- "m_1 must be executed at least one request out of two": m_1 is always in the set. If m_1 has been executed, an artificial method $m_\varnothing$ with execution time zero is added. When the system is overloaded, $m_\varnothing$ is executed but next time it will not be in the methods set.
- "m_1 deadline can be violated once out of two": m_1 is always in the set. When the constraint has been violated a *guarantee* is requested. When accepted, the server ensures the message execution (except when a *critical* message causes a violation).

3.5.2.3 Task-Pair

Execution times may be difficult to measure as for example when a reply from another object is expected. This problem may be faced by scheduling two methods [28]: one *normal* method m_n with an optimistic duration smaller than the worst case duration and an *exception* method m_e with a very small duration that will be executed if the execution of method m_n doesn't end before its deadline. This mechanism can be realized by putting m_n in the methods set and by planning a second artificial message with *guarantee* request, with method m_e in its methods set. At the normal end of m_n the artificial message is destroyed, otherwise the execution of m_e starts at the last possible time that guarantees its execution (*last chance moment*). As one of its first tasks, m_e destroys the unfinished message m_n.

In conclusion the provided mechanisms allow any combination of set implementation imagined to deal with the problem in hand. A chosen combination must consider not only the global decision policy which is well known and fixed but also the local policy and all special cases such as the guarantee rejection. The experience with the prototype has shown a very good relation between *expression power* to *execution time*.

3.5.3 Message Selection

An object controller is started whenever an event affecting the object allows the treatment of a new message. For each message, the methods set go through filters that eliminate the methods according to different criteria (incompatible with those in execution, not executable in the current object state, don't produce the requested QoS, etc.). Messages are processed as shown in Fig. 2. The process starts with the *Message* → { *method* } mapping which depends on the application mode, and then continues applying the filters that eliminate unauthorized methods.

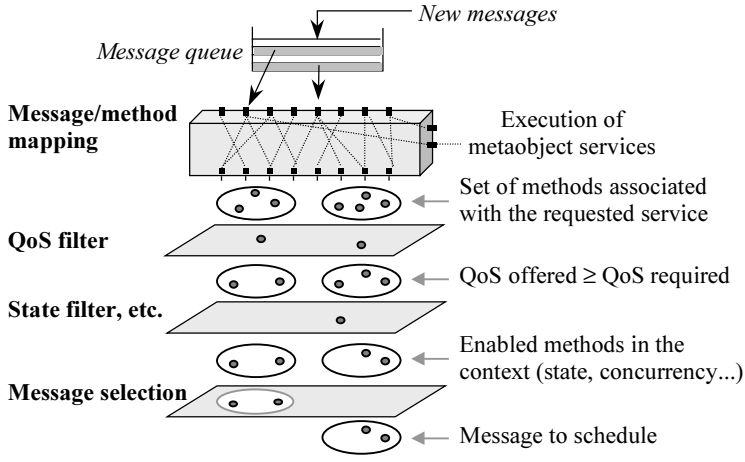


Fig. 2. *Message treatment*

Adaptation capabilities are increased with this filtering procedure because it allows to put into the set methods that are executable in different contexts. According to the context (object state, methods in execution), some methods are eliminated. For instance let us consider two methods, one of them being executable in the *running* state and the other in the *stopped* state: according to the actual object state one of them will be eliminated.

All messages in the queue are treated and only executable ones remain after filtering, i.e., those with non-empty methods set. Then objects apply their local policy to select one message. The basic policies that are provided are: earliest deadline first message selection (EDF), most important first, most important first with a thread reserved for *critical* messages, and selection according to message arrival time (FIFO). Of course it is possible to add other policies according to particular needs, for instance in function of a *message stamps* to preserve an ordered processing of messages coming from the same object.

4 Description: A Quick Look

To describe the adaptation policies is not the slightest problem, even if the choice of the UML stands out [30]. UML provides extension mechanisms to create new notions derived from existing ones (stereotypes), to add properties (tagged values) and constraints to any model element. The extensions relating to a specific area

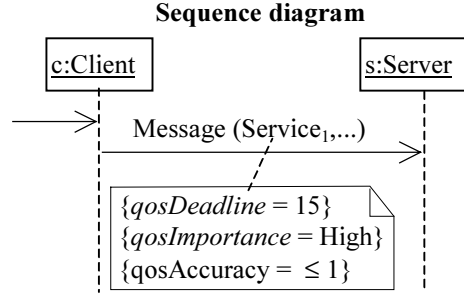


Fig. 3a. *QoS description in the UML*

are usually gathered into a profile, for instance the « Schedulability, Performance and Time » profile [22].

The application is modeled in a standard way, but adornments supply additional information that will be used to generate the code (see [27]). Figures 3a and 3b show adornments (italic for mandatory). Tagged values have the form {*tagName* = *value*}. The *value* is any string with syntax to be defined, for instance Fig. 3a shows a bound for accuracy and Fig. 3b shows a collection of modes. In this description, values are relative to a given execution context, for instance the duration of methods depends on the OS. In the Figures 3a and 3b, descriptions are gathered into UML notes but in practice, such a process is not viable because it is too heavier for the user: a tool to assist and control the inputs is necessary.

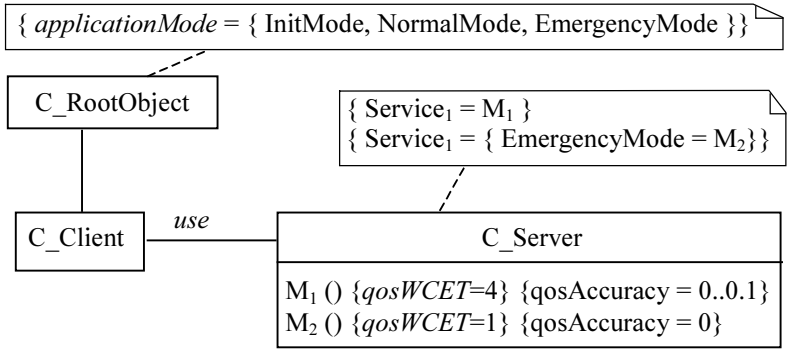


Fig. 3b. *A UML Class diagram with QoS adornments*

5 Implementation

A main objective in choosing the architecture was to preserve the qualities of *portability* and *reusability* of object-oriented applications, therefore mixing context dependent and context independent information is not desirable. On the other hand, adaptable objects must hold knowledge about themselves to modify their behavior. These requirements lead naturally to a *reflective architecture* [26] with a *base level* for the object and a *metalevel* to control its behavior.

5.1 Metaobject Architecture

The metalevel is implemented by using metaobjects. Every object (at base level) is controlled by one metaobject, that holds information about its object (behavior, current state, etc.), and deals with all its execution context and adaptation aspects (schedu-ling, concurrency, etc.) (see Fig. 4). The reified mechanisms of the language are message sending, objects creation and destroying.

In ARTO, objects are completely encapsulated and only know their metaobject: all the communications outside the object go through metaobjects. Unlike open C++ [5], messages don't go directly to the receiver's metaobject but sent to their own metaobject (Fig. 4). This last one may decide to send the message to the planned receiver, to another object or even not to send the message (for instance returning a default value). At the receiver's metaobject side, the message is generally mapped to an object method that will be executed, but it could be delegated or refused. The architecture was designed having in mind distributed execution contexts: metaobjects intercept all messages and send them to distant objects via proxies and a naming service, so objects don't know where the receiver is (local or distant). Next, the interface of active objects is a list of service names that are mapped to methods at the receiver side, and parameters are passed by value only. Thus, the same service can be requested to servers of any type.

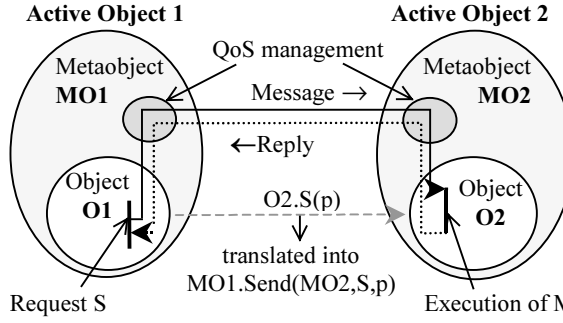


Fig. 4. *Reification of communication*

This architecture is not truly reflective in the sense of KRS [18], but as a great advantage it is easily implemented in current O-O languages. As another additional advantage, it naturally provides a separation between functional and non-functional concerns, thus joining aspect oriented programming [15].

5.2 Metaobject Structure

An ARTO metaobject is a composition of elements dealing with basic requirements such as scheduling, concurrency constraints or adaptation mechanisms. As these elements may be optional and may have several versions, a way to integrate their contribution is required. A solution would have been to use meta-metaobjects to modify the behavior of metaobjects as in ABCL/R2 [12], or even to compose metaobjects supplying each one a precise service [21]. But given the fact that each metaobject describes the features and properties of one object running within one context (states and transitions, execution duration, etc.), and also due to performance reasons, a different alternative was chosen. The ARTO framework is parameterizable, and the code is automatically generated with the required parameters.

Standard mechanisms are provided at several levels to adapt the code to be generated. First the metaobject type is chosen according to application needs. Next there is a set of empty methods to be redefined in metaobject descendants, typically to catch events such as "out of deadline", and to execute code when passing through particular points such as *before* or *after* a method execution. Moreover, standard methods implementing policies such as local decision can be redefined. At last, it is possible to redefine any metaobject function, for instance the scheduling function (this is an extreme solution).

5.3 Prototype

The prototype is a C++ program running on Windows NT, implementing most of the functions of the ARTO model. To ensure predictability, there are no dynamic memory allocations except at object creation time, therefore values such as the maximum number of messages or queue length should be bounded. To avoid time leaks, all execution times are taken into account. Consequently, when an application starts, experiments (execution of a null method) are done to measure the time used by the metaobject machinery in the current execution context. Metaobject and object tasks are scheduled in the same way. However, the current task is interrupted and the metaobject task executed, only if there is no risk to violate deadline. In view of the supplied services, a very careful design has been done to avoid increasing dramatically the execution time. In particular, as many threads may co-exist and furthermore sharing data, it is important to reduce context switches. The OS should provide dynamic priorities and must always execute the thread with the highest priority first.

From the user point of view, the framework is made of a set of basic metaobject classes, each one providing different services and from which user metaobject classes inherit. Besides the numerous adornments mentioned earlier, additional information is needed such as the number of object threads or the objects to automatically notify when an object modifies its state. Practically, a development tool more sophisticated than simply adding comments on UML models (as in Figure 3b) was necessary. The details of the development tool are out the context of this paper.

5.4 Example of Results

We have made a workbench to test and evaluate the performances of the ARTO framework. Messages are sent to objects according to the selected arrival law, and we trace their path. Since messages are queued, the first observation is that the arrival law has no effect when the system is overloaded (we use uniform random arrival). On the other hand, among the many parameters some are more important: the size of the input queue, the posting policy (wait if full or cancel), the number of threads, the global policy (e.g., rescheduling of cancelled tasks), the percentage of critical and high importance tasks, etc.

The curves in Figure 5 show experiment results for a representative set of parameters values, e.g., 5 threads per object, 10% of critical tasks and 30% or high importance tasks. Here is show a few examples of results, because a deeper analysis would require varying all the parameters.

Figure 5 shows the average percentage of faults and the quality of the result (the ratio between the quality resulting from the executed methods over the quality of the best quality methods) in function of the processor load (duration of all methods with best quality over the available CPU time). Messages are mapped to 1 and to 3 methods with duration *d, d/2* and *d*/8. The local decision for the message choice, when a thread is freed, is either the closest deadline first (*EDF*) or the most important first (+*imp*). If enough threads are available all messages are scheduled. As expected, the addition of shorter methods decreases the number of faults but also the quality of

the results. This reduction depends on the local policy: *EDF* causes less faults, but looking deeper at results it causes more faults among important messages than the *+imp* policy. This result was expected since the "most important" policy always tries to execute the most important message first, whatever its deadline. A synthesis of the « *quality* » and « *faults by importance* » dimensions could lead to either *EDF* or *+imp* policy according to the weight of each criterion.

5.5 Performances

Obviously the ARTO framework introduces overheads but these are acceptable when considering the supplied services, especially those related to adaptation. On the other hand, the ARTO framework has been recently compared with real time tools such as Rhapsody [13] and Rose RT [24] implementing an industrial example [25]. As a representative result, the response to an event involving synchronous and

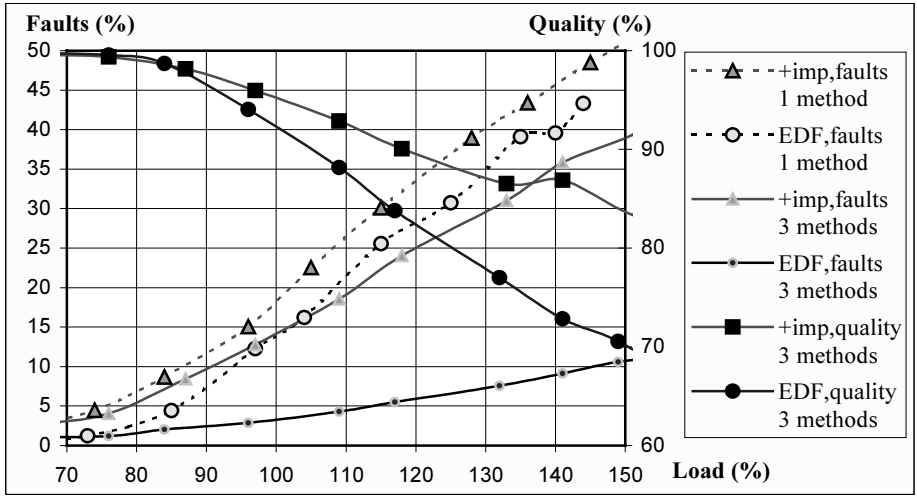


Fig. 5. *Percentage of faults and QoS according to the processor load (Processor time requested / available time)*

asynchronous messages as well as diffusion to several objects takes 270µs in ARTO while it takes 142µs (average) with the other tools. In the same context, sending a message and then in the receiver replying a response takes 60µs with ARTOs but 44µs with the other tools. Nevertheless, the additional features of ARTO are scheduling, adaptation capabilities, QoS management, concurrency control, etc. The extra time in ARTO is the price to pay for its additional services, and of course the interest depends on the needs.

6 Related Works

This project addresses problems similar to those focusing on fault tolerance, graceful degradation and QoS management, all of them aiming to increase system

dependability. A main design concern was to define an architecture focused around adaptation [6]. The result is a framework providing a general adaptation mechanism that supports some existing real-time mechanisms such as *task-pair* [28], decision rules [9], and polymorphism [29]. Numerous works about « distributed execution + QoS + adaptation + middleware » such as [1] and [3], use similar techniques to modify the behavior according to the available CPU time or bandwidth. Nevertheless, due to their intrusive nature, ARTOs provide fine-grain application tunability. Moreover, they directly provide object independence. On the other hand, each object applies its own policy and there is no middleware or any other layer that forces some policy such as the *Resource Manager* in [4]. Finally, an important requirement common to most works is to describe the adaptation policy separately.

In some related works the QoS is fixed by the application user and not by the application itself using some kind of dialog [16]. In other cases, the adaptation policy is frozen or described in a script that will be exploited by the adaptation layer [3]. In these approaches, requests do not specify the QoS criteria and the service provider decides the QoS level that will be supplied to the client. This means that these approaches view the QoS as a parameter of the execution context and not as an additional dimension to be managed by the service requester. In the proposed model, unlike most of the related works, the service request is a whole including functional and non-functional aspects. Moreover, the client does not define "how to fulfill the needs" but "what it needs", and the server decides the best way to provide the service.

In relation to reflective architectures, they are not new in dependability [10], neither in real-time domain [11] [12] [19]. Sometimes several metalevels are used but in ARTOs this is not useful: metaobjects hold several objects each one implementing some aspects, for instance scheduling. Metaobject-based architectures provide a very simple way to separate concerns [15]: aspects are introduced catching events such as message sending. In ARTOs, the behavior associated with an event is set by using parameters or by method redefinition. Apart from object creation and message sending which are object responsibilities, all specific code is in the metaobjects.

The general trend of current related works is to adapt the application to its execution context in a transparent way via non-intrusive management catching requests for resources. The proposed approach is one of the very few to choose an intrusive management with applications having their own policy according to available resources, and seems to be the only one that provides decisions capabilities at object level. These choices lead to a decentralized architecture with actually autonomous objects: the "A" in ARTO means "Adaptable" but also "Active" and "Autonomous".

In relation to a future distributed version of ARTO, some middleware object brokers may play an important role, as for example CORBA or J2EE. Nevertheless, it must be noted that in the CORBA model, proxies (or any other intermediary) aim to control object communication only. In our model, meta-objects aim to control objects, i.e., to change their behavior according to the context. Meta-objects hold data about objects, not about communication. This is a main difference, and even in a distributed context, this difference will remain.

7 Conclusions

This project takes QoS management as the way of dealing with non functional aspects affecting dependability. The optimization of QoS triggers adaptation mechanisms that induce a reduction of the temporal faults and a graceful degradation. This result has been achieved without sacrificing object-oriented approach principles: at local level object decisions are specific and each object keeps its autonomy, while at global level the decisions are common to optimize resource utilization. The metalevel architecture separates context dependent concerns from non-dependent ones, thus reducing complexity and ensuring portability, reusability and flexibility. Finally, the development process is consistent with best software engineering approaches: description with a modeling language aided with a checking tool, automatic code generation freeing developers from the burden of low-level tasks.

From performance point of view, execution times used to send messages and to manage QoS including scheduling is close to similar tools that generate code from UML descriptions. However, these tools do not deal with time constraints and QoS, therefore they don't provide neither scheduling nor adaptation. Results show that adaptation overload is not too expensive. Many extensions are possible in the future, among which distributed execution, that will increase dramatically the adaptation capabilities, and management of other QoS dimensions such as bandwidth or memory. All of these possible enhancements will, in fact, improve dependability.

References

1. Abdelzaher T., « QoS Adaptation in Real-Time Systems », PhD, Michigan, 1999
2. Caplat G., *Modélisation Cognitive*, PPUR ed., ISBN 2-88074-495-4
3. Chang F . and Karamcheti V., « Automatic Configuration and Run-time Adaptation of Distributed Applications «, IEEE Symp. HPDC, 2000, p.11–20
4. Chatterjee S., & al. « Modeling Applications for Adaptive QoS-based Resource Management », *Proc. IEEE Workshop HASE'97*, 1997
5. Chiba S., Masuda T., « Designing an Extensible Distributed Language with a Meta-Level Architecture », *ECOOP'93, LNCS 707*, 1993, p.482–501
6. Clark R., Jensen E. D. & al., « An adaptive, Distributed Airborne Tracking System », *Workshop on Parallel and Distributed RT Systems* (www.real-time.org), 1999
7. Contreras J.L., Sourrouille J.L., « A Scheduling Strategy to Preserve Object Autonomy », *WRTP'00*, 2000, p.153–159
8. Contreras J. L., Sourrouille J. L., « A Framework for QoS Management », TOOLS'39, USA, IEEE press, 2001, p.183–193
9. Delacroix J., « Towards a stable Earliest Deadline scheduling algorithm », *Real-Time Systems Journal*, Vol.10(3), 1996, p.263–291
10. Fabre J.C, & al., "Reflective fault-tolerant systems: from design to validation", Rapport LAAS N° 01007, Contrat CNET N° ST.CNET/DTL/ASR/97049/DT, Janvier 2001.
11. Honda Y., Tokoro M., « Soft Real-Time Programming through Reflection », *IMSA Workshop Reflection and Meta-level Achitecture*,1992
12. Honda Y, Tokoro M., « Reflection and Time-Dependent Computing: Experiences with the R2 Architecture », *Sony C.S. Lab -TR-93-017, 1993*
13. I-Logix, http://www.ilogix.com
14. ISO - Information Technology – Quality of Service – Guide to Methods and Mechanisms – ISO/IEC 13243 Draft 1.0 | Project JTC1 21.57, 15/10/1997

15. Kiczales G., & al., « Aspect-Oriented Programming », Proc. ECOOP'97, LNCS 1241, Springer Verlag, 1997, p.220–242
16. Lee C., Siewiorek D., « An Approach for QoS Management », CMU-CS-98-165, 1998
17. Liu C.L. , Layland J.W., « Scheduling algorithms for multiprogramming in a hard real time environment », *Journal of the ACM*, Vol 20(1), 1973, p.46–61
18. Maes P., « Concepts and experiments in Computational Reflection », *OOPSLA'87*, 1987, p.147–155
19. Matsuoka S., Watanabe B., Yonezawa A., « Hybrid Group Reflective Architecture for O.O. Concurrent Reflective Programming », *ECOOP'91*, LNCS 512, p.231–250
20. De May V., « Visual Composition of Software Application », In Nierstrasz and Tsichritzis eds: *Object-Oriented Software Composition*, 1995, p-276–303.
21. Mulet P., Malenfant J., Cointe P., « Towards a Methodology for Explicit Composition of MetaObjects », *OOPSLA'95*, 1995, p.316–330
22. OMG, "Schedulability, Performance and Time" Reply to the RFP of the OMG, ARTiSAN Software Tools, I-Logix, Rational Software Corp., Telelogic AB, TimeSys Corporation, Tri-Pacific Software (2000)
23. OMG, "Real-Time Corba 2.0: Dynamic Scheduling Specification", September 2001.
24. Rational, http://www.rational.com, Smith B., « Reflection and Semantics in a Procedural Language », MIT, TR272, 1982
25. Schneider Electric, Technical Report, http://prisma.insa-lyon.fr/pages_personnelles/sou/Jls-fr/Rapports/RT-01-2003, grant INSAVALOR number 008686
26. Smith B., « Reflection and Semantics in a Procedural Language », MIT, TR272,1982
27. Sourrouille J.L., « UML Behavior: Inheritance and Implementation in Current Object-Oriented Languages », *UML'99*, LNCS 1723, 1999, p.457–472
28. Streinch H., « Task Pair Scheduling: an Approach for Dynamic Real-Time Sytems », *Journal of Mini& Microcomputers*, Vol.17(2), 1995, p.77–83
29. Takashio K., Tokoro M., « DROL: an object-oriented programming language for distributed real-time systems », *OOPSLA*, *ACM Sigplan*, 1992, p.276–294
30. UML - OMG Unified Modeling Language Specification (Action Semantics), V1.4, 2002

A Genetic Algorithm for Fault-Tolerant System Design

Klaus Echtle and Irene Eusgeld

FB 5, ICB / Informatik
University of Duisburg-Essen
{echtle, eusgeld}@informatik.uni-essen.de
http://dc.informatik.uni-essen.de

Abstract. Due to high cost, considerable complexity and long design cycles of fault-tolerant systems, a (partial) automation of the design process becomes attractive. This paper presents an approach to automatic design by use of a genetic algorithm. Unlike typical genetic algorithms the individuals (which represent a fault-tolerant system structure each) are represented by a non-cyclic graph rather than a string. Special crossover and mutation operations modify the individuals such that reasonable fault-tolerant systems are likely to be generated. The biggest problem in using genetic algorithms lies in the definition of an appropriate fitness function one has to apply to each of the many generated individuals. A complete analysis of a single fault-tolerant system would comprise time-consuming fault-tree analysis, reachability analysis of the state space, etc. A substantial speed-up by orders of magnitude has been achieved by the development of a completely new fitness function, which can be considered as a simplified reachability analysis. For many fault tolerance techniques it visits each component only once (or very few times in the case of mechanisms like rollback, retry etc.).

Keywords: Fault tolerance, fault model, genetic algorithm, fitness function, analysis of fault-tolerant behaviour.

1 Introduction

In the area of fault-tolerant computing several standard patterns like duplication and comparison, triplication and voting, rollback to checkpoints, reconfiguration, exception handling, coding, etc. are widely used. Nevertheless, these well-established techniques cause considerable cost in terms of both redundant components and computation time. For this reason designers try to adapt fault tolerance techniques to their system as much as possible in order to reduce the overhead, as can be seen from the following examples:

- Wait phases of elements in a multiprocessor system are used to perform redundant computations “for free”.
- Redundant nodes may share spare components.
- The limited gradient of sensor and actuator signals is used to allow for less costly fault tolerance techniques which do not prevent wrong outputs for very short durations.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 197–213, 2003.

Such properties contribute to the complexity of fault-tolerant system design. Finally, the standard patterns of fault tolerance are modified in many respects. Each fault-tolerant system has its own individual methods. This saves redundant resources, but leads to a drastic increase of the design space, which cannot be completely covered by humans. Advantageous solutions might be overlooked. For this reason, one can think of an automatic design or of a partly automatic design of fault tolerance techniques.

Besides better search for efficient solutions there is a second motivation for automation. Humans tend to think in examples: "If this fault happens, then take that countermeasure." When developing ideas designers seem to prefer fault cases they consider as typical for some systems. Other hidden faults, however, may slip through the designed net of countermeasures, thus leading to system failure. This undesired focus on virtually typical scenarios can be avoided by an automatic design process, which is neutral in this respect.

The third reason for automatic design lies in the speed-up of the design process. The development of fault-tolerant systems for safety-critical applications has turned out to require particular effort and takes a long time. The first phases (not later phases like certification, for example) have the potential of being shortened when a large design space is explored automatically and efficient fault-tolerant structures are generated.

In the past, automatic support to the design of fault-tolerant systems has been applied mainly in the following ways:

- *Optimization* with respect to parameters of a given fault tolerance technique: Typically the degree (e.g. the number of nodes) or the density (e.g. the frequency of recovery points to establish) are calculated from dependability requirements and properties of the components.
- *Choice* among given structures: By modelling different alternatives and respective analysis one can obtain a good decision basis to select one alternative. A rich variety of very precise assessment methods has been developed during the recent decades. However, these methods are time-consuming to both man and machine and, moreover, do not generate the models themselves.

The complete construction of a fault-tolerant system by some algorithm has been applied to few experiments only, not to real systems. Special fault-tolerant features of networks and VLSI chips have been developed using genetic algorithms. Most of this work uses the traditional representation of structures, namely binary strings, to encode network topologies [2, 4], tree-based networks [7] or integrated circuits [10]. Different types of systems, however, require more flexible structures. Other known approaches encode phenotypes as directed attributed graphs [e.g. 11], but do not address fault tolerance. As a very special case reconfigurable hardware has been designed by evolutionary methods [e.g. 12]. From the limitations of these experiments the necessity for further research towards more general approaches becomes clear.

By our work we aim at a fully automatic design, in particular with respect to the structure of a fault-tolerant system. The approach presented in this paper can be seen as a first major step in this direction. In the future it will be refined to cover more details. An interesting extension will cover peripheral devices of safety-critical systems (sensors, filters, AD-converters, DA-converters, power outputs, passivation

units, actuators, etc.). The automatic design of their fault tolerance techniques is of particular interest, because in this area individual solutions are typically preferred to standard fault tolerance techniques.

Exhaustive search in the design space is not feasible for an automatic design method. The number of potential solutions is by far too large – whether we consider the functional or the structural level. Moreover, the number of fault cases one has to check for each system is extremely high as well. Traditional methods of search and optimization are too slow in finding a solution in a complex search space, even when implemented in supercomputers. All realistic applications of such methods suffer from combinatorial explosion. In these cases, iterative constructive approaches are suited much better. One of them is the imitation of natural evolution by probabilistic algorithms, which implement "DNA-like" operations. In many areas *genetic algorithms* have demonstrated their power in finding practical solutions to complex optimization problems. In particular, design problems from the engineering area have been solved successfully.

Fault tolerance seems to be difficult for genetic algorithms. Redundant structures do not match with linear "DNA strings" unless they are coded in a complicated way. Our solution (see Section 2) expresses the structure "directly" by an appropriate graph. An even more severe problem caused by the application of genetic algorithms to fault tolerance lies in the definition of the crossover operation (see Section 5) and the fitness function. The latter is applied to each individual system in each generation to select the fittest. For this reason the fitness function must execute very fast on one side, and assess accurately the achieved degree of fault tolerance on the other. We defined a special fitness function (see Section 4) which can be seen as a good compromise between the two contradicting requirements. The fitness function needs a special fault model (see Section 3), where the level of details has been chosen carefully with respect to the compromise just mentioned. In all, the fitness function can be characterized as a special type of a coarse one-pass reachability analysis (requiring additional passes in the case of repeated computations).

This paper is organized as follows: First the level of design (Section 2) and the fault model (Section 3) are determined from the fault tolerance viewpoint on one hand, and genetic operations and the fitness function on the other (Sections 4 and 5). Examples are given in Section 6, results and conclusions are discussed in Section 7.

2 Level of Design

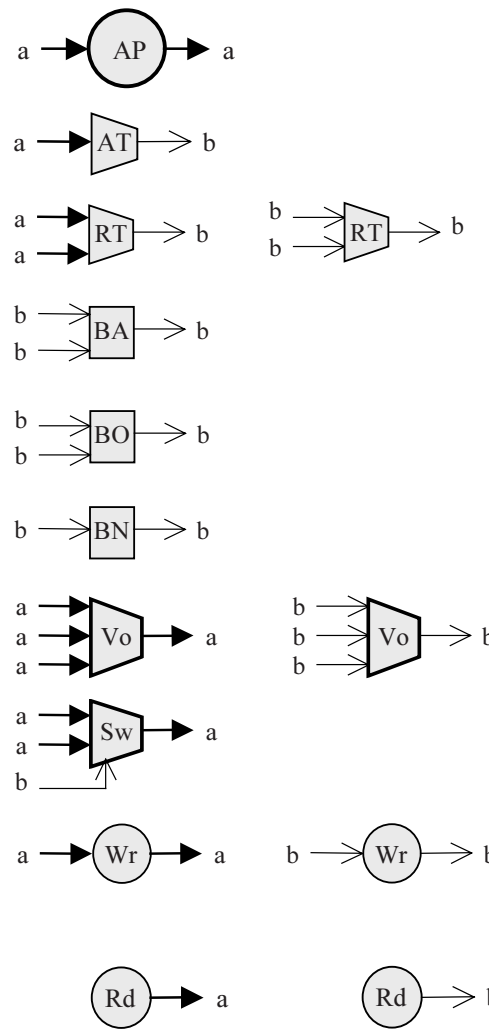
Our genetic algorithm treats fault-tolerant system structures as individuals to be processed by genetic operations. The first question to answer is the individual's degree of details. Too high design levels allow for simple fault tolerance mechanisms only. Too many details, on the other hand, will result in very slow optimization by the genetic algorithm (if it optimizes at all). Our compromise is a *fault tolerance graph* similar to the graphs in [5, 6]. This graph concentrates on the functionality of fault tolerance. Countermeasures against faults (whether implemented in hardware or in software) are expressed in much more detail than the rest of the system. Consequently, fault-tolerant decisions like rejection of erroneous information or voting among redundant information are in the focus of the genetic algorithm.

Nodes in a fault tolerance graph represent components. Directed edges express information flow between them. For the genetic algorithm the information itself is less important. However, the data type of the information plays an important role. Outputs of nodes are only connected to inputs of subsequent nodes if they are of identical data type. By this rule genetic operations are prevented from generating too many meaningless individuals. Currently we have defined the following data types:

- *application data* (a), the set of values is {initial, result1, result2}. The result values express the outcome of two different application functions in an abstract way.
- *binary test outcome* (b), the set of values is {true, false}.

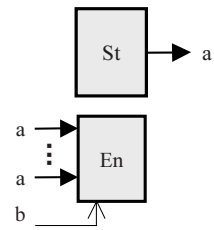
Additional data types can be defined easily. Nodes in a fault tolerance graph are of the following component types (The number and data types of inputs and data types of outputs can be seen from the respective component symbols. Some components are defined for different data types):

- *application processes* (AP), with full abstraction from the application program,
- *absolute test* units (AT), which assess incoming information as either faultfree or faulty,
- *relative test* units (RT), which compare 2 incoming values and decide on equality,
- Boolean *"and"* units (BA), to process the outputs of absolute and relative tests,
- Boolean *"or"* units (BO), to process the outputs of absolute and relative tests,
- Boolean *"not"* units (BN), to process the outputs of absolute and relative tests,
- *voters* (Vo), to chose the majority of 3 incoming values (if the majority exists),
- *switches* (Sw), to select one information out of two, depending on a boolean input,
- *writers* (Wr), to store information in memory, in particular for retry of processing after a previous processing has failed (in addition, the information is output to a successor component),
- *readers* (Rd), to read stored information from memory.



A memory access component is subdivided into a writer and a reader to keep the graph free of cycles. Conceptually, both nodes could be combined as a "memory component" regarding their function.
There are two more components *start* and *end*, which are helper nodes to complete the graph. They do not participate in evolution, and are taken as perfect (this means: always faultfree):

- a single node *start* (St) provides the initial input information to the system,
- a single node *end* (En) reads the final output information of the system (which can be redundant).



The end node can be defined with different number of inputs, with or without a boolean input, and with some further parameters. They express the service expected from the whole system. In other words, the end node represents the specification of the fault-tolerant system (see Section 3, and Table 3.2 for the choices of different specifications).

3 Fault Model

The fault model has to be designed in the light of the fitness function, which should neither be too coarse, nor too fine-grained. Generally a fault model has to cover two issues:

- *Fault location*:
  As in many other fault models we have chosen a simple binary fault model [e.g. 5]. It distinguishes the states *faultfree* and *faulty* for each component. The model subdivides the set of components into *single fault regions*, which are disjoint sets of components. A parameter F specifies the maximum number of single fault regions which components are assumed to be simultaneously faulty. Here, we simply take each component as a single fault region, because the reasoning about an appropriate abstraction for the genetic algorithm has resulted in the definition of components (which are nodes of the fault tolerance graph).
- *Malfunction* of a unit assumed to be faulty:
  This issue is much more difficult. The functions of faulty components can fail in many different ways, causing different patterns of interaction among faultfree components (in order to tolerate the fault). Nevertheless, the fitness function must execute fast and thus can only distinguish very few types of malfunctions. Hence, our challenging task consists of finding classes of malfunctions, which are most characteristic for faulty behaviour (including "tricky" faults) and, moreover, suit to a variety of different component types. Finally, we have decided the definition of three classes:
  - no result (for short: *none*, comprises fail-silent and fail omission),
  - *detectable* malfunctions (comprises code violations, timeout-violating delays, etc.),
  - *arbitrary* malfunctions (incl. generation of undetectably wrong results).

For each fault case the malfunctions are assigned to the outgoing edges of a component assumed to be faulty. This is simply done by extending the values of the datatypes accordingly:

- *application data* (a): {initial, result1, result2, none, detectable, arbitrary}.
- *binary test outcome* (b): {true, false, none, detectable, arbitrary}.
  Remark: In case of a binary test outcome "detectable" presupposes a redundant code to represent the value of the test outcome.

The value "none" is also taken when a faultfree component does not provide any result under some input conditions.

As can be seen from various statistics, arbitrary failures are rare compared to the benign cases "none" or "detectable". For this reason the state "faulty" is subdivided into "benign" and "malicious". For benign components the "arbitrary" malfunction is excluded, whereas malicious components may exhibit any malfunction. The fitness function counts the fault cases with weights in the following order (expressed by the so-called fault case index). This is also the sequence in which the computation of the fitness function proceeds:

– single fault of type benign (fault case index 1)
– single fault of type malicious (fault case index 2)
– double fault, both of type benign (fault case index 3)
– double fault, one benign, one malicious (fault case index 4)
– double fault, both of type malicious (fault case index 5)
– triple fault, all three of type benign, etc. (fault case index 6, etc.)

Formally, each fault case can be expressed by a function
$f : \{\text{components of the system}\} \rightarrow \{\text{faultfree, benign, malicious}\}$,
where the cardinality $|\{\text{component } x : f(x) = \text{benign or } f(x) = \text{malicious}\}| \leq F$.
Some components are taken as always perfect: f(*start*) = f(*end*) = faultfree. The implication holds: component x is of type Wr $\Rightarrow$ f(x) is faultfree.

The start node and the end node are always faultfree, because the system as a whole could never be fault-tolerant otherwise. Faults of writer components (Wr) are excluded for simplification. These faults are summarized by faults of the respective readers.

Fault-tolerant systems have to take countermeasures against faults. In our model their effect can be checked as follows: For each component the mapping of input values to output values is specified for the three fault states "faultfree", "benign" and "malicious" (see Table 3.1 for an example). According to this function the

Table 3. 1. Function of node AP

input	*output*
If application process is faultfree:	
none	none
initial	result1
result1 or result2	result2
detectable	detectable
arbitrary	arbitrary
If application process is benign:	
other than arbitrary	detectable
arbitrary	arbitrary
If application process is malicious:	
any	arbitrary

Table 3.2. Four different specifications of expected final value in the end node. In case of violations the system is considered as failed

Number of specification	Name of specification	Correct final value	Phase when final value must be correct	Final values allowed before correct value appears for the first time	Final values allowed after correct value has appeared for the first time
0	always correct	result1	phase 1		result1
1	correct after delay	result1	phase d	none	result1, none
2	detectable failures sometimes	result1	phase d	none, detectable	result1, none, detectable
3	arbitrary failures sometimes	result1	phase d	initial, result2, none, detectable, arbitrary	initial, result1, result2, none, detectable
d is a parameter of the end node.					

outgoing edges of a faulty component carry a fault value. Subsequent faultfree components are to process it such that the final value in the end component is correct. The user can specify whether it must be correct either immediately or only after some computation phases (a parameter of the end node, see Table 3.2). The final value indicates the success or unsuccess of fault tolerance.

As already mentioned there is a choice between different types of *end* nodes:

- *single input* end node (inputs of type a): The system is required to provide the correct result to the end node.
- *double input* end node (inputs of type a): At least one of the two inputs must contain the correct result (a final 1-out-of-2 decision).
- *triple input* end node (inputs of type a): The majority of the input values must be correct (a final 2-out-of-3 decision).
- *single input* end node with error indication (1 input of type a, indication of type b): Either the a-input must be correct or an error must be indicated (b-input = false). This end node can be applied to systems which can enter a safe state after fault detection.
- *double input* end node with error indication (2 inputs of type a, indication of type b): Without error indication the first input must be correct. With error indication the second one must be correct (a final switch).

Most of the end nodes correspond to well-known fault-tolerant mechanisms, which are applied as the last action of the system.

4 Genetic Algorithm for Designing Fault Tolerance

Genetic algorithms are probabilistic methods based on simulation of the survival of the fittest in the population of individuals, each of which presents a point in the solution space of an optimization problem. A genetic algorithm proceeds by generating sequences of populations, each in one evolutionary cycle. A successor population is formed by selecting the fittest individuals and applying modifying genetic operations to them: mutation of a single individual, or crossover between two individuals (parents get children). The fitness function evaluates the extend to which an individual achieves the goal of optimization. Both selection and modification support an improvement of individuals from population to population – in our case an improvement of the fault tolerance property.

4.1 Genetic Operations

In our approach each population consists of the same number p of individuals. The initial population is generated at random. Typically its fitness is low. Then, in each evolutionary cycle the following steps are applied (see Figure 4.1) according to the standard scheme of genetic algorithms:

- *selection*: First, the *fitness* of each individual in the current population is calculated. According to an elitism parameter e, the e fittest individuals are copied to the next population [3]. As is usual for genetic algorithms one has to experiment with this parameter (see experiment series in Section 6). For all remaining p – e individuals to be generated for the next generation individuals are selected in the current population. Different methods can be chosen to select fitter individuals with higher probability. If the fitness-proportional scheme is chosen, then the probability is proportional to the value of the fitness function. In case of binary tournament, however, a parent is selected by random determination of two candidates and decision for the fitter of the two. Remark: Generation of a new individual by crossover requires selection of two parents, whereas mutation requires only one (nevertheless we call it parent here).

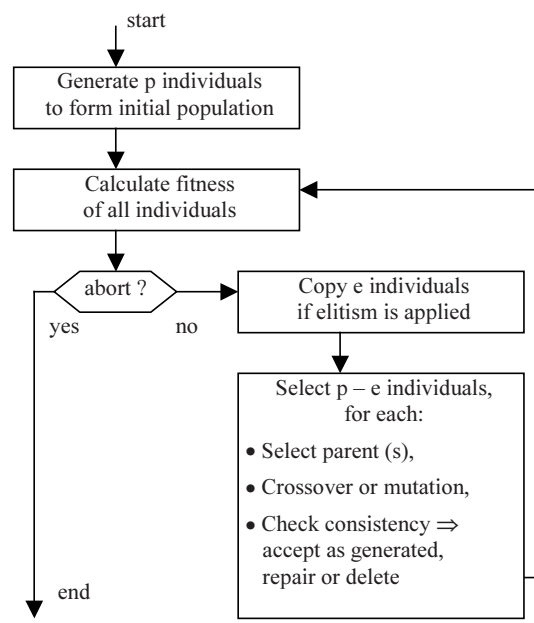


Fig. 4.1. Evolutionary cycle

- Genetic *modification* by *crossover* (combining parts of parent individuals to form child individuals) or by *mutation* (changing an element of a single individual). The probabilities of crossover and mutation can be determined by parameters q_c and q_m, respectively. Mutation plays a secondary role in genetic algorithms [8].
- *Consistency check*: Genetic modification does not guarantee to generate always a valid individual. Some constraints of the optimization problem might be violated (syntactical properties of a system description, for example). For this reason a consistency check is applied. In case of detected violation the individual is repaired, provided consistency can be reconstructed in a clear algorithmic way. Otherwise the result of the modification has to be deleted, thus requiring repeated parent selection and modification.

In this way a predecessor population is replaced by a new one. The evolution ends after a number of cycles when a predefined fitness is obtained or the specified maximum number of populations is reached.

4.2 Representation of Individuals

Each genetic algorithm works with a coding of individuals, which is an important and non-trivial issue. In Holland´s work [9] encoding is carried out using a binary string. Since then, most authors have taken this approach. For many problems in computer science binary strings cannot be recommended due to the fact that more complex structures are subject to optimization. However, for each type of encoding the genetic operations must be defined in an appropriate way. Among the many ways of potential definitions one has to find the one which promises fastest optimization. This problem has caused many authors to fall back to simpler and known encodings. In our case we have decided for a cycle-free directed graph which seems to suit best for fault tolerance techniques (the node types have been presented in Section 2), although standards for genetic operations for it are missing. Cycle-free graphs are well-suited to express the typical function of a fault tolerance mechanism: the distinction between faultfree and faulty cases. Lots of loops or deep recursions, however, are not typical for fault tolerance.

In simpler structures too much fault-tolerance-specific information would be hidden in a string, thus misleading the genetic algorithm. On the other side, general graphs would impose complications to cross-over operations. We can get rid of cycles, because we know typical fault tolerance mechanisms to have one major reason for cycles: repetition of computations which have failed before. This type of retry can be expressed by passing parts of a non-cyclic graph in subsequent phases to evaluate its fitness. However, the graph itself does not need cycles.

Our decision to use a cycle-free graph instead of a string has several implications on the genetic operations:

- Unlike binary-coded genetic algorithms, where *mutation* may be done by flipping a bit, we have to define *node mutation*. Our mutation operator selects a node at random and changes it to another randomly chosen node with the same number and data types of inputs and outputs (to preserve the validity of the fault tolerance graph).

- Our *crossover* requires the two parent graphs to be cut into two parts each. Besides the need for random selection of the cut points there is also a need for finding a possibility to connect the cut parts in a meaningful way. In other words, for each input of one part we must find an output of the same data type in the other part. This cannot be guaranteed for all cuts, even in cycle-free graphs. Special issues of this particular problem will be discussed in Section 5.
- Repair will be necessary in some cases where crossover does not lead to a consistent graph.

4.3 Fitness

4.3.1 Fitness Function

The key issue of genetic algorithms is *fitness*. Not only the performance, but the whole success of the evolution depends on its proper definition as a problem-specific optimization criterion.

The goal of our genetic algorithm is the automatic design of a fault-tolerant system with a maximum fitness value. It seems to be straightforward to take the number of tolerated faults as fitness measure. According to Section 3 we have to distinguish three fault states in up to F nodes: faultfree, benign and malicious. For comparison of systems with different numbers of nodes, we have counted the portion of tolerated faults (relative to the node number). Moreover, we count faults with a weight increasing with its approximate probability of occurrence. Since the real probability is unknown we take the fault case index (see Section 3) as a very coarse substitute. Formally, the fitness of a system is expressed by

$$\sum_{i=1}^{m} 100^{-i} \cdot \left\lfloor \frac{100 \cdot \text{number of tolerated fault cases with fault case index i}}{\text{number of fault cases with fault case index i}} \right\rfloor$$

where parameter m is the maximum fault case index taken into account. If the fault-free case is not tolerated, then fitness is set to 0 (A genetic algorithm may generate many “unreasonable” individuals which provide the wrong final result even in the absence of any fault).

4.3.2 Efficient Algorithm to Calculate Fitness

The calculation of the fitness function comes close to a reachability analysis, because the set of possible final results has to be determined for each fault case. Nevertheless it can be executed efficiently by use of the abstractions defined in the system model and the fault model. For each system generated by the genetic algorithm the calculation of its fitness proceeds as follows:

- Examine the faultless case.
- Loop for fault case index i from 1 to the maximum value m.
- For each fault case index i generate all L(i) sets of respective fault locations (nodes in the fault tolerance graph assumed to be faulty). If the cardinality L(i) of this set is too high, a random selection can be taken, leading to a smaller L(i).
- Loop for j from 1 to L(i).
- For each j examine the fault case with the respective fault locations.

The examinations of the faultless case and the fault cases determined by i and j deliver a boolean outcome each. It indicates whether the final result is correct, i.e. fault tolerance has been achieved. For each i the number of "true" outcomes is taken as enumerator, while L(i) is taken as denominator for the respective term of the sum in fitness function formula above.

Each examination of a particular fault case is done by calculating the values at each edge in the cycle-free fault tolerance graph as follows:

- The output of the start node is set to "initial".
- For each subsequent node the output is calculated by applying the component function (see Table 3.1 for the AP example). This function is deterministic. Consequently, the reachability analysis does never fork to distinguish different cases.
- After a single pass through the fault tolerance graph the end node is reached and the final result is determined according to its function (s. Section 3 and Table 3.2 for the functions of the different types of end nodes). If and only if the final result is "result1", then the boolean outcome of the examination is true.

If a writer component has written a new value to memory, then a further pass with the same steps as just described is added – unless the total number of passes exceeds d (expressing the latest point in time when the final result must be correct, see Table 3.2). Since d is typically a small value (less than 10), the number of passes is small.

The procedure for fitness calculation shows the big benefit of the fault model and its influence to the component functions. If real faults would lead to non-deterministic choices, the fault model covers all these choices by a single value. The value is "detectable" for benign faults, and "arbitrary" for malicious faults. This technique tends to be pessimistic, because the concrete wrong value in the real system may be processed more specifically (and maybe corrected immediately) whereas "detectable" and "arbitrary" are only processed in a fault-tolerant way, if the fault tolerance technique guarantees (!) to do so. Pessimistic approaches suit best when fault tolerance techniques are examined with uncertainty.

It should be noticed that our approach solves the problem of time consumption for the analysis in our fault tolerance graph. However, it does not generalize to any reachability analysis where the degree of details is different from that of our model.

5 Genetic Operations and Consistency Check

Crossover is the major modification we apply to individuals. After having cut two parent graphs into two pieces each (with randomly chosen cut positions), the "left half" of one parent has to be recombined with the "right half" of the other parent (Remark: For efficiency reasons we use the cuts twice by also recombining the remaining two halfs). The recombination must be done carefully because all the cut edges must fit to each other with respect to number and data type. Note that all nodes have exactly one output from which an arbitrary number of edges can originate. On the other side, each node has a predefined number of inputs where exactly one edge must be connected to. Hence, for each data type the following practical problems arise:

- If one parent has less cut outputs than the other (but at least one), then we insert edges as required by the cut inputs. Still there is a choice which ones we want to connect. We decided to chose the output-input pairs at random.
- Otherwise the recombination problem cannot be solved by just inserting edges. Obviously there is an alignment problem. It enforces an appropriate manipulation of the nodes.

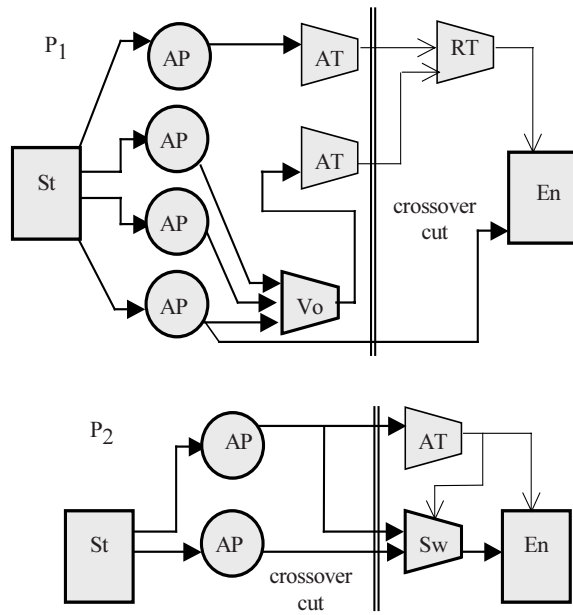


Fig. 5.1. Two parent graphs P_1 and P_2 with randomly chosen cuts

An example illustrating the alignment problem can be seen from Figure 5.1. The first parent P_1 has one cut for data type a (application data) and two cuts for b (binary test outcome), whereas the second parent P_2 has three cuts for a and none for b. If we recombine the left part P_1 and the right part of P_2, leading to child C_{12}, then the outputs of two absolute tests (AT) cannot be connected, because cut b-inputs are missing in P_2 (see Figure 5.2). Recombination of the remaining parts, leading to child C_{21}, is problematic as well. The inputs of the relative test (RT) cannot be connected, because cut b-outputs are missing in P_2 (see Figure 5.2). Moreover, even non-cut b-outputs are missing in the left part of P_2.

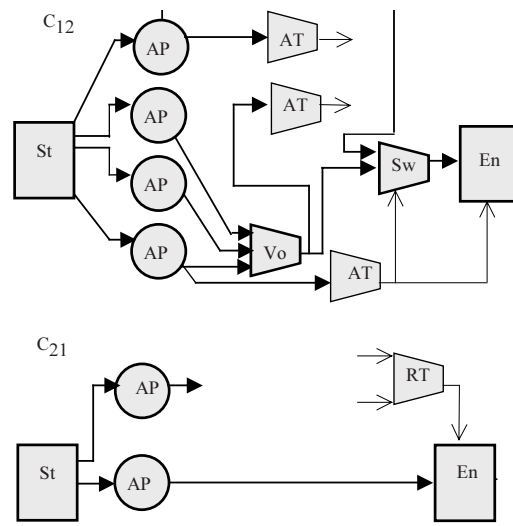


Fig. 5.2. Alignment problem caused by crossover

In case of alignment problems, for efficiency reasons, we form a child individual nevertheless. Admittedly, the child is not a well-formed graph since it owns open

edges. The invalid offspring will be discovered by the consistency check, which is applied after each crossover. According to [1] there are two main methods to handle invalid offsprings: punishing (which means deletion) and repairing. In the case of alignment problems in a fault tolerance graph the following repair actions are successful in most cases:

- Blind nodes are deleted: Nodes are considered blind, if their cut outputs deliver data of type x without having an input requiring x. In child C_{12} two blind absolute tests (AT) are deleted. The resulting repaired individual R_{12} is a valid fault tolerance graph (depicted in Figure 5.3). In general, deletion of blind components may cause further components to get blind. For this reason the repair action has to be applied recursively.
- Data converting nodes are inserted: If cut inputs require data type x, and x is not provided by any output (whether cut or non-cut) then a component is inserted which converts an existing output of data type y into data type x. In child C_{21} absolute tests (AT) are inserted to generate binary test outcomes (b) from application data (a). The resulting repaired individual R_{21} is a valid fault tolerance graph (depicted in Figure 5.3). If different data types and/or different outputs allow for the required conversion, random choices are made.
- Writer-reader pairs are inserted: A special feature of our model is the subdivision of memory access into writer and reader components (see Section 2). We can use these "half-components" for repair in the following way: If an output with the required data type exists somewhere in the model, we can connect it to a writer of this data type. Furthermore, the corresponding reader is placed just before the cut input, where the data type is needed.

In few cases repair may fail: After definition of more data types than the two we use currently, both conversion and writer-reader insertion may turn out to be impossible. Then the child is deleted.

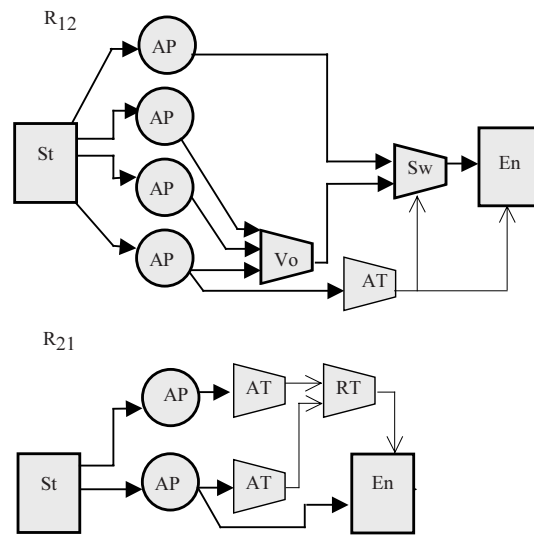


Fig. 5.3. Repaired child individuals

6 Examples of Automatically Designed Fault-Tolerant Systems

The complete genetic algorithm including the fitness function has been implemented in Java. The developed program allows for the specification of several parameters:

p number of individuals of a population,

n maximum number of nodes (= components) in a fault tolerance graph,

e number of fittest individuals transferred to the subsequent population without being modified (e stands for elitism),

s portion of fitness-proportional scheme (rest is binary tournament scheme).

q_c probability of crossover,

q_m probability of mutation,

m maximum fault case index considered by the fitness function,

c maximum number of evolutionary cycles (= number of subsequent populations).

The population size p is kept constant throughout a complete run of the genetic algorithm. The graph size is limited by n to avoid infinite growth of the system. Whenever genetic operations or repair form a graph with more than n nodes, it is made smaller by deletion of nodes, followed by repair actions to make it consistent ("deletion of blind nodes" or "insertion of converting nodes" or "writer-reader pair insertion", see Section 5).

We have designed our algorithms to be as flexible as possible and to allow for any combination of strategies (fitness-proportional scheme and binary tournament in the same evolutionary process, for example). After having gained more experience we will be able to come up with recommendations on parameter values.

Table 6.1. Parameters of experiments

Parameter		Exp. Series 1	Experiment Series 2
population size	p	50	50
max. number of nodes	n	14	20
elitism	e	0	12
specification of end node (see Table 3–2)		always correct	detectable failures sometimes
end node parameter	d	don't care	2
portion of fitness-prop.	s	1	0.3
probability of crossover	q_c	1	0.85
probability of mutation	q_m	0	0.05
max. fault case index	m	3	3
max. n. of evolut. cycles	c	100	40

The problem of designing fault-tolerant systems is highly constrained. In constrained search spaces some effort must be spent in analysing the obtained results. In order to validate our approach several series of experiments have been conducted. The systems they have produced demonstrate that our genetic algorithm can design fault-tolerant systems automatically.

The parameters of two series can be seen from Table 6.1. In series 1 we have chosen the strongest specification in the end node: "always correct" (specification 0 in Table 3.2). In series 2, however, the system is allowed to take some time (precisely speaking d = 2 phases) to deliver the correct final result.

As was expected, known solutions to fault tolerance (like TMR, see Figure 6.1, or use of spare components, see Figure 6.2) have been

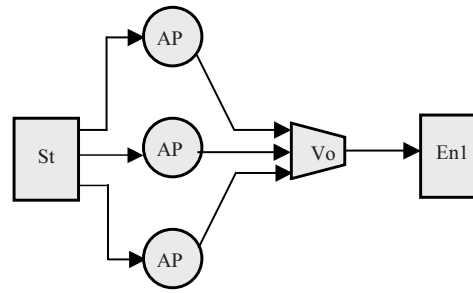


Fig. 6.1. Voting generated by genetic algorithm in series 1 (end node is of type "single input")

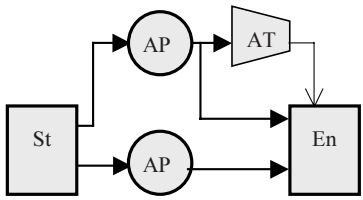


Fig. 6.2. Simple fault-tolerant system generated by genetic algorithm in series 1 (type of end node is "double input with error indication")

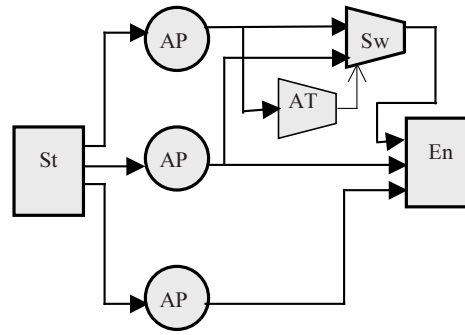


Fig. 6.3. Fault-tolerant system generated by genetic algorithm in series 2 (type of end node is "triple input")

"reinvented" by the genetic algorithm. More complex structures, able to tolerate multiple faults have been generated, too. The system from Figure 6.3 tolerates many double faults, with few exceptions: If both the process (AP) shown on top of Figure 6.3 and the subsequent absolute test (AT) (or the switch (Sw)) fail simultaneously, then fault tolerance may fail. The systems depicted in Figures 6.3 and 6.4 are not standard solutions to fault tolerance. They can be considered as interesting designs the genetic algorithm has provided. The system from Figure 6.4 is able to tolerate twelve double-faults.

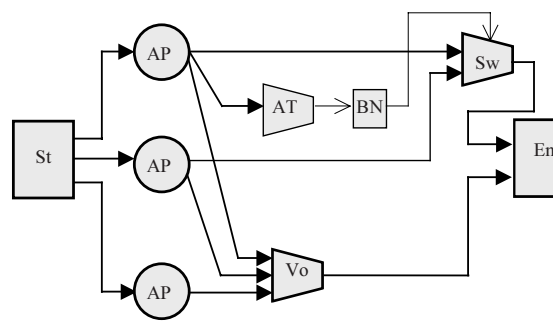


Fig. 6.4. Fault tolerant system produced by genetic algorithm in series 2 (end node is of type "double input")

By our approach a framework has been built consisting of all the necessary definitions and algorithms: system model, fault model, genetic operations and fitness function. In all these four areas we have developed special methods which focus on the design of fault-tolerant systems. The experiment series have demonstrated that the generated systems exhibit good fault tolerance properties compared to known systems. Table 6.2 lists more example systems and compares them to known solution. The achieved fault tolerance capability support the statement that the genetic algorithm can be considered successful. All systems have been modelled at the same level of abstraction that is commonly used for the description of the functionality of known fault tolerance techniques. Higher degrees of details and usage of layered structures will be subject to future work. We need to define further component types and data types for this purpose, but from our current point of view it is expected that the fault model, the genetic operations and the fitness function are also suitable for more detailed system models.

Table 6.2. Comparison of example systems generated by the genetic algorithm and known solutions to fault tolerance

name of system	number of components	number of processes	type of end node	portion of tolerated single faults	portion of tolerated double faults
Systems generated by the genetic algorithm:					
System 1	10	5	double input	100 %	98 %
System 2	10	4	triple input	100 %	89 %
System 3	9	5	triple input	100 %	95 %
System 4	10	4	triple input	100 %	80 %
System 5	7	3	double input	100 %	90 %
Systems with known standard fault tolerance techniques:					
Voter	5	3	double input	100 %	60 %
Switch to spare	5	2	double input	100 %	60 %

7 Summary and Perspective

In this paper we have presented a genetic algorithm and a special fitness function which is appropriate for the automatic design of any fault-tolerant structure. Unlike other approaches there is no given scheme for redundancy. Instead, our algorithm “invents” redundant arrangements of components by itself in a very flexible way.

One of the key points behind our solution is the definition of a suitable fault model which allows for a fitness function which is a good compromise between accurate assessment and efficient execution.

The fault-tolerant systems generated in the experimental series clearly deviate from the known solutions to fault tolerance with respect to their structure. Yet they achieve comparable, and sometimes even better fault tolerance properties. In this sense our results are promising.

Currently we refine our approach and conduct more experiments with a variety of parameters and end node specifications. In future work we plan to extend our work into four directions:

- The crossover strategy will be modified such that the necessity of repair is minimized.
- A cost criterion will be added to the fitness function to allow for two-dimensional optimization in the fault tolerance and cost space.
- The system structure will be more detailed by definition of additional component types. Moreover, a predefined fixed part can be added to the system model to express a given environment.
- The most interesting extension will penetrate the field of fault-tolerant peripheral devices. Here less standard techniques are known because efficient solutions to fault tolerance highly depend on the types of peripheral devices and their properties. Further components (node types) will be defined which are "between" computing systems and the mechatronic outside world (like passivation of a digital-analogue converter, for example). In addition, fault tolerance by pure mechanical devices might also be included in the process of automatically designing fault-tolerant systems by genetic algorithms.

References

[1] Blickle, T.; Thiele, L.; Teich, J.: System-Level Synthesis Using Evolutionary Algorithms. Design Automation for Embedded Systems, Kluwer Academic Publishers (1998)
[2] Deeter, D. L.; Smith, A. E.: Heuristic Optimisation of Network Design Considering All-terminal Reliability. Proceedings of the Annual Reliability and Maintainability Symposium (1997) 194–199
[3] De Jong, K. A.: An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Ph.D. Thesis, University of Michigan (1975)
[4] Dengiz, B.; Altiparmak, F.; Smith, A. E.: Efficient Optimisation of All-terminal Reliable Network Using Evolutionary Approach. IEEE Transaction on Reliability, vol. 46, no.1. (1997) 18–26
[5] Echtle, K.: Fehlertoleranzverfahren. Springer, Heidelberg (1990)
[6] Echtle, K.; Niedermaier, A.: Graph-Oriented Assessment of Fault-Tolerant Behaviour. Second European Workshop on Dependable Computing EWDC–2, Firenze (1990)
[7] Gen, M.; Cheng, R.: Genetic Algorithms and Engineering Optimisation. John Wiley & Sons, Inc., NY (2000)
[8] Goldberg, D.E.: Genetic Algorithms in Search, Optimisation, and Machine Learning. Addison-Wesley Publishing Company, Incorporated, Reading, Massachusetts (1989)
[9] Holland, J.: Adaptation in Natural and Artifical Systems. MIT Press, Cambridge, MA (1975)
[10] Mazumder, P.; Rudnick, E. M.: Genetic Algorithms for VLSI Design, Layout & Test Automation. Prentice Hall PTR, NJ (1999)
[11] Schiffmann, W.: Encoding Feedforward Networks for Topology Optimization by Simulated Evolution. Proceedings of the Fourth International Conference on Knowledge-Based Intelligent Engineering Systems & Allied Technologies, KES '2000, Vol. 1. (2000) 361–364
[12] Thompson, A.: Evolutionary Techniques for Fault Tolerance. Proceedings of UKACC International Conference on Control. (1996) 693-698

Cyclic Strategies for Balanced and Fault-Tolerant Distributed Storage

Ricardo Marcelín-Jiménez[1] and Sergio Rajsbaum[2]

[1] IIMAS, UNAM and EE Dept., UAM-Iztapalapa; Atlixco 186; 09340 México D.F.
calu@xanum.uam.mx
[2] Instituto de Matemáticas, UNAM; Ciudad Universitaria; 04510 México D.F.
rajsbaum@math.unam.mx

Abstract. Given a set V of active components in charge of a distributed execution, a *storage scheme* is a sequence of subsets, $B_1, B_2, \ldots, B_b$, of V where, succesive global states are recorded. The subsets, called *blocks*, have the same size and are scheduled according to some fixed and cyclic calendar of b steps. During i-th step, block B_i is selected. Next, a global snapshot is taken and each component sends its corresponding local state to one of the appointed places in B_i, in a way that each component stores (approx.) the same number of local states. Afterwards, if a component of B_i crashes, all of the data stored in the block is useless, because the global state can not be reconstructed. In this case, the information recorded in an earlier block can be used to recover a global state, provided there is at least one such block where no component has crashed. The goal is to design storage schema that tolerate as many crashes as possible, while trying to have each component participating in as few blocks as possible and, at the same time, working with large blocks (so that a component in a block stores a small number of local states). In this paper several such schema are described and compared in terms of these measures.

1 Introduction

In a distributed execution where active components have a non-zero probability of crash failures, the regular recording of the global state allows the possibility of restarting the system from the most recently saved, reconstructible global state in the event of a crash [1].

This work is about using distributed storage to save global snapshots of a distributed execution. Its main contribution is the establishment of the critical measures and related concepts about balanced distribution of local snapshots, as well as presenting and analyzing a variety of efficient distribution schema.

1.1 Contributions

In its simplest form, distributed storage consists of spreading a collection of files across the components of a network. Of course, each application has special requirements.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 214–233, 2003.

The regular recording of global states [2,3] in a distributed execution brings about a storage management problem in the components selected for this purpose. This document addresses the performance issues of storage policies that could be regarded as balanced strategies, that is where responsibilities are fairly distributed among all available components. There is a trade-off between the number of steps a component works as a storehouse, and the amount of data stored in each step by the component; each of the solutions here considered must have the following properties:

- In every step, a global state consisting of local states of every network component must be stored in a subset of the components, called a *block*.
- The set of blocks to be used is statically defined before the computation starts. These blocks are used cyclically (one per step) in a predefined order.
- Each component must participate in as few blocks as possible, and every component must participate in as many blocks as any other component.
- Blocks must be as large as possible, to reduce the amount of storage required of each component participating in the block.

A *storage scheme* consists of the ordered set of blocks mentioned above. It is a distributed solution, in the sense that no special component is necessary to coordinate the storing procedures. This is because the list of blocks is defined a priori, and hence every component can compute locally the block to be used in a step, using a table describing the list of blocks. We describe various storage schema, show more efficient ways of computing them than with an explicit table, and analyze performance measures for each one.

In section 2 the underlying model is reviewed. All storage schema to be developed belong to this formal description. The parameters used to evaluate and compare the performance of each solution are introduced: the storage period measures the total amount of memory claimed by a given scheme, it also indicates the recycling term of any buffer; the number of contents indicates the number of buffers that each participant must contribute with, in order to support the storage solution; finally, the critical number of failures measures the strength of the policy to resist crash failures. The concept of balanced and incomplete strategy is defined.

In section 3 the first scheme named "All k out of v" is developed. This could be considered as an elementary solution that consists of building all the possible subsets of size k from a set of v components, each subset corresponds to a storage block. In this section and the next two, results focus on the calculation of the parameters for each scheme introduced. In all cases, difficulty generally comes from determining the critical number of failures. In section 3, particularly, results are drawn from simple counting arguments: given a set of v elements, how many of them should be eliminated, in order to make impossible the construction of at least one subset of k elements?

In section 4 the second scheme named "sliding slot" is developed. Here, places are imagined as situated around a circular table, an arc of circumference that spans k spaces is rotated and a component, or place, is only selected to be in a

storage block if it is spanned by the arc on its current position. Results here are based on well known concepts from subgroups of finite cyclic groups. The main challenge again, is to determine the critical number of failures (see [4] pp. 62).

In section 5 the third and final scheme named "finite geometries" is developed. Here each block is a set of k points defining a line in a finite geometry, either the projective plane or the affine plane. We use results known about *blocking sets* in block designs to determine the supportable number of failures.

Section 6 includes the comparative performance of each scheme and directions for future work. Time is a dimension that is also considered, it is clear that blocks are rescheduled at the end of the cycle term, but this condition does not prevent from interleaving two or more strategies. We also explore different approaches to improve fault-tolerance. Finally, section 7 contains the conclusions.

1.2 Related Work

Distributed storage and retrieval will become key procedures as long as telecommunications and computer systems depend on them not only to support the services provided to the increasing number of users, but also to keep up with the inner operations of these very systems. The OceanStore project [5] is only a selected example of the forthcoming applications of this emerging technology.

The strong link between algebra and combinatorics [6,7,8], is the main theoretical base of this work. [9] offers a wide and updated set of samples where *codes* and *designs* are applied to IT problems. One of the most relevant subjects there considered, as far as this work is concerned, are erasure codes used to build up Redundant Arrays of Independent Disks (RAID) [10]. Nevertheless, RAID is a centralized solution while here a distributed approach is developed. [11] and [12] worked with the so-called Redundant Array of Distributed Disks (RADD) which is the type of solution that will be addressed, using the new idea of coordination schema. Several works use designs to study cooperation with limited communication. See [13] for a survey and more references on the use of designs, and in general to some more of the many papers on load balancing techniques.

Up to now data integrity has been achieved using erasure codes [14]. Somewhat orthogonal to the main stream of this work is [15], that provides a way to tolerate failures by dispersing each data item into several components. This method is more demanding on communication, because reconstructing information means multiple remote accesses. On the other hand, it has advantages in supporting backup copy storage. Indeed, this view point is explored in section 6.2.

There is also a close relationship between quorum systems and storage schema, both are collections of sets of components where information is stored [16]. Load balancing is a very important issue to be considered in both cases. The differences arise from the fact that here, in storage schema, consistency is not an important aspect, while capacity and fault-recovery are the main problems. This means that for storage schema it is not necessary to establish a chain of causality between any two storage actions and that the intersection property between blocks can be relaxed. When storing a massive amount of information, new performance figures are to be measured.

Additionally, it is worth mention the work of [17] on revolving hierarchies to accomplish a fair distribution of work in a distributed environment. Like the second scheme here presented, called "sliding slot", it can be understood as permutations on the roles each component plays during the computation and storage schedule.

2 Selection Strategies and Storage Schema

Let V be a set of v elements, or places. A *selection strategy* (strategy for short) $\mathcal{B}$ on V is a collection of b subsets of V called *blocks*, where

- $k_j = |B_j|$, $j = 1, \ldots, b$, is the cardinality of the j-th block, and
- $r_i = \sum_{j=1}^{b} |\{v_i\} \cap B_j|$, $i = 1, \ldots, v$, is the number of blocks the element v_i is in.

Having a strategy $\mathcal{B}$, its incidence matrix $\mathbf{A} = (a_{ji})$ of size $b \times v$ is defined as

$$a_{ji} = \begin{cases} 1 \text{ if place } i \text{ is in block } j, \\ 0 \text{ otherwise,} \end{cases}$$

Clearly, row j corresponds to block B_j. Alternatively, column i shows the blocks which element v_i is located.

A strategy is called *balanced* if

$$k_j = k,\ j = 1, \ldots, b \text{ and}$$
$$r_i = r,\ i = 1, \ldots, v.$$

Counting the total number of 1's in the incidence matrix of a balanced strategy in two different ways: first by rows and then by columns, it is straightforward that for a strategy of this type

$$bk = vr. \tag{1}$$

A strategy is called *incomplete* if

$$r_i < b,\ i = 1, \ldots, v,$$

For a given strategy $\mathcal{B}$ on V, a *spanning subset* is a subset V' of V such that for every block $B_j \in \mathcal{B}$, $V' \cap B_j \neq \emptyset$. A *minimum spanning subset* V'_m, of cardinality $f = |V'_m|$, is a spanning subset such that for any spanning subset V', $f \leq |V'|$.

A *storage scheme* is a distributed coordination procedure that runs on a set of network components and is executed through *steps*. Every component is in charge of one disk of its own. At each step, a subset of the components is selected to participate as storehouses, and every component knows which others are working in this role. Subsets are eventually rescheduled.

A step begins when each component takes a snapshot of its local state, called a *fragment*. Next, it determines, according to the storage scheme that is being used, the storage component to send its fragment for storage. All the fragments stored in the same component, during one step, define a *content*. For each step a component participating as a storehouse must have a different buffer space and some mechanism to recognize those components whose states it stores.

Components, are assumed to have a unique number $i \in (0, v-1)$. During the execution, each component risks a crash failure. We assume that once it stops working, it is never restarted and it is considered permanently out of service [18, 19].

For each scheme under investigation, three parameters are evaluated: the *storage period*, the *number of contents*, and the *critical number of failures*. The storage period, or period for short, is defined as the maximal number of steps that must elapse, before a particular content is renewed. The number of contents counts the different steps where a component plays a storage role, therefore it also measures the different contents stored in this place. Finally, the critical number of failures is the minimal number of places whose failure would make impossible the recovery of even one stored global state, thus forcing the entire computation to be restarted.

Once it starts working as a storehouse, a participant must clear and re-utilize the same buffer it filled in one period ago. It may be possible for a storehouse to clear one of its buffers without refilling it, for example when it misses the record of a faulty processor. Unless a verification procedure is triggered regularly, all buffers might become cleared, completely disabling the recovery procedure. It seems that having the verification procedure at the end of a cycle is a good compromise between the time spent by control procedures and the part of the execution that could be lost in case of failure.

Storage schema can be constructed from selection strategies. Let V be the set of available storehouses. For any step, the subset of appointed places can be drawn from a given strategy $\mathcal{B}$ on V. A storage scheme that achieves a trade-off between the resources a processor spends as a storehouse, and the resources spent on its main task, must be an incomplete strategy. Incompleteness implies that every component spends at least one step not as storage, therefore contributes to the computation. A storage scheme that evenly distributes this same duty among all the available places, must be a balanced strategy. Balance guarantees that any place acts as storage as often as any other.

Suppose now that a balanced and incomplete strategy (good strategy, from now on) is under the control of a cyclic calendar that appoints each block in a fixed order before blocks are rescheduled. This means that the storage period will be equal to the maximal number of blocks b that can be used before one of them is selected again. The number of contents will be r, i.e. the total of blocks where a participant is appointed. Last, it is sufficient to eliminate all of the f elements from a minimal spanning subset in order to destroy every block in $\mathcal{B}$.

Additionally, some other measures drawn from the main parameters can provide helpful insight on the performance that a scheme bears: the *bottleneck* is the

amount of fragments stored by a single participant during a step, which is also the content size. Now, since there are v different fragments for a storage step, and these are supposed to be evenly distributed among the k available places, the bottleneck η will be $\frac{v}{k}$, which is the total service requests any storehouse receives at the same time. The *occupation rate*, denoted by ρ, is the fraction of the period a component participates as a storehouse. Clearly $\rho = \frac{r}{b}$.

In the following sections it will be shown that, for a given good strategy, most of its parameters depend on the size k of each block. A small k means that every processor keeps a small number r of contents, but for the few steps when it works, it risks to be overloaded by the total number of fragments η that are sent to it. On the other hand, a big k means that every processor has to put off its main task in order to work as storehouse for most of the steps during a period. Of course, each time it only handles a reduced number of requests. So, it seems that there are two conflicting measures contributing to the *total price* of the scheme:

$$P = r + \eta = \frac{b}{v}k + \frac{v}{k}, \tag{2}$$

A good trade-off could be realized when P were minimized. i.e. if there exists a value $k \in [1, v)$ that produces the least total price in this range. The best schema will be based on good strategies with the smallest number of blocks, bearing the minimal total price, but not sacrificing robustness.

For any storage scheme it is very important to develop and install a procedure to generate, on the fly, the next scheduled subset in an efficient and distributed way. Otherwise, the entire collection of blocks should be recorded on each place before the scheme starts. All of the solutions to be developed bear this important property that enables a very simple procedure, sometimes called *the revolving ordering algorithm*, to determine the next block to be appointed.

It must be noticed that in order to implement a whole recovery program there are some other modules that must be developed together with the storage scheme. For instancea fault detection module, or a load redistribution and balance module. Also, before the scheme is started an initialization module must be called in order to guarantee that for every block, each of the appointed components stores the same number of fragments and, simultaneously, achieving some optimality measure. For instance, minimizing the distance between the source and the place where a fragment is stored. It is in this sense that word "best" is used for the following sections. This work does not address any aspect of these subsystems.

3 Selecting All k Out of v

The first storage scheme of this work is based on a very simple strategy: include as blocks all the subsets of size k from a set V of v elements.

Lemma 1. *Given a fixed v, for an "all k out of v" scheme, their parameters are the following:*

```
< 1> Initial conditions
< 2>    j = 0;
< 3>    M = v!/(k!(v−k)!);
< 4>    let c = (c_1, c_2, ..., c_{k+1}) be a vector,
        such that c_i ∈ [1...v], 1 ≤ i ≤ k;
< 5>    let c_{k+1} = v + 1, be a constant coordinate of c
< 6> In the j-th storage step
< 7>    l = 1;
< 8>    while (l ≤ k) and (c_l == l)
< 9>       l = l + 1;
<10>    if (k ≠ l mod 2)
<11>       if (j == 1);
<12>          c_1 = c_1 − 1;
<13>       else
<14>          c_{l−1} = l;
<15>          c_{l−2} = l − 1;
<16>    else
<17>       if (c_{l+1} ≠ c_l + 1);
<18>          c_{l−1} = c_l;
<19>          c_l = c_l + 1;
<20>       else
<21>          c_{l+1} = c_l;
<22>          c_l = l;
<23>    let B = {c_i ∈ c};
<24>    let i − 1 be the best place such i ∈ B;
<25>    send local snapshot[j] to i − 1;
<26>    j = (j + 1) mod M;
```

Fig. 1. Which are the storehouses for the next step, in all k out of v?

$$b = \binom{v}{k},$$
$$r = \binom{v-1}{k-1},$$
$$f = v - k + 1.$$

Proof. Clearly, the total number of blocks b corresponds to the binomial coefficient $\frac{v!}{k!(v-k)!}$. The second parameter follows from eq.(1). Last, suppose that $v - k$ elements are dismissed from V, the remaining k elements still make up a block in $\mathcal{B}$. But, if $v - k + 1$ elements fail, it will not be possible to build any complete block, with intact elements. □

For this particular strategy, the number of blocks depends on k. From the behavior of eq.(2) it is realized that the associated scheme meets its optimal

```
< 1> Initial conditions
< 2>       j = 0;
< 3>       d = gcd(v, γ);
< 4>       b = v/d;
< 5>       let c_0 = (c_0, c_1, ..., c_{v-1}) be a binary vector,
           such that c_0 = ... = c_{k-1} = 1, and c_k = ... = c_{v-1} = 0;
< 6> In the j-th storage step
< 7>       let c_j = [c_0, c_1, ., c_{v-1}] be the j-th slot, from rule s.s.;
< 8>       let B = {i | c_i ∈ c_j & c_i == 1};
< 9>       let i be the best place such i ∈ B;
<10>       send local snapshot[j] to i;
<11>       j = (j + 1) mod b;
```

Fig. 2. Which are the storehouses for the next step, in sliding slot?

total price when $k = v$ and also for a non-integer value such that $1 < k < 2$. In the first case, the underlying strategy is not incomplete, which means it might be a very expensive solution. In the second case, bottleneck is the main problem.

The collection of blocks of this scheme is generated by following the procedure shown in fig. 1. It is based on the minimal change ordering algorithm also called revolving door ordering algorithm, by Kreher and Stinson [20]. The main property of this construction is that any two consecutive blocks differ from each other by two elements exactly. It is a very efficient procedure that produces all of the possible blocks of the scheme with the minimal amount of resources (i.e. space and time), for it only requires the preceding block and manipulates at most two coordinates of a given vector. Once the next k-subset is determined, every participant can pick from this set, the best place to send their data fragment.

4 Sliding Slot

The second storage scheme of this work, named "sliding slot" is presented here. It can be explained by using an analogy: let v persons be seated before a circular table; k consecutive are selected from them as a block. This selection defines a *slot* that can also be understood as an arc of the circumference including k persons. Next, this arc rotates γ *positions*, clockwise for instance, and the k persons spanned by the arc in this new place are the following selection. Successive selections or blocks are established this way.

A participant that was selected up to the previous step, but it is no longer inside the slot, is said to *leave* it. In contrast, a participant that joins a slot this step is said to *get in* the slot.

Each block generated with this scheme can be understood as a v-tuple of the form $(c_0, c_1, \ldots, c_{v-1})$, where the i-th component is 1 or 0, whether place i

works or not as a store in this block. Blocks can be determined according to the following:

rule s.s. If $\mathbf{c} = (c_0, c_1, \ldots, c_{v-1})$, where $c_i \in F_2$ for $i = 0, 1, \ldots, v-1$, represents a slot, then each one of them has a corresponding polynomial of the form $c_0 + c_1 x + \ldots + c_{v-1} x^{v-1}$ in the ring $F_2[x]/ < x^v + 1 >$ of the polynomials on F_2 modulo $x^n + 1$ such that, if $\mathbf{c}_j$, $\mathbf{c}_{j+1}$ are two consecutive slots and $p_j(x)$, $p_{j+1}(x)$ their respective polynomials in $F_2[x]/ < x^v + 1 >$, then $p_2(x) = x^\gamma p_1(x)$. For this scheme there exists a slot $\mathbf{c}_0 = (c_0, c_1, \ldots, c_{v-1})$, such that $c_0 = c_1 = \ldots = c_{k-1} = 1$, $c_k = c_{k+1} \ldots = c_{v-1} = 0$ and all the successive slots are derived from this one, according to the procedure just explained.

Clearly, for an initial slot called $\mathbf{c}_0$, the set of rotations of $\mathbf{c}_0$ forms a cyclic group G of order v such that $G =< 1 >$. This means that the one-position rotation generates the entire group. Also, G is isomorphic to Z_v. Any other element in this group, say γ, will also generate a cyclic subgroup H such that $|H| = T = v/d$, where $d = \gcd(v, \gamma)$. From the Euclid's algorithm, it means that the set of slots generated applying γ positions rotations to $\mathbf{c}_0$ would be the same that the set of slots generated applying d positions rotations to $\mathbf{c}_0$.

Suppose, for instance, a fixed v and a fixed γ, such that $\gcd(v, \gamma) = 1$, the parameters for this scheme are: $b = v$, $r = k$, $f = \lceil \frac{v}{k} \rceil$. The first two are easily determined. Next, given an initial slot $\mathbf{c}_0$ that is rotated one position per step, it will take v steps to encompass the original places again. Also from the fact that the slot rotates only one position, and the slot size is k, any place is selected to participate in k different blocks. The bottleneck is immediate. Now, cover the circumference with the fewest number of slots. This number is $\lceil \frac{v}{k} \rceil$. Drop out the first place on each of the slots. There are not k consecutive remaining places.

A very useful feature, for this particular case, comes from the fact that the total number of blocks $b = v$ is a small number and additionally does not depend on the block size k. According to eq.(2), this scheme meets its optimal total price $P = 2\sqrt{v}$, when $k = \sqrt{v}$. Indeed, for some values of v and $\gamma > 1$ it is possible to achieve a lower price, but these solutions will be weaker than the case where $\gamma = 1$.

Lemma 2. *Given fixed values for v, γ and k, with $d = \gcd(v, \gamma)$ then the parameters of a "sliding slot" scheme are the following:*

$$
\begin{aligned}
b &= \frac{v}{d}, \\
r &= \frac{k}{d}, \\
f &= \begin{cases} \lceil \frac{v}{d\lceil r \rceil} \rceil & \text{if } d \leq k, \\ \frac{v}{d} & \text{otherwise.} \end{cases}
\end{aligned}
$$

Proof. If $d \leq k$, a segment of length d fits $\lceil \frac{k}{d} \rceil$ times into a segment of length k. This means that the distance between the point c_0 at position 1 of a fixed

slot $\mathbf{c}$ and the point $c_{k'}$ at position 1 of the closest slot disjoint from $\mathbf{c}$ is $s = d\lceil \frac{k}{d} \rceil$. It follows that the maximum number of disjoint slots required to span a circumference of length v is $f = \lceil \frac{v}{s} \rceil$. Dismissing the first position at each of these spanning slots will be enough to destroy all of the blocks in the underlying strategy.

Now, suppose that it is possible to eliminate $f' < f$ points to affect the entire strategy. For instance, dismissing every $s + 1$ points from c_0 on. Equivalently, it means that circumference is divided into $\lceil \frac{v}{s+1} \rceil$ disjoint segments, each one of length $s + 1$. But, if there are at least d consecutive segments of this length, there will be enough room to accomodate a complete block that will remain unaffected. On the other hand, in order to have $f' < f$ there must be at least s segments of length $s + 1$. Since $s > d$, there will be at least one undestroyed block(!).

If $d > k$, then all the slots are disjoint so it will be enough to pick one point from each to be in a spanning subset: $\frac{v}{d}$ points. □

Following figure 2 it can be seen that the distributed coordination procedure works this way: starting from $\mathbf{c}_0$, every participant generates by itself the next slot, when a new storage step is called. Then each one recognizes, from this new tuple, the best active storehouse and sends its local snapshot to it.

5 Finite Geometries

The third storage scheme, named "finite geometries" is presented in this section. Two related strategies called *finite projective plane* and *finite affine plane* will be developed. This last family of solutions has a geometric motivation: places are regarded as points. Subsets of points (i.e. blocks) are defined by lines which inherit particular properties from the geometry. There exists a very rich theory about these mathematical objects [6,7,8].

Let v, k, λ be positive integers such that $v > k > 2$. A (v, k, λ)-*balanced incomplete block design* or (v, k, λ)-BIBD is a pair $(V, \mathcal{B})$ such that the following properties are satisfied:

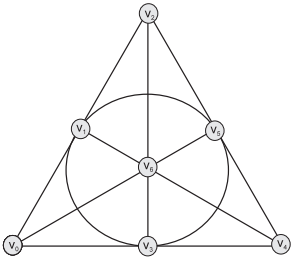


Fig. 3. Projective Plane of Order 2

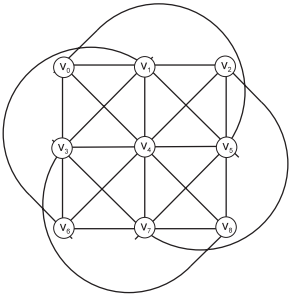


Fig. 4. Affine Plane of Order 3

1. V is a set of v elements called *points*.
2. $\mathcal{B}$ is a collection of b subsets of V called *lines* or *blocks*.
3. each block has exactly k points.
4. every pair of distinct points is contained exactly in λ blocks.

An $(q^2+q+1, q+1, 1)$-BIBD is called a *projective plane of order* q. Note that the projective planes are symmetric BIBDs, which means that $b = v$ and also that every pair of blocks intersect in a unique point. It is well known that projective plane of order q exists whenever q is a prime power. Fig. 3 depicts a projective plane of order 2.

An equivalent, and also well known, definition for the finite geometry here considered is the following [21,22,23]: A projective plane is a collection of points and lines satisfying: (i) there is a unique line containing any two points, (ii) any two lines meet at a unique point, and (iii) there exist four points no three of which lie on a line. From the above definitions the next results are drawn:

Lemma 3. *Given a fixed v such that $v = q^2+q+1$, where q is a prime power, the parameters of a storage scheme based on a projective plane of order q are the following:*

$$b = q^2+q+1,$$
$$r = q+1,$$
$$f = q+1.$$

Proof. The first two parameters are properties of the projective plane. Now, for the third parameter, in a projective plane of order q, a spanning subset will either contain a line or not. If it does contain a line, then since all lines of a projective plane intersect, the points of a line B_1 produce a spanning subset with $q+1$ points. It is known that a set of points in a projective plane of order q which does not fully contain any line and which has non-empty intersection with all lines must have at least $q+\sqrt{q}+1$ points which is always larger [21, 24]. Therefore, the minimal spanning set is $q+1$ the same size as a block, k. □

Projective planes, like sliding slot, offer a very good number of blocks, $b = v$, for a balanced incomplete selection strategy. On the other hand,the block size of a projective plane is fixed with respect to the number of components and cannot be improved. The good news is that, $k = O(\sqrt{v})$ anyway, which means that eq.(2) is nearly optimal.

Let $(G, +)$ be a finite (additive) Abelian group of order v in which the identity element is denoted 0. (In many applications $G = \mathcal{Z}_v$, the integers modulo v). A (v, k, λ)- *difference set* in G is a subset $D \subseteq G$, that satisfies the following properties:

1. $|D| = k$,
2. The collection of values $\{x - y : x, y \in D, x \neq y\}$ contains every element in $G \backslash \{0\}$ exactly λ times.

```
< 1> Initial conditions
< 2>      j = 0;
< 3>      let D be the difference set for the p.p.of order q;
< 4> In the j-th storage step
< 5>      let B = {x + j mod v|x ∈ D};
< 6>      let i be the best place such i ∈ B;
< 7>      send local snapshot[j] to i;
< 8>      j = (j + 1) mod v;
```

Fig. 5. Which are the storehouses for the next step, in projective plane?

Difference sets can be used to construct symmetric BIBDs. Let D be a (v, k, λ)-difference set in an Abelian group $(G, +)$. For any $g \in G$, define

$$D + g = \{x + g : x \in D\},$$

any set $D + g$ is called the *translation* of D. Then define

$$\mathrm{Dev}(D) = \{D + g : g \in G\},$$

which is the set of all translates of D and is called the *development* of D. It is known [8] that $(G, \mathrm{Dev}(D))$ is a symmetric (v, k, λ)-BIBD, and that there exist a $(q^2 + q + 1, q + 1, 1)$-difference set in $\mathcal{Z}_{q^2+q+1}$ for every prime power q. These results will be fundamental for the construction of the distributed coordination procedure associated to storage schema based on projective planes:

Following figure 5 it can be seen that the distributed coordination procedure works this way: starting from the corresponding difference set, every participant generates by itself the next line of its underlying geometry, when a new storage step is called. Then each one recognizes, from this new block, the best active storehouse and sends its local snapshot to it.

An *affine plane of order* q is an $(q^2, q, 1)$-BIBD. It has $b = q^2 + q$ blocks. For any prime power q there exists an affine plane of order q.

Suppose $(V, \mathcal{B})$ is a (v, k, λ)-BIBD. A *parallel class* in $(V, \mathcal{B})$ is a subset of disjoint blocks from $\mathcal{B}$ whose union is V. Observe that a parallel class contains $\frac{v}{k} = \frac{b}{r}$ blocks, and a BIBD can have a parallel class only if $v \equiv 0 \bmod k$. A partition of B into parallel classes is called a *resolution*, and $(V, \mathcal{B})$ is said to be *resolvable* if $\mathcal{B}$ has at least one resolution. It is well known that an affine plane is resolvable.

An alternate, but equivalent, definition of an affine plane is the following [21, 22,23]: An affine or Euclidean plane is a collection of points and lines satisfying (i) there is a unique line containing any two points, (ii) given any line L and any point $p \notin L$ there is a unique line through p which does not meet L, and (iii) there exist three points not on a line. It is also important to notice that an

affine plane is obtained by deleting the points of a fixed line from a projective plane. Fig. 4 depicts an affine plane of order 3.

For this second kind of finite geometry it can be concluded that

Lemma 4. *Given a fixed v such that $v = q^2$, where q is a prime power, the parameters of a storage scheme based on an affine plane of order q are the following:*

$$b = q^2 + q,$$
$$r = q + 1,$$
$$f = 2q - 1.$$

Proof. Recall that, for the affine plane of order q, $v = q^2$, $b = q^2 + q$, $k = q$ and $r = q + 1$. The first two results follow immediately from these parameters. For the last parameter it is known that in an affine plane of order q, the smallest set of points that has non-empty intersection with every line must have size at least $2q - 1$ [21,25]. This set can either contain or not contain a line and still achieve this optimal number of points. An example of a spanning set that does contain a line is the set of points of one line and exactly one (arbitrary) point from each other line in the same parallel class from the affine plane. An example of such a set of points that does not contain a line is the following [21], consider points v_0, v_1, v_2, v_3 and lines:

- B_1 through points v_0 and v_1,
- B_2 through points v_0 and v_2,
- B_3 through points v_0 and v_3,
- B_4 through points v_2 and v_3,
- B_5 through points v_1 and v_3,
- B_6 through points v_1 and v_2,

such that B_1 is parallel to B_4 and B_2 is parallel to B_5. Choose a point $v \in B_6 \setminus \{B_3 \cap B_6, v_1, v_2\}$. Then $S = (B_1 \cup B_2 \cup \{v, v_3\}) \setminus \{v_1, v_2\}$ is a spanning set that does not fully contain any line. □

Affine planes offer a number of blocks almost as small as projective plane or sliding slot do, this is $b = O(v)$. On the other hand, $k = \sqrt{v}$, which means that eq.(2) is nearly optimal. Nevertheless, its best feature is the critical number of failures $f = 2k - 1$, which means that the associated scheme is able to bear up twice the number of failures of its best competitors.

It is a well known fact that the affine plane of order q is obtained by deleting the points of a fixed line from the projective plane of the same order. The distributed coordination procedure for affine plane storage schema is based on this principle in the following way (see fig. 6): every participant generates the lines of a projective plane and then dismisses from it any point belonging to a fixed block, this block is also the difference set that generates the underlying geometry. Then each one recognizes, from this new line, the best active storehouse and sends its local snapshot to it.

```
< 1> Initial conditions
< 2>      j = 1;
< 3>      v' = q^2 + q + 1;
< 4>      let D be the difference set for the p.p.of order q;
< 5> In the j-th storage step
< 6>      let B = {x + j mod v' | x ∈ D};
< 7>      let B' = B − D;
< 8>      let i be the best place such i ∈ B';
< 9>      send local snapshot[j] to i;
<10>      j = (j + 1) mod v + 1;
```

Fig. 6. Which are the storehouses for the next step, in affine plane?

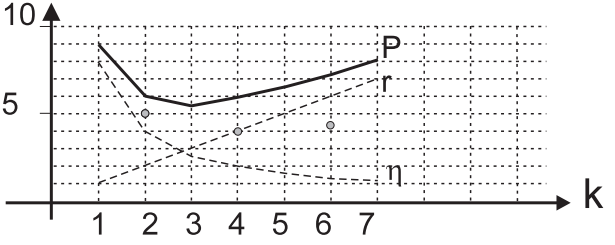


Fig. 7. r, η and P for a s.s. scheme, as functions of k, ($v = b = 8$, $\gamma = 1$ for solid and dotted lines, while $v = 8$, $b = 4$, $\gamma = 2$ for gray points).

6 Discussion and Future Work

Up to now, each storage scheme has been featured using too many parameters apparently unrelated. The goal of this section is to show that any two schema can be compared upon the basis of two parameters only: price and tolerance. As it could be expected, these are conflicting measures and it depends on the user to find a trade- off between them. The following analysis aims to provide a panoramic view where the schema just introduced seem to sketch the space where any solution could be found. This section ends with a short discussion about some possible combinations of the schema already introduced.

Suppose it is required to install a s.s. scheme on a set of v places. From its properties it is granted the construction of an underlying selection strategy with a number of blocks $b = v$, a sliding step $\gamma = 1$ and any block size $k \in [1, v-1]$. Figure 7 shows with dotted lines parameters $r = bk/v$ and $\eta = v/k$ as functions of k for a fixed value of $v = 8$, a number of blocks $b = v$, and a sliding step $\gamma = 1$.

Notice that, for any $k \in [1, v-1]$, $r\eta = b$ which means that in any scheme the amount of information stored in a single place is constant. Does it mean that any solution is equally good?. Price curve shown in solid line (defined in eq. 2) can

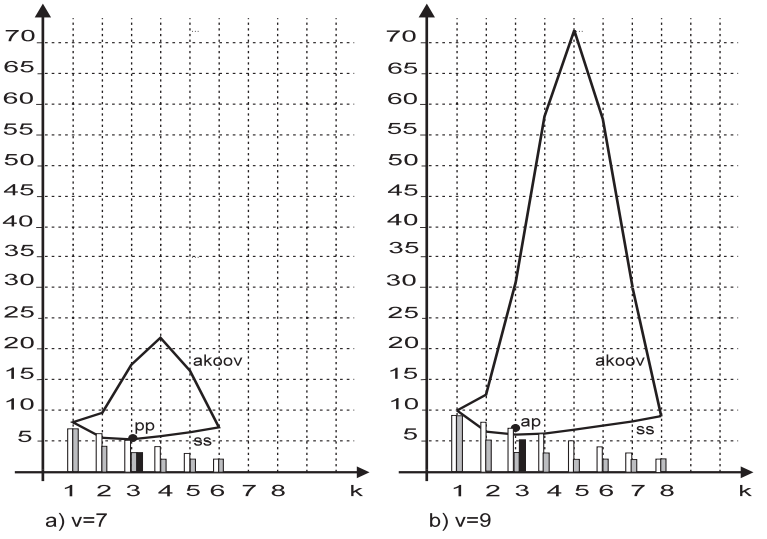


Fig. 8. P and f as functions of block size k, for a.k.o.o.v., s.s., p.p. and a.p.

be understood as a fundamental figure that summarizes different contributions: the required space on each participant, the potential slow of the main execution, individual and collective effort. Therefore, a compromise must be met optimizing the addition of these shaping forces. Gray points show that for the same value of $v = 8$, a sliding step $\gamma = 2$ and some values of k, it is possible to built up balanced and incomplete strategies showing a price below the lower curve, but these are rather isolated points, whereas in any case (i.e. for any $k \in [1, v-1]$) there exists a solution at least as cheap as that of the s.s. curve.

Figure 8 shows price-curves for the all the schema introduced in this work. It is also important to notice that a.k.o.o.v is the only scheme where the number of blocks depends on the block size. It can be shown that any scheme will bear a price lower or equal to that of a.k.o.o.v. under the same conditions. As for the finite geometries family, they only exist like points in the $k - P$ plane, but only for some values of v (shown in black), i.e. projective or affine planes can only be installed when the total number of participants is equal to $q^2 + q + 1$ or q^2, respectively (for instance $v = 7$ and $v = 9$, as shown in fig. 8). In the case where any of these solutions can be compared to the first two schema, their points lie very close to the s.s. curve.

Whatever the price is, there is a related second measure to be taken into account: tolerance, which is measured using parameter f. This parameter, shown in fig. 8 with white, gray or black bars for a.k.o.o.v, s.s. and finite geometries, respectively, evaluates the ability of a scheme to withstand crash failures. In the above paragraph it was mentioned that, for some values of v and k, it is possible to built up some strategies with very low price. On the other hand, as it could be expected, these are also the most vulnerable solutions. Table 1 shows that

Table 1. Best price case

	k.o.o.v s.s.		p.p.	a.p.
	$k = 1$	$k = \sqrt{v}$	$k = \frac{1}{2} + \sqrt{v - \frac{3}{4}}$	$k = \sqrt{v}$
P	$v + 1$	$2k$	$k + \frac{v}{k}$	$2k + 1$
f	v	$\lceil k \rceil$	k	$2k - 1$

under the best available price premise, the more reliable a scheme, the higher its price. It is worth notice that under similar circumstances, affine planes show a tolerance about twice better than that of s.s. or p.p. and with a lightly higher price. It also must be pointed out that, contrary to the other schema, finite geometries like all solutions based on combinatorial designs, show a robustness that directly depends on the block size. Finally, a.k.o.o.v. and s.s. curves meet when $k = 1$ or $k = v - 1$, which in turn means that in these two cases the schema are undistinguishable from each other, as far as price and tolerance is concerned.

Time is a dimension of storage schema barely addressed, but deserving a thorough analysis. In other words, it is quite clear that blocks have to be rescheduled at the end of a cycle, but this condition does not prevent from the possibility of interleaving or shuffling two or more strategies, each one with a different calendar depending on its price-performance figures. Also, it could be possible to find a stochastic rather than deterministic calendar by implementing a Markov chain whose states correspond to the blocks of a particular selection strategy. From this view, participants may have different work rates especially adapted to their capacity, reliability, speed, charge, etc.

6.1 Degradation

Suppose one component crashes after its corresponding scheme has been in operation for at least one cycle. How does it affect the remaining survivors? What happens with more failures? Is there any way out? These questions are analyzed in this section, exploring pros and cons of the addressed solutions.

First, recall that any participant works in r different blocks, which means that in the worst case, the main execution will have to be rolled r steps back. Evidently, the underlying strategy is destroyed. At least three possible action paths can be taken, all of them are corrective, but the last one implies a preventive step:

1. Let the survivors know the identity of the missing component, keep the same scheme but skipping all the blocks that have been touched.
2. Install a new scheme on the survivors with the same or with a new (optimal) block size.
3. Prevent this condition starting with v' components, but installing a scheme as if there were only v participants and keeping $v' - v$ as spare components. If a crash happens, a spare part takes the place of the missing one.

In any case a broadcast operation is required in order to synchronize the corrective step. Nevertheless, some actions can be more effective:

In the first case, the actual cycle reduces to $b - r$ and balance is broken, for there will be some places working harder than others, also bottleneck increases. Each new failure will worsen these conditions. This solution can not be supported beyond f crashes.

In the second case, as well as in the last one, balance is preserved. If optimal block size is recalculated upon each new failure, this solution can be applied up to the moment when only one component survives (whether or not it makes sense for the distributed execution). Obviously, compared to the scheme to be replaced, the new parameters b and f will decrease while r and η will increase. It must be considered that, although the corrective announcement takes a broadcast operation only, installing the new scheme may be a very expensive procedure.

In the last case, the same scheme is preserved with all of its parameters. On the other hand, there is a small balance problem that can be compensated: during normal conditions, spare components do not work in the role of storehouses, but do generate fragments that must be stored somewhere else. Another possible drawback is that a spare can be located away from the component it replaces, which means that an extra communications price should be paid. This solution supports up to $v' - v$ crashes but is totally compatible with the preceding alternatives i.e. case 1 or 2 can be applied, when case 3 does not hold any longer.

Therefore, it looks to be a good design principle to install a scheme on loose conditions to have room for fast, effective solutions based on spare components, while keeping a secondary plan in case all of them were required.

6.2 When Components Are Missed

In many of the variants of distributed storage, there is a file to be stored in a hard disk belonging to a network component. So, if this component has a stop failure, the information retrieval will be canceled. The most frequent solution for this problem consists of storing a backup copy in a second place. The main drawback of this solution is that the alternate place reduces its effective available space in order to locate a file that might never be read from it. Besides, if the first and the second places fail, the file will be lost.

A new procedure is here presented, based on the so-called "Information Dispersal Algorithm" developed by Rabin [15]. Let F be a fragment of a global state to be stored. Using Rabin's algorithm, F could be transformed into $k - 1$ dispersed files. Each of size $|F|/m$, where $k - 1 = m + l$. Then, the original fragment is stored on its due place, while the dispersed files are stored in the remaining $k - 1$ places that built up the corresponding block. From the above explanation, it means that now a block can withstand up to $l + 1$ failures before it could be considered as destroyed.

Since each place works $r = \frac{bk}{v}$ times during a cycle. Now on each step it works as a main storage device for $\frac{v}{k}$ fragments and as a backup device for $v - \frac{v}{k}$ dispersed files. Therefore, the total information it manages will be:

$$\frac{bk}{v}(\frac{v|F|}{k} + (v - \frac{v}{k})\frac{|F|}{m}) = b|F|(1 + \frac{k-1}{m}),$$

which means that each place keeps $\frac{k-1}{m}b|F|$ bits of redundant information. This number must be compared to the $(k-1)b|F|$ bits that should be stored if each fragment were replicated and saved in every place of the block.

When dealing with files, the finite field $\mathcal{GF}(2^8)$ seems to be the best suited for this purpose, since any of its elements can be represented as an eight bits number. Arithmetic in $\mathcal{GF}(2^8)$ can be efficiently accomplished in the following way: All the elements in the field are generated from the primitive polynomial $g(x) = x^8 + x^6 + x^5 + x^4 + 1$, the logarithm and inverse logarithm tables are also constructed, by filling in the corresponding entries of two "char" vectors of size 2^8. These data structures grant a constant access time for any element of the field, which in turn means that any arithmetic operation also takes a constant time.

Information dispersal must be considered as a very robust technique to support the storage of backup copies. It means that the main file can be entirely stored in a single place, while the copy is "split" and "spread" over a set of $k-1$ places. So, retrieval is accomplished reading the file from the main place, unless this has experienced a stop failure. In this case, the high cost of the reconstruction could be accepted due to the "amortized" storage shared among the alternate places, and also due to the superior tolerance it exhibits.

7 Conclusions

Storage schema will become important procedures for massive distributed computing like those in charge of high performance distributed discrete events simulation. In these kind of scenarios, it may be very expensive to restart the execution from zero when a crash failure happens, and storing the "history" of the system is the only way out.

Three different storage schema have been presented in this work. The first scheme, "all k out of v", does not seem to be quite useful, unless the application is prepared to pay a big price in order to tolerate a very large number of failures. In fact, there are some other strategies based on symmetric block designs that offer a tolerance similar to "all k out of v", but are significantly cheaper in total price. The second scheme, "sliding slot", meets its optimal price at the lowest number of blocks. In a sense, "projective planes", from the last family, look similar to "sliding slot", except that "projective planes" only exist for prime powers, while it is always possible to install a "sliding slot" no matter the number of components. The last scheme, "affine planes" is also very close to these two preceding solutions, but it offers a superior fault tolerance figure. It seems to be a good trade-off between price and reliability.

It is very interesting that some of the best solutions come from designs which have strong pair intersection properties, while the original problem from distributed computing does not seem, at first glance, to have any obvious relation

to pair intersection. May this could be proven in future work? Also, it would be interesting to investigate the existence of some other schema.

The revolving order of blocks is a nice property that helps producing the underlying scheme in a very economical way. Besides, random designation also fits well with this procedure. It means, for instance, that rotating an interval set, either with a fixed or random number of steps, is enough to produce the whole sliding slot scheme, also rotating a difference set produces a projective plane. What are the properties of storage schema that are obtained from the rotation of sets that are neither intervals ("sliding slot") nor difference sets ("finite geometries")?

References

1. A. Azagury, M.E. Factor, and J. Satran. Point-in-time copy: Yesterday, today and tomorrow. In *19th IEEE Symposium on Mass Storage Systems*, pages 259–270. IEEE, 2002.
2. M. Chandy and L. Lamport. Distributed snapshots: Determining global states in distributed systems. *ACM Trans. on Computer Science*, 3(1):63–75, 1985.
3. O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, chapter 4, pages 55–96. ACM, 2 edition, 1993.
4. J.B. Fraleigh. *Álgebra Abstracta*. Addison-Wesley Iberoamericana, 1987.
5. S. Rhea et. al. Maintenance-free global data storage. *IEEE Internet Computing*, pages 40–49, sep/oct 2001.
6. F.J. MacWilliams and N.J. Sloane. *The Theory of Error-correcting codes*. North-Holland, 8th edition, 1993.
7. V. Tonchev. *Combinatorial Configurations Designs, Codes, Graphs*. Longman Scientific and Technical, 1988.
8. D.R. Stinson. An introduction to combinatorial designs. Technical report, University of Waterloo, Dept. of Combinatorics and Optimization, dec 1999.
9. C.J. Colbourn, J.H. Dinitz, and D.R. Stinson. Applications of combinatorial designs to communications, cryptography, and networking. Technical report, University of Vermont, 2000.
10. D. Kotz. Introduction to multiprocessor i/o architecture. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic Publishers, 1996.
11. A.N. Mourad, K.W. Fuchs, and D.G. Saab. Site partitioning for redundant arrays of distributed disks. *Journal of Parallel and Distributed Computing*, (33):1–11, 1996.
12. P. Berenbrink, A. Brinkmann, and C. Scheideler. Design of the presto multimedia storage network. In *International Workshop on Communication and Data Management in Large Networks*, pages 2–12, 1999.
13. A. Russell and A. A. Shvartsman. Distributed computation meets design theory: Local schedulling for disconnected operations. *Bulletin of the EATCS*, pages 120–131, Jun 2002.
14. R. Bhagwan, D. Moore, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer storage. In *International Workshop on Future Directions in Distributed Computing*, 2002.

15. M. O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, Apr 1989.
16. D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, dec 1995.
17. V.K. Garg and J. Ghosh. Repeated computation of global functions in a distributed environment. *IEEE Trans. on Parallel and Distributed Systems*, 5(8):823–834, Aug 1994.
18. N. Lynch. *Distributed Algorithms*. Morgan Kaufman Pub., 1996.
19. Y. Deswarte. Tolérance aux fautes, sécurité et protection. In R. Balter et. al., editor, *Construction des Systèmes d'exploitation Répartis*. INRIA, Paris, France, 1991.
20. D.L. Kreher and D.R. Stinson. *Combinatorial Algorithms*. CRC Press, 1998.
21. Lynn Margaret Batten. *Combinatorics of finite geometries*. Cambridge University Press, 1997.
22. P. Dembowski. *Finite Geometries*. Springer-Verlag, 1968.
23. J.N. Cederberg. *A Course in Modern Geometries*. Springer-Verlag, 1989.
24. A. A. Bruen and R. Silverman. Acs and blocking sets ii. *European Journal of Combinatorics*, 8(4):351–356, 1987.
25. A.E. Brouwer and A. Schrijver. The blocking number of an affine space. *Journal of Combinatorial Theory ser. A*, 24(2):251–253, 1978.

DisCusS and FuSe: Considering Modularity, Genericness, and Adaptation in the Development of Consensus and Fault Detection Services

Lásaro J. Camargos* and Edmundo R.M. Madeira

IC - Institute of Computing
UNICAMP University of Campinas
Avenida Albert Einstein, 1251
PO 6176, 13083–970, Campinas, SP - Brazil.
Phone: +55 (19) 3788-5839. Fax: +55 (19) 3788-5847
{lasaro.camargos,edmundo}@ic.unicamp.br

Abstract. Although fault tolerant systems are badly needed, their development is not a trivial task. Considering fault tolerant distributed systems, this difficulty is even bigger, and any artifact that could make this task easier becomes highly valuable. In this paper, we propose and model a *dis*tributed *c*onsens*us* *s*ervice and a *fau*lt detection *se*rvice, namely, DisCusS and FuSe, that can be used as building blocks in the development of distributed fault tolerant applications. We also show the compliance of FuSe to FT-CORBA fault detection, and give some insights on the use of the proposed consensus service in a possible FT-CORBA implementation. Moreover, this paper presents some comparative tests of the influence of adaptive and *non*-adaptive fault detectors over consensus.

Keywords: Distributed Consensus, Failure Detectors, Performance, Fault Tolerant CORBA.

1 Introduction

Fault tolerant systems are highly necessary but their development is not a trivial task. The complexity is increased even more if we consider distributed fault tolerant systems, because the difficulty inherent of distributed systems is aggregated to fault tolerance requirements. Any artifact that could make the development of such systems easier becomes highly valuable. Some of these artifacts act providing services like fault detection and agreement support, and can be used as building blocks, providing an infrastructure to applications. But even these services can be composed of smaller pieces, and the capability of selection over these pieces can lead to better results on their utilization. This paper proposes

* Work supported by CNPq, process 133415/2001-5.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 234–253, 2003.

an architecture for a consensus and a fault detection services composed of selectable parts and presents some tests that help on the task of selecting these parts.

The *Consensus* problem is well-known as the greatest common denominator of several agreement problems, such as Atomic Commitment [13] and Total Order Broadcast [4], and is often present in the development of distributed fault tolerant applications [12]. Fischer, Lynch and Patterson [10] have shown that consensus cannot be deterministically solved in asynchronous environments where processes are subject to crash faults. This impossibility stems from the difficulty of determining whether a process is faulty or just very slow. On the other hand, Chandra and Toueg [4] have shown that distributed consensus can be achieved even in asynchronous systems, by employing *Unreliable Failure Detectors*, a distributed oracle that tries to guess the states of processes.

The work of Guerraoui and Schiper [15] proposes an architecture for a generic consensus service that can use any failure detector available in a given system. This work, however, does not address the actual implementation of this service and only gives ideas of how it could be used. Some works have used CORBA [19] as infrastructure. The Common Object Request Broker Architecture (CORBA) is an infra-structure for distributed systems that provides language and localization transparency when invoking methods in distributed objects. The Fault Tolerant CORBA [20] (FT-CORBA) is the specification for fault tolerance mechanisms in CORBA. The use of CORBA allows the construction of really reusable software. Felber [9] has proposed and implemented consensus and failure detection services as the base for an object replication service, both of them in CORBA. This work is very comprehensive and detailed, but it is prior to FT-CORBA and, therefore, not compliant with that specification. Furthermore, it defines a monolithic failure detection service that does not take adaptation on failure detection parameters [26,25,2] on the monitored process side [26,25] into consideration.

In this paper, we propose and model a distributed consensus service and a fault detection service, namely, DisCusS and FuSe. These services have been designed with the aim of being generic, highly configurable and scalable. Generic in the sense that they can be used to solve diverse agreement problems, using several different algorithms, both in consensus and in fault detection levels; highly configurable because of their strong modularization, which allows the exchange of basic blocks and the use of different ones at the same time, e. g., using different failure detectors to monitor objects in different hosts; and scalable, due to the hierarchical arrangement capabilities in fault detection. We also show that our service is compliant to FT-CORBA fault detection, and give some insights on the use of the proposed consensus service in a possible FT-CORBA implementation.

In the literature, there seems to be a trend toward making the modules responsible for consensus and failure detectors independent [9,15,4,21]. When real systems and performance factors are taken into account, however, we believe that they should be considered closely related [8,23]. The behavior of failure detectors can interfere on consensus through many factors, e.g., the network and

processing contention. We are interested in these interferences, specially in the differences between the impact of adaptive and *non*-adaptive failure detectors on consensus. In the literature, there are few evaluations of the influence of failure detectors over consensus and even fewer if we consider only systems that were actually implemented. This paper presents some comparative tests of the influence of failure detectors on consensus. We also present a model with two layers (consensus and fault detection), which were made independent with the aim of providing a higher configurability degree, but that work very closely, for performance purposes.

The rest of this paper is organized as follows. In Section 2 we present some basic concepts. Some related works are presented in Section 3. The architecture for the proposed services is depicted in Section 4. A proposed object-oriented modeling for the architecture is given in Section 5. Sections 6 and 7 present some aspects of the prototype implementation and some evaluation tests done on it, respectively. Final conclusions are given in Section 8.

2 Basic Concepts

In this Section, we firstly present some basic concepts about the consensus problem and then on failure detectors.

The consensus problem is defined as follows: each process p_i in a given set P of processes has an estimate v_i of some variable, and it is necessary that all of them agree on a common value v equal to some v_j, previously estimated.

Traditionally, consensus algorithms are defined over two primitives: the first, propose(v_i), is responsible for consensus instantiation, and the second, decide(v), for gathering the consensus result.

Fischer, Lynch and Patterson [10] have proved that the consensus problem cannot be solved in asynchronous environments with crash-stop faults (without recovery). Dolev *et al.* and Dwork *et al.* [6,7] have worked on the synchrony needed to solve consensus. These works show themselves as too complex, defining several classes of synchronism, from asynchronous to totally synchronous. Chandra & Toueg [4] have introduced the *unreliable failure detectors* in order to circumvent the impossibility of consensus on asynchronous environments. Working as oracles trying to guess if a process is faulty or not, failure detectors work sending periodic *signals* from a *monitored* process to a *monitor* one. Failure detectors can be seen as a surrogate for synchronism in the system, abstracting the classes of synchronism pointed out in [6,7]. The *inter-sending* interval and *timeout* parameters dictate the periodicity of signals and how long a monitor should wait for them.

Failure detectors are classified according to their ability to correctly identify non-faulty processes (*accuracy*) and faulty ones (*completeness*). The simplest failure detector that can be used to solve consensus is called $\diamond S$ [4], and it satisfies two properties: i)a correct process eventually suspects every faulty process (*strong completeness*); and ii)there is a time after which a correct process is not suspected by any non-faulty one (*eventual weak accuracy*). In fact, if these

properties hold within a long enough period of time, a consensus instance can be solved.

Real systems are subjected to variable workload, and the monitoring tasks are also processor and network consumers. So, to prevent resource exhaustion and performance degradation of the system as a whole, failure detectors must have the ability to dynamically adjust their monitoring parameters (inter-sending interval and timeout, as previously mentioned). Failure detectors which present this feature are called *adaptive* [26,25,2].

It is important to to mention that in this paper we will use fault and failure in an almost interchangeable way, since the difference depends only on the abstraction level in which the observer is. So, we will keep using *fault* (the CORBA nomenclature) when referring to the developed service and to the crash of processes, and failure when referring to detectors (mainly for historical reasons).

3 Related Work

In the architecture level, two works are closely related to ours: the Generic Consensus Service and the Object Group Service. The Generic Consensus Service [15] (GCS) provides a generic interface for solving agreement problems. A specific module in GCS translates the given problem in an instance of consensus. GCS uses any failure detector available in a given system and provides, based on these detectors, different properties on consensus level, *i.e*, depending on the class of the failure detector, different properties at consensus are met, for example, the number of tolerated faults. The idea of a generic consensus service has strongly influenced our work. However, we are interested in modeling consensus and detection services at a lower abstraction level than the one presented in [15], keeping the genericness and also defining the application programming interfaces.

The Object Group Service [9] (OGS) is a CORBA service for active object replication, which uses both a fault detection and a consensus service. The fault detection service of OGS is not, however, compliant with the FT-CORBA specification, actually, it is prior to it. While the OGS consensus service (OConsS) is planned to give support for the replication service, our work focuses on providing a couple of services that can have their components selected for specific environments, and with an interface that is generic enough to solve a wide range of agreement problems. To use OConsS in an application other than OGS, one has to specialize the protocols it uses, due to its design decision of letting consensus to decide over application specific data. On DisCusS, on the other hand, the application data is inside a bigger structure containing an identifier of the data (e. g., its hash code) and the data itself (See Annex A.4.). Of course, specializations that directly use the data are also implementable. Furthermore, OGS's fault detection service does not employ adaptive fault detection support.

The General Agreement Framework [18] (GAF) gathers several optimizations proposed over the consensus algorithm by Chandra and Toueg [4] and associates them with *versatility* parameters. While this approach is very useful, in the

sense it allows the construction of *ad hoc* protocols, it lacks some optimizations proposed in other algorithms, such as the one proposed by Schiper [21], that lowers the dependency of consensus participants on a coordinator, and the one proposed by Brasileiro *et al.* [3], which allows the decision to be made in one communication step. Furthermore, taking into account the great number of systems developed using CORBA, the framework approach is more restrictive, from a code reuse perspective, than the service approach.

The feasibility of selecting among different consensus algorithms is attested by the number of algorithms present in the literature. Chandra and Toueg [4] proposed a simple algorithm that centralizes the decision on a *coordinator* process. Several optimizations were proposed over this algorithm [18] allowing it, for example, to fast terminate or to decide over a set of values instead of just one. Schiper [21] proposed an algorithm that minimizes the dependence on the coordinator, allowing, in certain circumstances, the consensus achievement even if it becomes faulty. The algorithm proposed by Oliveira [14] works over *stubborn* communication channels, a kind of channel that limits its message buffer. Lamport's Paxos [16] is an algorithm that supports *crash recovery* faults and that keeps safety in decision even if several processes consider themselves coordinator at the same round. Brasileiro [3] points out an algorithm that allows, if some specific conditions are met, the decision in only one communication step. All of these algorithms can be implemented under the architecture we propose.

Some failure detection algorithms have taken into account the performance of the global system. Chen *e*t al. [5] presented an algorithm that estimates a new timeout value, based on the delays of the last messages and dynamically alters it. Furthermore, it presents an estimator that takes some parameters of the network and a set of quality of service requirements as inputs, and produces inter-sending and timeout intervals that satisfy those requirements in the given network, as outputs. The proposal from [25,26] changes, besides the timeout, the inter-sending interval, and tries not only to cope with variable message delay but also tries to minimize the contribution of the failure detector in the system degradation. Sergent *e*t. al. [23] have proposed specialized detectors, which work closely to its client application (e.g., consensus). They are informed by these applications about the periods and whom to monitor, eliminating unnecessary message exchanging. All of these detectors are easily implemented in the model we propose.

In the literature, there are few empirical evaluations of the influence of failure detectors on consensus performance. Sergent *e*t. al. [23] have performed some simulations and Esteffenel [8] reports measurements in an actual system. We report here independent measurements made on a test implementation of our proposed model. We have no knowledge of other studies comparing the impact of adaptive and non-adaptive failure detectors on consensus as the one presented here.

4 Services Architecture

Due of the low information exchange and the relative independence between consensus and failure detection problems, we have explicitly divided them in two services. This design allows not only the substitution of objects within a service, but of the service as a whole. Following this approach, one can set up the services even in different stations, with different characteristics. For example, one could place failure detectors in a network without reliable channels [8] but with greater bandwidth or even in a synchronous network, providing more quickly and accurate detection, while placing consensus on a more reliable network. The only inconvenience of this approach is the addiction of another indirection among the objects of the system, leading to a slightly degraded performance when compared to a monolithic approach.

The Fig.1 presents the architectural view of the proposed fault detection and distributed consensus services. Some duplicated objects and relationships were omitted to simplify the visualization.

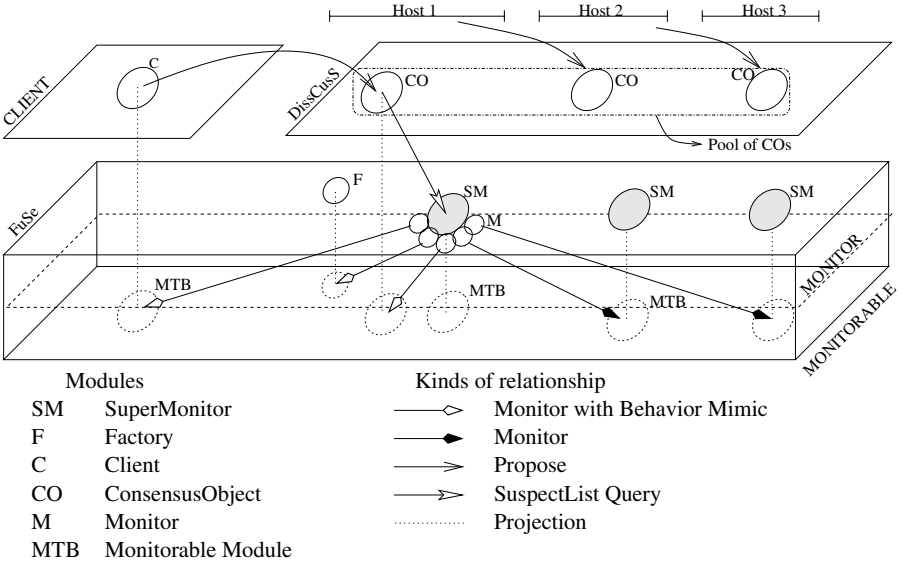


Fig. 1. Architecture.

4.1 *Fau*lt Detection *Se*rvice – FuSe

Every object in the system has a **Monitorable** (MTB) projection in the FuSe, the fault detection service. This projection is used by the **SuperMonitor** (SM) for noticing, or at least trying to, any faults on objects. Actually, the **SuperMonitor** delegates the responsibility of monitoring to a set of **Monitor** objects. Due to a generic interface and the use of factories [11] of monitors, this

set may contain different implementations of fault detection algorithms. This feature makes it possible to monitor different objects, which present different properties, using different monitors, and hides this heterogeneity from the upper levels, such as the consensus one (see Fig.1).

Faults on **Factory** objects are handled by the use of a **Monitor** created by a different **Factory**, in order to ensure that a faulty **Factory** does not go unnoticed due to faults on its **Monitor** objects.

An important feature of our design is the behavior mimic that the **SuperMonitor** performs. If an object is so important that a fault on it must be considered as a fault on a whole group, the **SuperMonitor** responsible for this group may mimic its fault behavior. To exemplify this feature, consider the objects of the *Host 1* on Fig. 1. If the consensus object on this host fails, the host becomes useless until the object is up again. In this situation, **SuperMonitor** must mimic the CO's fault, forcing its own **Monitorable** projection to act as if it was faulty, *i.e.*, stopping sending monitoring messages. In this manner, objects can be monitored either individually or in groups, allowing flexible hierarchical arrangements that provide an arbitrary degree of granularity for fault detection.

All **ConsensusObject** and **Factory** objects in the architecture are monitored with mimic, since **ConsensusObject** are essential and faults on **Factory** could mask faults on that objects. They must be replaced as soon as the **SuperMonitor** detects their faults.

4.2 *Dis*tributed *C*onsens*us S*ervice – DisCusS

The consensus service considers the implementation of consensus algorithms based on failure detectors (implemented within FuSe). The **ConsensusObject**s (see Fig. 1) act as clients for the fault detection service, and take to themselves the responsibility of reaching an agreement, based on the proposals of the actual **Client** objects. Both PULL and PUSH models may be used for querying the list of processes suspect of being faulty, which is used by consensus objects. The relationship between **ConsensusObject**s and **SuperMonitor**s is not necessarily 1-to-1. In the same host, for example, several instances of **ConsensusObject**s could share a same **SuperMonitor** responsible for monitoring all **Factory**s. Furthermore, these **ConsensusObject**s instances could still use other **SuperMonitor**s to monitor objects on other hosts. In this case, a many-to-many relationship is set.

The decoupling between clients and the actual consensus solvers is a very important feature. This decoupling is possible because **Clients** only get involved at the beginning and at the end of the algorithm. So, as soon as **Clients**' opinions have arrived (or the **ConsensusObject** has decided to use a default opinion), they don't need to be monitored anymore, freeing resources and allowing disconnections, programmed or not, of clients (from the point of view of clients, proposition and decision are completely asynchronous operations). Due to this design, even clients spread through the Internet or in a mobile network could use the service. Furthermore, this design allows the placement of **ConsensusObject**s on a well-known and controlled network where, for example, stronger

timing assumptions could be made, simplifying the implementation of failure detectors and increasing the possible quality of service offered.

The association between **ConsensusObjects** and **Client** objects is known by both sides, allowing explicit and implicit consensus instantiation. Respectively, explicit invocation occurs when all clients send their opinions to the pool of **ConsensusObjects**, and implicit invocation occurs when objects in the pool have to query clients for their opinions [9].

5 The Proposed Model and FT-CORBA

This section presents a mapping from the proposed architecture to an object oriented model, and relates this model to the FT-CORBA specification.

5.1 The Model

Mapping the proposed architecture to an object oriented model is straightforward. The UML class diagram [1] in Fig. 2 presents our modeling.

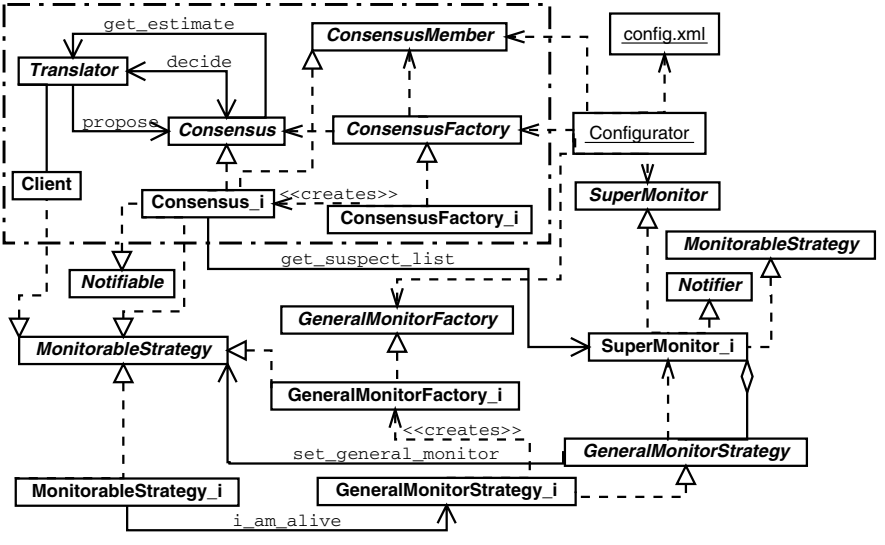


Fig. 2. The proposed model.

In this diagram, classes with suffix *_i* are implementations of the homonym interfaces, without the suffix. The **client** side of the architecture (see Fig.1) is mapped to a **client** object that is associated with the `Translator` interface. **ConsensusObject** is mapped to a **consensus_i** object, that implements `ConsensusMember`, `Consensus` and `Notifiable` interfaces, in order to achieve

communication with clients, with other consensus objects and fault monitoring reports, respectively. **SuperMonitor** objects are mapped to **SuperMonitor_i** objects, which implement the `SuperMonitor` and `Notifier` interfaces. In this manner, a **SuperMonitor** can manage monitors and notify faults to consensus implementations. **Monitor** and **Factory** objects are mapped to `GeneralMonitorStrategy` and `GeneralMonitorFactory` implementations. The object's monitorable projection is given by the `MonitorableStrategy` interface implementation. IDLs for the remote accessed objects are presented in the Annex A.

A **Configurator** object is responsible for building the initial relationships among the other objects. For example, it is responsibility of the **Configurator** to inform the **SuperMonitor** which objects it must monitor. This information is extracted from a configuration file, *config.xml*.

Design patterns [11] play an important role in our model. Thanks to the Strategy pattern, both failure detection and consensus algorithms are hidden by a generic interface. Furthermore, they are easily interchangeable, providing means for quick reconfiguration. The use of abstract factories [11] and a manager [24] for failure detectors allow several different detectors to be used at the same time. From the monitored object side, the same object can respond to different detectors, which can have distinct quality of service requirements for fault detection over this object.

Specifically to the fault detection service, this model allows the use of both PUSH and PULL models for querying for faults. Thanks to the `Notifier` interface, a `Notifiable` object may register itself to receive fault reports or just poll for a list of fault suspected processes.

The `MonitorableStrategy` interface defines a method that allows the dynamic adjustment of the inter-sending interval (See Annex A.3.) of monitored objects, giving support to the use of adaptive failure detectors like ADAPTATION [26]. On the monitoring side, adaptive detection can be achieved by simply adjusting the internal state of the monitor.

In the consensus service, the complexity of the agreement problem is hidden from the clients by the `Translator`. Implementations of `Translator` are totally problem dependent and must convert the problem to be solved in an instance of consensus. The problem dependent data and an identifier to this data must be generated by `Translator` implementations and put inside a `Value` type variable. The identifier may be a hash number of the data (allowing the comparison of the data), the process id or any value suitable for the consensus algorithm. `Translator` also implements the asynchronism noticed between clients and **ConsensusObjects**. The result of an instance of consensus can either be called back to `Translator` or polled by it from the consensus implementation.

Relationships between `Consensus` objects have not been defined, since this relationship is highly dependent on the implementation and on the chosen algorithm. Defining strict means for communication among `Consensus` objects could limit their application. Defining these means and still allowing the addition of others is not justifiable. So, we have opted to let these definitions to the developer.

5.2 Meeting FT-CORBA

Now, we show a brief introduction to FT-CORBA mechanisms and point how a consensus service following our model could be used to implement the FT-CORBA specification.

The key feature of FT-CORBA is the object replication. A group of objects acts as if it was a single object. The replicas are controlled by a **Replication Manager**. Two levels of fault detectors (local and global) provide a **Fault Notifier** with fault information and the **Fault Notifier** sends compiled fault reports to the **Replication Manager** and any other registered object. Application-specific fault analyzers can also be registered for fault notification. These analyzers may be used to correlate and filter this information.

The hierarchical model for fault detection in FT-CORBA is totally compatible with the proposed architecture. Objects can be monitored by distinct detectors, making easy the replication of global fault detectors. A **Fault Notifier** can be implemented as an specialization of a fault detector, since fault detectors are ready to receive and dispatch fault notifications through the simple interfaces `Notifier` and `Notifiable` (See Annex A.3). These interfaces can substitute or work in parallel to the interfaces defined in FT-CORBA specification.

The FT-CORBA fault detection is based on the PULL model, *i.e.*, the fault detector periodically invokes an `is_alive()` method on the monitored object. An anticipated extension for PUSH model is included in the FT-CORBA specification and [26] has proposed extensions for adaptive variants to both PUSH and PULL. Our definitions for fault detectors have taken into account both, PULL and PUSH models, and even these adaptive alternatives for fault detection (See Annex A.2). The FT-CORBA `PullMonitorable` interface is an specialization of our `MonitorableStrategy`.

As defined in the specification, the **Fault Notifier** and the **Replication Manager** must be replicated and still act logically as one object. Consensus algorithms can be used to maintain the consistency among the replicas, mainly on the group membership control performed by the **Replication Manager**.

Consensus can also be used to achieve consistence in the logs on recovery mechanisms, to atomically deliver messages to replicas, to coordinate actions after the detection of a fault, and to implement (or just substitute) the `ACTIVE_WITH_VOTING` [20] replication style.

6 Implementation Issues

Our prototype has been implemented using ACE+TAO [22], a CORBA 2.6 ORB compliant. TAO has its development focused on *high-performance* and *real-time* distributed applications, that perfectly fits our needs. As TAO and ACE (ACE stands for Adaptive Communication Environment, and is the framework that TAO is built on top), our implementation has been done in C++. The use of ACE has simplified the resolution of some problems as thread, active object and queue implementations.

TAO's *Naming Service* has been used for object locating although it is not fault tolerant. We have assumed a stable period in the beginning of the operation of the system. This stability is necessary to establish the communication among the several objects that compose the system. Even if we do not extend this initial stability restriction to the Naming Service operation, a fault tolerant one can be used (*e.g.*, [17]).

In this first implementation, on the consensus level, Schiper's *early consensus algorithm* [21] was used. The reasons of this choice are given on the next Section. The algorithm has two tasks that work waiting for a consensus decision message arrival and actually trying to perceive consensus on their own. Hence, the algorithm considers the delivery of specific messages, delaying (sometimes forever) the messages that are not expected in the moment. A delivered message is matched to a block that will correctly process it. In the implementation, the task waiting or a decision message became another block on the second task, minimizing the overhead caused by the concurrency.

Decoupling the execution of several instances of consensus can lead to performance improvements. So, although this feature has not been used in the tests presented here, concurrent consensus has been implemented. A central function multiplexes and queues up all the messages for consensus tasks based on the `ConsensusID` they carry. Messages are removed inside the task and discarded, if it is too old to be useful, or re-queued, if it could be necessary in the future.

7 Tests

We have carried out some tests to measure the influence of failure detection in consensus solving. The use of resources (processing, memory, network) by the detection infrastructure conflicts with the consensus service. The use of contention aware metrics [27], such as *latency* and *throughput*, considers these conflicts, and this is the reason why they have been chosen. In fact, the work on [27] has an analytical view of these metrics, while we work with measurements.

As could be expected, the more the network is used, the less the consensus throughput. In periods of high usage of the network, monitoring messages contribute augmenting this usage even more. Increasing the inter-sending interval during these periods and decreasing it after, seems to be a good idea. Sotoma and Madeira [26] have proposed a failure detector that works in this way. Basically, the algorithm tries to estimate a new inter-sending interval based on the delay suffered by the messages, increasing this interval if the messages are delayed, and decreasing it if messages arrive earlier than expected. The estimation is based on the history of message arrivals. In these tests we try to measure if this approach is really suitable. The obtained results are compared to a *heartbeat*-like failure detector. Both detectors have been implemented in the same way. Actually, the heartbeat-like is just a simplified (without adaptation) version of the other.

In the consensus level, as aforementioned, Schiper [21] algorithm was used. Due to its decentralized and broadcast based message pattern, this algorithm

became more sensitive to the network traffic, allowing a better observation of the influence of the contention on network due to failure detectors, on consensus.

All the tests have been done using Intel Celeron-1200MHz stations, running Linux (kernel 2.4.18), over a 10-BaseT network in low usage periods (0.2 ms of maximum round-trip).

7.1 Tests on Consensus

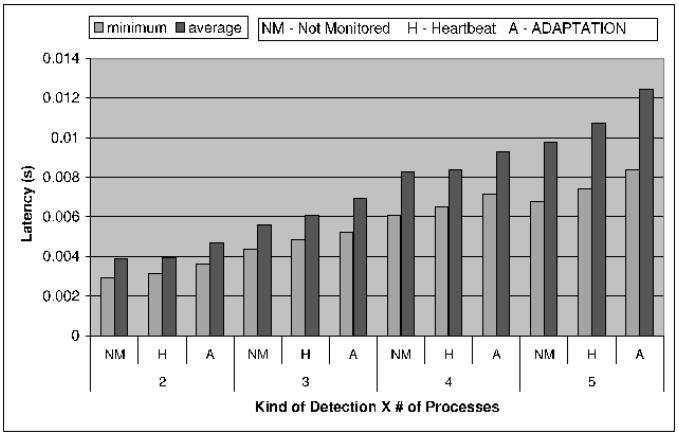
The tests on consensus were carried out on two scenarios: A) *with no fault* and *no wrong suspicions* (all of them were discarded); and, B) *with coordinator faults* and *no wrong suspicions*. It is remarkable to say that in B), all the faults occurred on the coordinator, the process that tries to impose its estimative to the others, before the consensus launch. So, when the consensus had started, the fault had been detected by all the detectors.

The results presented comprise the average from at least a thousand of consensus operations running sequentially. Since our service could be used to implement replicated objects, and since the first response is all the client needs to wait in many applications (replicated servers, for example), the time the first consensus object spent to reach the agreement is also shown in the results.

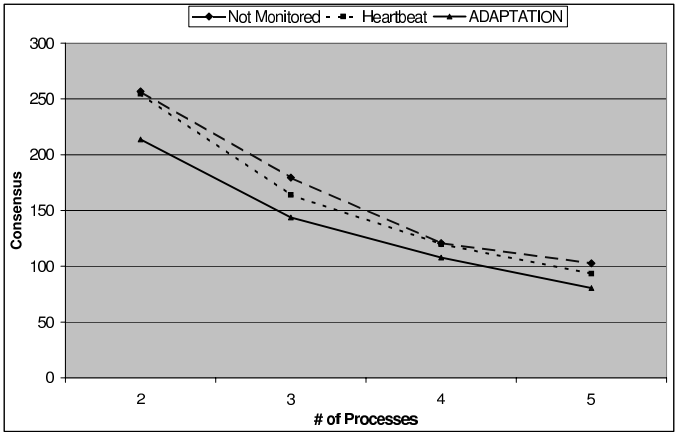
Because the clocks of the stations involved in the tests were not perfectly synchronized, there was a necessity of a start message to synchronize the executions. Rather than using an external application to this goal, we have opted by the run of n (the number of members) consensus executions before the actual measured executions. Every process coordinates an instance of consensus in these n runs, even the latest one. In the absence of faults all the other processes have to wait for the coordinator to proceed to the next execution. So, the decision message sent by the latest process to start to work when it is the coordinator is used as a synchronization message.

The graphic in the Fig. 3.a shows the latency when solving consensus in the first scenario, with two to five stations involved. Tests were run over a heartbeat-like (H) and over the ADAPTATION (A) failure detector. A schema with no actual monitoring (NM) has been added to give comparing parameters among the algorithms. The inter-sending interval to the heartbeat detector has been set up to 20ms and its timeout to 40ms. For the ADAPTATION detector, the lower bound for monitoring interval has been set with the same value, 20ms (remember that the round-trip is a hundred of times lower). These values were chosen not to flood too much the network with monitoring messages, and keep justice in comparisons. Remember that wrong suspicions are all discarded in this test, and so, the unique conclusion that can be drawn is about resource contention.

It can be easily noted that the monitoring schema has influence over the consensus latency, even in an environment without background traffic. The constant relation NM<H<A can be explained by the number of messages and resources the detectors use. ADAPTATION algorithm uses 33% more messages than heartbeat one (it estimates the new inter-sending interval based on the last 3 messages), and uses more processing time to calculate monitoring interval and for process control. Of course, the greater the number of process, the greater the



(a)

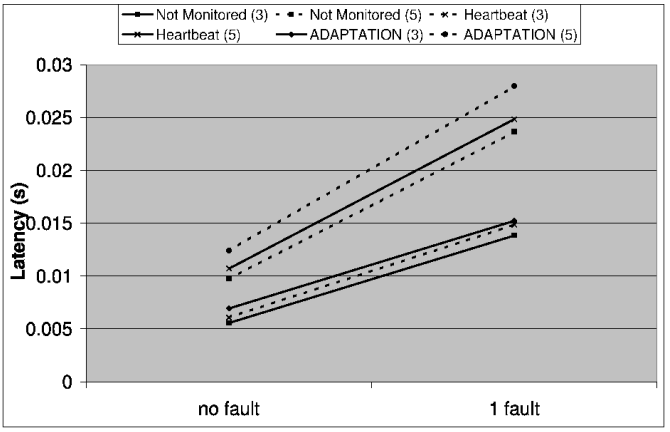


(b)

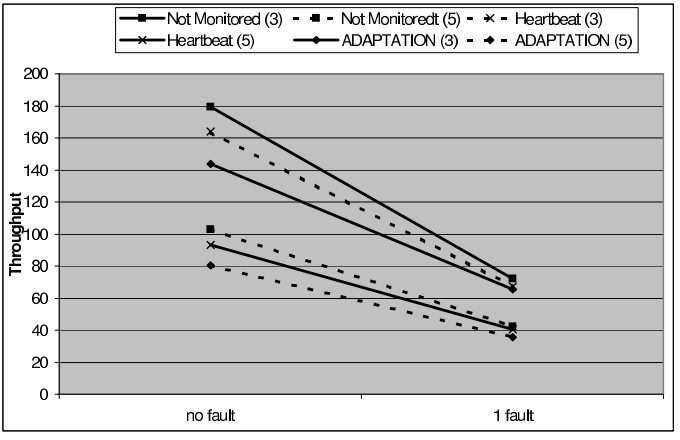
Fig. 3. Tests without faults and without wrong suspicions. Througput is measured in consensus per second.

latency. This can be explained by the growth in the number of messages needed to achieve consensus and the exponential number of monitoring messages and processes. Figure 3.b presents the consensus throughput in the first scenario.

A fault forces the consensus objects to a new round, with a new coordinator. Since the failure detectors are not reliable, a majority of objects has to decide to pass to the next round. The cost of doing such transition was measured on tests in the second scenario. The results are evidenced in Fig. 4.



(a)



(b)

Fig. 4. Tests with coordinator faults and without wrong suspicions. Througput is measured in consensus per second.

The throughput and latency running three and five consensus objects, the values which make sense with faults and less than seven stations since four and six would not increase the number of faulty stations permitted, show a great decrease in performance (up to twice times). The cost derived from faults cannot be discarded. Wrong suspicions, however, bring unnecessary costs (it can make a round inviable or increase the latency in communication due to the messages it results), and minimizing these wrong suspicions is a great goal when using failure detectors.

7.2 Tests on Fault Detectors

Since wrong suspicions have a high cost to consensus, we have made some comparisons between the implemented failure detectors, to measure the frequency of wrong suspicions in each one.

Keeping the lower bound of 20 ms in the *inter-sending interval* for the ADAPTATION algorithm, we have run several tests. Because of its adaptive nature, ADAPTATION has varied the inter-sending interval many times during the execution, keeping an average value of **20.44 ms**. This was the value used as inter-sending interval in the heartbeat-like failure detector (actually, **23 ms** have been used). In this way, both algorithms would report real faults within the same time, allowing the use of the number of wrong suspicions as comparison parameter.

Figure 5 presents the occurrence of fault suspicions in a period of eight minutes. These tests were run during a normal utilization period and in parallel.

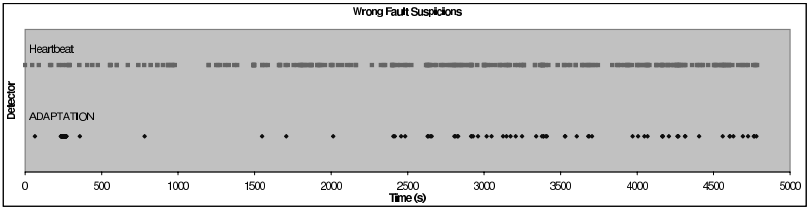


Fig. 5. Wrong suspicion occurrences.

The line representing the occurrence of fault suspicion in the heartbeat-like detector is clearly more dense than the ADAPTATION one. Heartbeat has committed 262 wrong suspicions, against 64 committed by the ADAPTATION. A detector with high number of fault suspicions is likely to force a consensus algorithm to discard rounds. The obvious conclusion is that in these environments, the ADAPTATION failure detector will lead to better performance on consensus.

The graphic in Figure 6 shows the latency when running consensus involving five processes in an environment without faults, with variable workload and subject to wrong suspicions. The workload was generated by the instantiation of processes following the Poisson distribution. CPU utilization rate was kept in an average of 90% and 100% for medium and high workload.

One can see by Fig. 6 that ADAPTATION had a slightly better performance than heartbeat. In the long term, this little difference implies in a greater difference in the total of solved consensus instances.

8 Conclusion

In this paper, we present an architecture for *Distributed Consensus* and *Fault Detection* services. Due to the importance of the consensus problem for several kinds of application, we believe this architecture can be very useful in the development of fault tolerant systems.

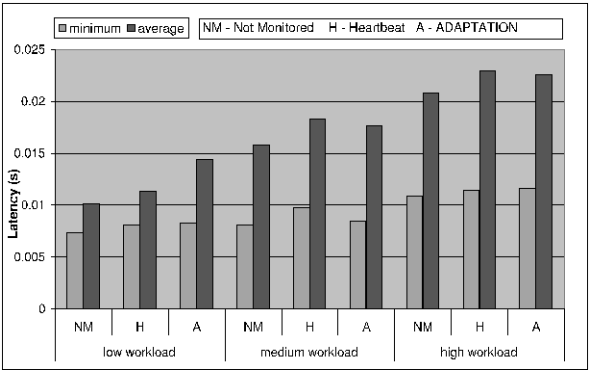


Fig. 6. Consensus latency(s) in presence of variable workload.

Some design issues of the architecture are specially interesting: i)the relative decoupling of consensus and fault detection services allows their use in an independent way; ii)the fault behavior mimic, implemented by the **SuperMonitor**, allows different organizations in the monitoring schemes such as hierarchical (as in FT-CORBA) or direct ; iii) the application of the *Strategy* pattern in monitor and monitorable objects makes it possible to use different failure detection algorithms simultaneously; and iv) the dissociation of consensus client applications and consensus objects allows *physical* independence between them. For instance, the pool of consensus objects can be located in a highly available network. A mapping from this architecture to an object-oriented implementation and some discussion about the choices made were also presented.

Some insights on how this architecture could be employed to implement FT-CORBA were given. In fact, even an extended version of such specification could be achieved, considering for example, the use of adaptive fault detection. Furthermore, the defined services could be used directly by applications, allowing the implementation of several distributed protocols.

Finally, based on tests carried out over a prototype implementation of the model, we have shown that i)the fault detection mechanism has influence over the consensus performance; ii) a fault occurrence or a wrong suspicion can have a high cost to the consensus execution; and iii) in systems subject to variable workload, it is possible to adjust the parameters of detection at run-time, minimizing the occurrence of wrong fault suspicions and their impact on consensus. The last conclusion is based on the comparison made between a *heartbeat-like* failure detector (a very common approach) and the ADAPTATION failure detector [26], which shows that the latter presents a much lower number of wrong suspicions.

References

1. Oestereich B. *Developing Software with UML.* Addison-Wesley, Harlow, England, 1999.

2. Marin O. Bertier M. and Sens Pierre. Implementation and performance evaluation of an adaptable failure detector. In *Proc. of the Int. Conference on Dependable Systems and Networks (DSN'02)*, page 354, Washington, D.C., USA, June 2002.
3. F. Brasileiro, F. Greve, A. Mostefaoui, and M. Raynal. Consensus in one communication step. In *6th Parallel Computing Technologies, 6th International Conference, PaCT 2001*, number 2127 in LNCS, pages 42–50, Barcelona, Spain, September 2001. Springer Verlag.
4. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
5. Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.
6. Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.
7. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
8. L. A. B. Esteffanel and Ingrid Jansch-Pôrto. On the evaluation of failure detectors performance. In *Proc. of IX Brazilian Symposium of Fault Tolerant Computing*, Florianópolis, Brazil, March 2001.
9. P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.
10. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.
12. Felix C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, 1999.
13. R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proc. of the 9th International Workshop on Distributed Algorithms (WDAG-9)*, number 972 in LNCS, pages 87–100, Le Mont-St-Michel, France, September 1995. Springer-Verlag.
14. R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn communication channels. Technical Report 98/272, École Polytechnique Fédérale de Lausanne, Switzerland, March 1998.
15. Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.
16. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.
17. C. Lung, J. Fraga, Jean-Marie Farines, M. Ogg, and A. Ricciardi. Cosnamingft - a fault-tolerant corba naming service. In *Proc. 18th IEEE International Symposium on Reliable Distributed Systems (SRDS'99)*, pages 254–262. IEEE Computer Society, 1999.
18. Hurfin M., Macêdo R., Raynal M., and Tronel F. A general framework to solve agreement problems. In *Proc. 18th IEEE International Symposium on Reliable Distributed Systems (SRDS'99)*, pages 55–65. IEEE Computer Society, 1999.
19. The Common Object Request Broker: Architecture and Specification. Technical Report 2.6, Object Management Group, December 2001.
20. Fault tolerant corba. corba 2.6. Technical Report 2.6, Object Management Group, December 2001.

21. Andre Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.
22. Douglas C. Schmidt. http://www.cs.wustl.edu/ schmidt/tao.html. Internet site, Dec 2002.
23. N. Sergent, X. Défago, and A. Schiper. Impact of a failure detection mechanism on the performance of consensus. In *Proc. IEEE Pacific Rim Symp. on Dependable Computing (PRDC)*, Seoul, Korea, December 2001.
24. P. Sommerlad and F. Buschmann. Manager design pattern. In *3rd annual PLoP*, Allenton Park, Illinois, September 1996.
25. Irineu Sotoma and Edmundo R. M. Madeira. ADAPTATION - Algorithms to ADAPTive FAulT MonItOriNg and Their Implementation on CORBA. In *Proc. IEEE of the 3rd International Symposium on Distributed Objects and Applications (DOA01)*, pages 219–228, Rome, Italy, September 2001.
26. Irineu Sotoma and Edmundo R. M. Madeira. DPCP(Discard Past Consider Present) - A Novel Approach to Adaptive Fault Detection in Distributed Systems. In *Proc. of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS2001)*, pages 76–82, Bologna, Italy, November 2001.
27. P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proc. of the 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, pages 582–589, October 2000.

Annex A – APIs for DisCusS and FuSe

A.1 Basic Definitions

```
//Basic definitions from FT-CORBA.
typedef unsigned short FaultMonitoringStyleValue;
const FaultMonitoringStyleValue PULL = 0;
const FaultMonitoringStyleValue PUSH = 1;
const FaultMonitoringStyleValue NOT_MONITORED = 2;
enum MonitorableStatus {SUSPECTED, ALIVE, NOT_MONITORED};

//Extensions for ADAPTATION algorithms.
const FaultMonitoringStyleValue PULL_ADAPTATION = 3;
const FaultMonitoringStyleValue PUSH_ADAPTATION = 4;

struct FaultMonitoringIntervalAndTimeoutValue {
  TimeBase::TimeT    monitoring_interval;
  TimeBase::TimeT    timeout;
};
```

A.2 Monitor Related Definitions

```
module FUSE{
  interface MonitorableStrategy;

  typedef Long MonitorUID;
  typedef Long FUSEObjectUID;
  typedef sequence<MonitorableStrategy> SuspectList;

  interface SuperMonitor {
    void start_monitoring_all ();
    void stop_monitoring_all ();
    void start_monitoring (in FUSEObjectUID uid);
    void stop_monitoring (in FUSEObjectUID uid);
```

```
    //To add/remove a Monitor <-> Monitorable association.
    FUSEObjectUID add_monitoring_assoc (in GeneralMonitorStrategy monitor,
                                        in MonitorableStrategy monitorable);
    Boolean remove_monitoring_assoc (in FUSEObjectUID uid);

    void change_suspect_list (in FUSEObjectUID uid,
                              in MonitorableStatus status);
    //To query about failures.
    Boolean is_it_suspect (in FUSEObjectUID uid);
    SuspectList get_suspect_list();
  };
  interface GeneralMonitorStrategy {
    //Used to manipulate the fault monitoring parameters.
    void set_monitoring_interval(in FaultMonitoringIntervalAndTimeoutValue mi);
    FaultMontoringIntervalAndTimeout get_monitoring_interval();

    void start();
    void stop();

    //To associate this monitor to a SuperMonitor.
    void set_super_monitor(in SuperMonitor sm);

    //Inform which monitorable this monitor must monitor
    //and what is its identifier on the SuperMonitor.
    void set_monitorable(in MonitorableStrategy ms,
                         in FUSEObjectUID mtb_uid);
    //Called by Monitorable. The message_id can
    //be used to identify out dated messages.
    oneway void i_am_alive(Long message_id);
  };
  interface GeneralMonitorFactory{
    GeneralMonitorStrategy create_general_monitor();
  };
}
```

A.3 Monitorable Object Related Definitions

```
module FUSE {
  interface MonitorableStrategy {
    boolean is_alive();

    oneway void set_monitoring_interval(in MonitorUID uid,
                                        in TimeBase::TimeT mi);
    TimeBase::TimeT get_monitoring_interval(in MonitorUID uid);

    oneway void run (in MonitorUID uid);
    oneway void stop(in MonitorUID uid);

    MonitorUID add_general_monitor(in GeneralMonitorStrategy gen_mon_stg);
    oneway void remove_general_monitor(in MonitorUID uid);
  };
  interface Notifier {
    void add_notifiable(in Notifiable ntb);
  };
  interface Notifiable {
    void notify(in SuspectList suspect_list);
  };
}
```

A.4 Consensus Definitions

```
module DISCUSS {
  typedef Long ConsensusID;
  typedef Long ConsensusMemberUID;
```

```
  struct Value {
    Long   ID;
    Any    data;
  };
  interface Translator {
    Value get_estimate(in ConsensusID cid);

    //To callback decisions.
    void decide (in ConsensusID cid, in Value decision);
  };
  interface Consensus {
    //For implicit invocation.
    void launch (in Translator translator);
    //For explicit invocation.
    void propose (in Value estimate, in ConsensusID cid);

    Value decide (in ConsensusID cid);
  };
  interface ConsensusMember {
    FUSEObjectUID add_super_monitor (in SuperMonitor sm);
    Boolean remove_super_monitor (in FUSEObjectUID smid);

    void add_consensus_member(in ConsensusMember member,
                              in ConsensusMemberUID cmuid);
    Boolean remove_consensus_member(in ConsensusMemberUID cmuid);
  };
}
```

A Lightweight Interface to Predict Communication Delays Using Time Series

Raul Ceretta Nunes[1,2] and Ingrid Jansch-Pôrto[1]

[1] Universidade Federal de Santa Maria, Depto Eletrônica e Computação
Av. Roraima S/N, Camobi, 97105-900, Santa Maria, RS, Brazil
ceretta@inf.ufsm.br
[2] Universidade Federal do Rio Grande do Sul, Instituto de Informática
Caixa Postal 15064, 91501-970, Porto Alegre, RS, Brazil
ingrid@inf.ufrgs.br

Abstract. Chandra and Toueg first investigated the problem of solving consensus in asynchronous systems with unreliable failure detectors. In practice, such failure detectors work by suspecting consensus participants that do not respond in sufficient time, i.e., within a *timeout* (a maximum waiting time). However, calculating this timeout is difficult. In this way, many systems explore prediction techniques to dynamically forecast the communication delay used in timeout computation. In this paper, we are concerned with the timeout associated with failure detection in asynchronous systems in which processes may crash and links may lose messages. We propose a lightweight interface to transform a non-periodic sampling of the round-trip communication delay (*rtt*) observed by a pull-style failure detector in periodic sampling. By using this interface, we explore the use of time series predictors to dynamically forecast the *rtt*. As a result, we show that these predictors are more accurate than others commonly used on failure detectors.

1 Introduction

Many of the distributed applications have requirements for fault tolerance; they are based on protocols that take into account component failures and need some form of distributed agreement. Unfortunately, agreement problems are not solvable with deterministic algorithms in an asynchronous distributed system if one single process may crash. This impossibility is caused by the difficulty of distinguishing a *very slow* process from a crashed one [1].

To circumvent this impossibility, Chandra and Toueg [2] encapsulated the problem in a named *unreliable failure detector*, a mechanism that can make mistakes. Informally, an unreliable failure detector is a service - usually related to each process - that provides (unreliable) information about component failures. Although the information provided by a failure detector may be unreliable, it is enough to allow deterministic solutions on a number of agreement problems [2]. However, as other time-based distributed protocols, the failure detector implementations are generally based on *timeouts* to ensure termination, making their accuracy dependent on the correct choice of the timeout. The choice of timeout

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 254–263, 2003.

values is crucial for the failure detector's ability to respect accuracy and completeness. Some timeouts allow a process to detect failures quickly but increase the number of false suspicions, with a risk of violating accuracy. Hence, there is a trade-off between latency (short timeouts) and accuracy (long timeouts).

To set the timeout, the system designer can choose among various off-line or on-line procedures. An off-line adjustment either follows a careful analysis of the execution environment [3] or is determined by experience [4]. On the other hand, an on-line adjustment may be dynamically predicted according to the observed system behavior by: *i*) using only the last sample value [5]; *ii*) using the average of the n last sample values [6]; or *iii*) using the exponential smoothing of the sample values [7]. The on-line approach depends on prediction techniques to dynamically forecast the communication delay used in timeout computation.

By investigating a self-tuned version of a pull style failure detector [8], we showed [9] that the non-periodic sampling of a round-trip communication delay (*rtt*) can be modeled as a time series [10] by using an interface between sampling and the time series representation. Time series have been successfully used in the econometrics area as a tool to forecast behaviors based on past samples and have emerged in the distributed system area to dynamically predict variables of interest [11,12]. Our previous result permits the use of time series prediction models to dynamically forecast the *rtt* for distributed algorithms.

In this paper, we define a new lightweight interface and use it to evaluate the accuracy[1] (mean square error - *mse*) of time series based predictors. To show that time series predictors are more accurate than others commonly used in failure detectors, we evaluated the *mse* of a large real data set (23 traces of 24 hour each) and compared them by using the analysis of variance[2] (ANOVA) and, where necessary, the Duncan multiple range test[3].

The remainder of this paper is organized as follows: in section 2, we specify an interface that transforms a non-periodic sample into a periodic sample. In section 3, we describe the properties of the seven evaluated predictors. In section 4, we study and compare the accuracy of the predictors using the same set of traces collected over a wide area network. Finally, in section 5, we present the main results of our analysis.

2 Time Series Interface

A time series Z_t is a sequence of observations taken sequentially in time [10] that can be used to forecast the stochastic variable, typically, under the assumption that adjacent observations are correlated. As we are interested in the round-trip communication delay (*rtt*) forecast, a time-variant variable, in this section we discuss how the *rtt* is associated with a discrete model (subsection 2.1) and we specify a lightweight interface to dynamically predict it.

[1] As we are interested in comparing the accuracy of predictors instead of the accuracy of the failure detectors, the metric *mse* was chosen.

[2] An appropriate procedure for testing the equivalence of several means.

[3] A test to evaluate the differences among multiple means.

2.1 Discrete Model of the Round-Trip Communication Delay

Consider a failure detector, running over a network like the Internet, where the monitor process p sends a *req* message to the monitored process q at each time interval t_i and wait for an *ack* message. As on the Internet the communication delays have a time-variant nature [13], *req* and *ack* transmission delays (d_{req} and d_{ack}) result in a time-variant *rtt* sequence in p, independently of the behavior of q. Thus, this failure detector is a system where the source is *time-invariant* but the *rtt* sequence is *time-variant*.

Now, consider a communication model where $u(t)$ represents the *req* input time-invariant sequence at instant t, $d(t)$ represents the variable *rtt* that begin at t ($rtt_t = d_{req_t} + d_{ack_t}$) and $v(t)$ represents the *rtt* time-variant sequence. Suppose a time window $\mathcal{T}$ and a function $V(\mathcal{T})$ that represents the set of *rtt* computed during the time window $\mathcal{T}$. Examining this set $V(\mathcal{T})$, sometimes it may contain multiple values while, at other moments, it may be empty. This result comes from the time-variant nature of $d(t)$ related to the discrete nature of $\mathcal{T}$.

From the point of view of the failure detector, $V(\mathcal{T})$ is the most important sequence, because it informs the *rtt* from the monitor process p to the monitored process q. Thus, if $\mathcal{T} = t_i$ and it begins at the req_t send time, in a failure-free execution, we expect $V(\mathcal{T}) = rtt_t$. With these settings, all req_t messages with $d(t) > \mathcal{T}$ will generate rtt_t measurements in a future time window $\mathcal{T} + m.\mathcal{T}$, where $m = \lceil d_t/\mathcal{T} \rceil$. Formally, we define $V(\mathcal{T})$ as:

$$V(\mathcal{T}) = \bigcup_{t \in U(\mathcal{T})} v(t) - u(t) \tag{1}$$

where $U(\mathcal{T})$ represents the set of time instants that produce outputs at time window $\mathcal{T}$ defined as:

$$U(\mathcal{T}) = \{t \mid t + d(t) \in \mathcal{T}, t \geq 0\} \tag{2}$$

Note that the above definitions are independent from the distributed system model (asynchronous or synchronous). If $d(t)$ is unbounded, we have an asynchronous system and $d(t)$ is defined as:

$$0 < d_{min} \leq d(t) \leq \infty \tag{3}$$

where d_{min} is unknown. We included the occurrence of $d(n) = \infty$ to consider lost messages (they will never arrive). Otherwise, if $d(t)$ is bounded, we have a synchronous system and $d(t)$ is defined as:

$$0 \leq d_{min} \leq d(t) \leq d_{max} < \infty \tag{4}$$

where d_{min} and d_{max} are known.

Equations 1-4 together represent our model of a time-variant discrete communication delay. Similarly to Bauer's [13] approach, the above model does not make any assumption about how the delays are interfaced with the failure detector predictor and it describes nothing more than the nature of the *rtt*.

2.2 The Interface

To present how the time-variant *rtt* sequence can be interfaced and represented as a time series, a model to represent time-invariant measures, we assume that the length of the time series lag is $\mathcal{T} = t_i$, where the period $\mathcal{T}$ starts at the time instant t. Also let $card(V(\mathcal{T}))$ be the cardinality of the set $V(\mathcal{T})$, defined by equation (1), and let $B(\mathcal{T})$ be the set of the out of order and duplicated messages collected and discarded during period $\mathcal{T}$. Note that $card(V(\mathcal{T}))$ represents the amount of *rtt* collected in the period $\mathcal{T}$. Thus, at each period $\mathcal{T}$, the monitor process p sends a req_t message and waits for the corresponding ack_t. For simplicity, we used $t_i = 1$ time unit.

If p receives only one ack_t in the lag $\mathcal{T}$, then there exists only one t such that $t \in U(\mathcal{T})$ ($U(\mathcal{T})$ is defined by the equation (2)) and $card(V(\mathcal{T})) = 1$. If $card(V(\mathcal{T})) = 1$ and $t \notin B(\mathcal{T})$, we have a stable period where only one valid message arrives (the expected ack_t or the high order *ack*) and we can put the rtt_t sample in the *rrt* time series. Thus,

$$\forall \mathcal{T} \text{ such that } card(V(\mathcal{T})) = 1 \text{ and } t \notin B(\mathcal{T}), \text{ let } z_t = rtt_t \tag{5}$$

In a situation where no *ack* message arrives during the lag $\mathcal{T}$, we get $U(\mathcal{T}) = V(\mathcal{T}) = \{\emptyset\}$ or, in other words, we get a *gap*. Consequently, when a gap occurs, $card(V(\mathcal{T})) = 0$ and there is no value to insert in the time series Z_t at period $\mathcal{T}$. As discussed in [9] and [14], we have different fill policies to deal with the gaps in a time series representation. Among them, an efficient policy to complete the time series is to ignore the gaps. Ryan and Giles [14] stated that this policy is the best choice for stationarity analysis, and we have used it [15] to analyze the stationarity of the *rtt* pattern. As it is the fastest method to complete the time series, we adopted it to lighten our interface. From the point of view of the failure detector, this is not a problem, because the predictor does not need to predict a gap. This task can be accomplished by adding t_i when computing the timeout or by using a specific algorithm, similar to the exponential speed-up from the Transmission Control Protocol. Thus, in this situation, instead of propagating the gap into the time series, our interface simply does nothing, i.e,

$$\forall \mathcal{T} \text{ such that } card(V(\mathcal{T})) = 0, \text{ do nothing} \tag{6}$$

In a scenario where only old or duplicated messages arrive, i.e, $U(\mathcal{T}) = B(\mathcal{T})$, all messages are discarded and we can apply (6).

After discarding all invalid messages, if only one message remains, we apply (5). But, if more then one message remains, $card(V(\mathcal{T})) > 1$ and $V(\mathcal{T}) = \bigcup_{t \in U(\mathcal{T})} rtt_t$. In the last situation, we can either insert in Z_t all the *rtt* measurements taken from valid messages or insert only one *rtt* measurement. Observing many non-periodic collected sequences, we noticed that some groups of adjacent messages are delayed and arrive all together later on (in a future interval), generating a burst of measurements, possibly as a consequence of network congestion. As the removal of these valid samples makes the time series less representative, our interface insert all valid samples in the time series. Thus,

$$\forall \mathcal{T} \text{ such that } card(V(\mathcal{T})) > 1, \text{ let } z_i = rtt_t \forall t \in U(\mathcal{T}) - B(\mathcal{T}) \tag{7}$$

where for each new t the index i is incremented by one.

The equations (5)-(7) formally define the behavior of the interface needed to transform the *rtt* time-variant sampling into a *rtt* time series, a time-invariant representation. The algorithm in Figure 1 implements this interface. In this algorithm, Π represents the set of processes, and $Z_{p,q}$ is the particular time series Z_t built by process p from the *rtt* that it notices when interacting with process q. The $timestamp_{ack_t}$ is the sequence number of the ack_t message, and $expectedTimestamp$ is the sequence number that p expects to receive from q. Figure 2 show the interface working.

$\forall q \in \Pi : Z_{p,q} \leftarrow \{\}$
$\forall q \in \Pi : expectedTimestamp \leftarrow 0$
$i = 0$
loop
  wait for a message ack_t from a process q
  if $timestamp_{ack_t} \geq expectedTimestamp$ **then**
    $Z_{p,q}[i] \leftarrow rtt_t$ computed from ack_t
    $expectedTimestamp \leftarrow timestamp_{ack_t} + 1$
    $i \leftarrow i + 1$
  end if
end loop

Fig. 1. Algorithm of the non periodic to periodic interface for each process p.

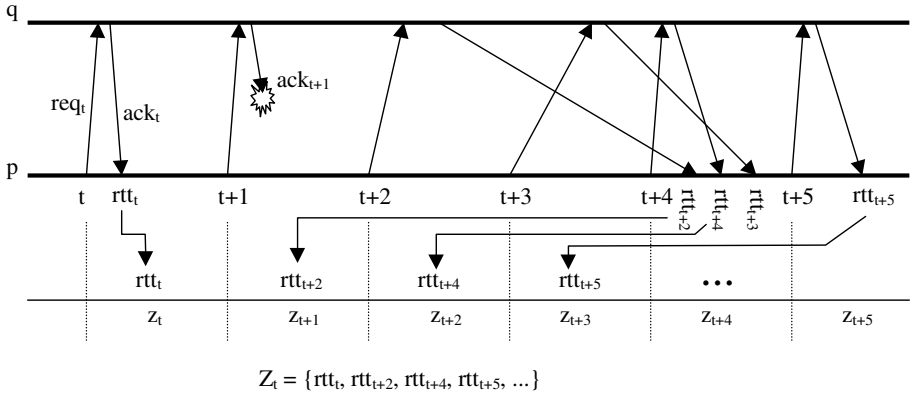


Fig. 2. Interface building a *rtt* time series.

3 Predictors

Under different assumptions, in this section we specify seven distinct predictors. The simplest predictor, named LAST, assumes that the next sample just follows

the last one. Under the assumption that a stochastic process follows a mean level, we specify three predictors: MEAN, that follows the population average; WINMEAN, a predictor that follows the average of the last $n = 30$ samples[4]; and LPF, that follows a low-pass filter. Under the assumption that sampling is a non-stationary time series[5], we specify two other predictors: DMA, that is based on a double moving average; and BROWN, that is based on a double exponentially weighted average. Finally, under the assumption that the time series could change between stationary and non-stationary, we specify the predictor ARIMA, which identifies an autoregressive integrated moving average model and makes predictions based on it. By assuming that *rtt* sampling is a time series, the predictors DMA, BROWN and ARIMA use the interface defined at section 2 to build a time series from time-variant *rtt* sampling.

As a function of the *rtt*, the predictor LAST estimates the next value by $\widehat{rtt}_{t+1} = rtt_t$, the predictor MEAN estimates it by $\widehat{rtt}_{t+1} = (\sum_{i=1}^{i=t} rtt_i)/t$, and the predictor WINMEAN estimates it by $\widehat{rtt}_{t+1} = (\sum_{i=t-n}^{i=t} rtt_i)/n$.

The predictor LPF implements a simple exponential smoothing by employing a low-pass filter technique [7] to predict future data values. Thus, $\widehat{rtt}_{t+1} = \alpha * rtt_t + (1-\alpha) * \widehat{rtt}_t$, where α is the linear prediction coefficient (set to 0.125 as suggested in [7]), and $\widehat{rtt}_t$ is the last smoothed average.

The predictor DMA (Double Moving Average) works by computing the moving average twice. It assumes that the non-stationary behavior can be described by a linear function as $rtt_t = \phi + \psi.t + a_t$, where ϕ and ψ are unknown parameters of the function and a_t is the white noise. Thus, from [16], by extrapolation, it estimates the *rtt* at instant $t+1$ by $\widehat{rtt}_{t+1} = 2.M_t - M_t^2 + \frac{2}{n-1}.(M_t - M_t^2)$, where M_t is the simple moving average and M_t^2 is the double moving average.

Like DMA, the predictor BROWN (Brown's method) assumes a linear trend model, but works as a double low-pass filter. Thus, by extrapolation $\widehat{rtt}_{t+1} = 2.rtt_t^1 - rtt_t^2 + \frac{\alpha}{1-\alpha}.(rtt_t^1 - rtt_t^2)$, where $\alpha = 0.125$, rtt_t^1 is the simple exponential smoothing, and rtt_t^2 is the double exponential smoothing, computed as $\widehat{rtt}_t^2 = \alpha.rtt_t^1 + (1-\alpha).rtt_{t-1}^2$.

The predictor ARIMA (*autoregressive integrated moving average*) allows the choice of the correct model among: a purely p-order autoregressive; a purely q-order moving average; an autoregressive and p- and q-order moving average ; and evidently an p-, d- and q-order autoregressive integrated moving average, where d represents the integration order (or non-stationary order) of the stochastic process. In our experiments, the forecast of predictor ARIMA is computed by the Resource Prediction System specified in [11] as $\widehat{rtt}_t = (\theta(B).a_t)/(\phi(B).\nabla^d)$, where B is the *backward shift operator* defined by $B.rtt_t = rtt_{t-1}$, ∇ is the *backward difference operator* defined by $\nabla rtt_t = rtt_t - rtt_{t-1} = (1-B)rtt_t$, a_t is the noise at period t, d is the non-stationary order, and $\phi(B)$ and $\theta(B)$ are

[4] By rule of thumb when $n \geq 30$ any probability distribution could be approximated to a Normal distribution [10].

[5] On previous work [15], we have analyzed many *RTT* time series collected from the Internet and have observed that most of them are non-stationary instead of stationary.

the B polynomials with p and q coefficients, respectively, where p is the number of autoregressive terms and q is the number of moving average terms. As this predictor carries through forecasts based on polynomial equations, to get a good forecast we periodically refit the polynomial structure and its parameters (p, d and q). In our experiments, after 120 new samples we enabled the model to refit based on the last 360 values of the *rtt* time series. The final decision to refit takes into account the square error of the prediction.

4 Predictors Evaluation

To evaluate the accuracy of predictors we use *mean square error* (*mse*) metric. Such metric considers both the *rtt* variance and its bias. The bias indicates the difference between the expectance of the estimator of *rtt* and the sampled value.

Evaluation methodology

To collect data sets, we used a pull-style monitoring program, that implements the interface defined in section 2. Running it over the Brazilian national research network during one week, we collected 23 traces, each one with 24-hour length (∼86400 samples from 00:00 a.m. to 12:00 p.m.).

To evaluate the predictors we followed three steps: 1 - from the 23 traces, we computed the *mse* of each predictor and we compared the results using the analysis of variance[6] (ANOVA). If the analysis of variance concluded that the averages were not all equivalent (one or more averages differ for more than 5% from), we used the Duncan multiple range test to study the differences; 2 - from the 17 working day traces taken between 6 a.m. and 12 p.m. (18-hour traces), we recomputed the *mse* and repeated the previous analysis; 3 - from the previous 17 traces, we have randomly generated 8160 traces with 300, 600, ... , 3300, 3600-length samples each (680 traces for each length) and repeated the analysis.

Accuracy analysis for 24-hour and 18-hour traces

Table 1 presents the results of the accuracy analysis for 24-hour and 18-hour traces, we can clearly see that the predictor MEAN offers the lowest accuracy with both data sets. Both *mse* values of predictor MEAN (78639 and 120196) are about 7 times greater than the WINMEAN *mse* values (10020 and 17217), the second worst predictor. In 24-hour and 18-hour experiments, the analysis of the variance indicates that at least one predictor differs from the others for more than 5% and Duncan's group shows us that this is the predictor MEAN (unique at group A). These results indicate that the predictor MEAN does not supply good predictions of the *rtt*, when considering long executions. From this table, we can also observe that statistically there is no difference among the other predictors, but that the two more accurate predictors are time series based (ARIMA and BROWN) and that smoothing-based predictors offer better accuracy than average-based predictors.

[6] An appropriate procedure for testing the equivalence of several means.

Table 1. Results for the 24-hour and 18-hour traces

24-hour traces				18-hour traces			
Group	*mse*	*n*	Model	Group	*mse*	*n*	Model
A	78639	23	MEAN	A	120196	17	MEAN
B	10021	23	WINMEAN	B	17217	17	WINMEAN
B	9240	23	DMA	B	15866	17	DMA
B	7981	23	LAST	B	13386	17	LAST
B	7569	23	LPF	B	12928	17	LPF
B	7280	23	BROWN	B	12389	17	BROWN
B	6407	23	ARIMA	B	10737	17	ARIMA

Accuracy analysis of different length traces

Analyzing the *mse* values and their variance, we observed that for all evaluated lengths at least one predictor differs from the others. The summary of Duncan's test is presented on Table 2. From this table, we can observe that all predictors based on arithmetic averages result in low accuracy for all length, especially the MEAN. It was also observed that the predictor ARIMA presents the better *mse* for all lengths. Although statistically we can not assert that the predictor ARIMA is the best predictor, we observed that: *i*) among evaluated predictors, the most accurate ones always include: LAST, LPF, BROWN and ARIMA; *ii*) predictors in the best group, when ordered in relation to accuracy, always appear in the following order: ARIMA, BROWN, LPF and LAST; *iii*) the accuracy order in the best group does not depend on the trace length.

5 Conclusion

In this paper, we proposed a new lightweight interface to transform a non-periodic sampling of the round-trip communication delay to a periodic sampling. This interface allows us to explore the time series based predictors to forecast the round-trip communication delay on unreliable pull-style failure detectors.

By evaluating the accuracy of seven distinct predictors using a statistical approach, we observed that: *i*) the predictors based on arithmetic average (MEAN, WINMEAN and DMA) do not offer competitive accuracy; *ii*) the predictors LAST, LPF, BROWN and ARIMA do not differ statistically, but among them, the time series predictors (ARIMA and BROWN) offer better results; *iii*) although the forecast provided by the predictor LAST is cheap and accurate, in practice it reflects the noises of the stochastic variables in its forecast and this characteristic can cause instability in its host system (it must be investigated in each case).

From these results, we have concluded that by using our lightweight interface, a system user can employ time series based predictors to improve prediction accuracy. Although statistically the predictors ARIMA and BROWN do not present significant accuracy increases on *rtt* forecast, this kind of predictor offers better accuracy and appears as a choice to predict non-periodic communication delay samplings.

Table 2. Summary of Duncan's test by execution length

	300 samples		600 samples		900 samples		1200 samples	
Predictor	group	mse	group	mse	group	mse	group	mse
MEAN	A	25443	A	28022	A	32206	A	34955
DMA	A	24333	A	25805	B	17910	B	15978
WINMEAN	B	12576	B	12724	C	12786	B e C	13150
LAST	B	11400	B	11426	C e D	11435	D e C	11766
LPF	B	9602	B	9565	C e D	9560	D	9812
BROWN	B	9230	B	9235	C e D	9241	D	9485
ARIMA	B	8595	B	8634	D	8582	D	8792
	1500 samples		1800 samples		2100 samples		2400 samples	
Predictor	group	mse	group	mse	group	mse	group	mse
MEAN	A	37030	A	38636	A	40712	A	43341
DMA	B	15208	B	14667	B	14566	B	15060
WINMEAN	B e C	13426	B	13481	B e C	13858	B e C	14707
LAST	B, C e D	12383	B, C e D	12846	B, C e D	12902	B, C e D	13318
LPF	C e D	10136	C e D	10301	C e D	10487	B, C e D	11088
BROWN	C e D	9852	C e D	10069	D	10204	C e D	10749
ARIMA	D	9135	D	9389	D	9512	D	9910
	2700 samples		3000 samples		3300 samples		3600 samples	
Predictor	group	mse	group	mse	group	mse	group	mse
MEAN	A	46722	A	48162	A	49776	A	52019
DMA	B	15940	B	15992	B	16232	B	16577
WINMEAN	B	15910	B	16125	B	16515	B	17002
LAST	B e C	13682	B e C	13908	B e C	14227	B e C	14487
LPF	B e C	11831	B e C	12028	B e C	12338	B e C	12676
BROWN	B e C	11380	B e C	11578	B e C	11877	B e C	12186
ARIMA	C	10294	C	10415	C	10644	C	10815

References

1. Fischer, M., Lynch, N., Paterson, M.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382
2. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
3. Raynal, M., Tronel, F.: Group membership failure detecton: a simple protocol and its probabilistic analysis. Distributed Systems Engineering Journal **6** (1999) 95–102
4. van Renesse, R., Minsky, Y., Hayden, M.: A gossip-style failure detection service. In: Proc. of the Middleware Conference. (1998) available at Cornell Digital Library (TR98-1687).
5. Macêdo, R.: Implementing failure detection through the use of a self-tuned time connectivity indicator. Technical Report RT008/98, Laboratório de Sistemas Distribuídos – LaSiD, Salvador-Brazil (1998)
6. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE Transaction on Computers **51** (2002) 561–580

7. Jacobson, V.: Congestion avoidance and control. In: Proc. of the ACM Symposium on Communications, Architectures and Protocols (ACM SIGCOMM), Palo Alto, ACM Press (1988) 314–329 Slightly-revised 1992 version of the 1988 paper.
8. Felber, P., Défago, X., Guerraoui, R., Oser, P.: Failure detectors as first class objects. In: Proc. of the Intl. Symp. on Distributed Objects and Applications – DOA'99, IEEE Computer Society Press (1999)
9. Nunes, R.C., Jansch-Porto, I.: Modelling communication delays in distributed systems using time series. In: Proceedings of the 21th IEEE Symposium on Reliable Distributed Systems (SRDS'2002), Osaka-Japan (2002) 268–273
10. Bowerman, B.L., O'Connel, R.T.: Forecasting and Time Series: an Applied Approach. Third edn. Duxbury Press, Belmont, CA (1993)
11. Dinda, P.A., O'Hallaron, D.R.: An extensible toolkit for resource prediction in distributed systems. Technical Report MU-CS-99-13, Computer Science Department – Carnegie Mellon University, Pittsburgh (1999)
12. Schulz, J., Hochberger, C., Tavangarian, D.: Prediction of communication performance for wide area computing systems. In: Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing, Mantova-Italy, IEEE Computer Society Press (2001) 480–486
13. Bauer, P., Sichitiu, M., Premaratne, K.: On the nature of the time-variant communication delays. In: Proceedings of the IASTED Conference Modeling, Identifi cation and Control (MIC'2001), Innsbruck, Austria (2001) 792–797
14. Ryan, K.F., Giles, D.E.A.: Testing for unit roots with missing observations. Advances in Econometrics (1998) Forthcoming in T.B. Fomby & R. Carter Hill (eds.).
15. Nunes, R.C., Jansch-Porto, I.: Non-stationary communication delays in failure detectors. In: Proceedings of the 3rd IEEE Latin-American test Workshop (LATW'02), Montevideo-Uruguay (2002) 16–21
16. Montgomery, D.C., Johnson, L.A.: Forecasting and time series analysis. McGraw-Hill, New York (1976)

A New Diagnosis Algorithm for Regular Interconnected Structures

Antonio Caruso, Luiz Albini, and Piero Maestrini

[1] Istituto di Scienza e Tecnologie dell'Informazione, Area della Ricerca, CNR di Pisa-S.Cataldo, 56100 Pisa, Italy.
[2] Computer Science Department, University of Pisa, Via Buonarroti 2, 56127 Pisa, Italy.

{caruso,albini,maestrini}@isti.cnr.it

Abstract. We present a new diagnosis algorithm (NDA) for regular interconnected structures. The diagnosis algorithm has time complexity $O(kn)$ when applied to k-regular systems of n units. It provides a correct diagnosis, although incomplete. The diagnosis is correct if the number of faulty units is not above a specified bound T_σ, asserted by the algorithm itself. The correctness and completeness of NDA is studied through simulations on toroidal grids and hypercubes. Simulation results show that NDA provides a correct diagnosis even when the number of faults is very high (near half of the system size). The comparison between algorithm NDA and other diagnostic algorithms shows that NDA provides a better diagnosis, i.e., it has a higher degree of completeness than other diagnostic algorithms.

1 Introduction

System-level diagnosis, which was introduced by Preparata, Metze and Chien [16], aims at diagnosing systems composed by units (usually processors) connected by point-to-point, bidirectional links. Diagnosis is based on diagnostic tests between adjacent units. Each test provides a binary outcome (0 means test-passed, 1 means test-failed).

The PMC model, introduced in [16], assumes that tests performed by fault-free units are completely reliable (that is, the test has perfect coverage), while tests performed by faulty units are completely unreliable.

The set of all test outcomes is called *syndrome*, denoted σ. Given a syndrome σ, the task of determining the status of the units is known as *syndrome-decoding*. A diagnosis algorithm performs the syndrome-decoding task and classifies units as faulty, fault-free, or unknown. The diagnosis is said to be correct if the diagnosed status of each unit corresponds to its actual status, it is said complete if all units are classified as fault-free or faulty.

System–level diagnosis may be exploited to identify faulty units and, in combination with replacement and recovery techniques, may be used to ensure dependability of massively parallel systems at a relatively low cost, thus providing

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 264–281, 2003.

an alternative to standard techniques of error codes or hardware redundancy [1, 5,12].

A promising application of system–level diagnosis is the wafer–scale manufacturing process of VLSI chips [14,10]. The test is based on a self–diagnosis of chips to be executed on the wafer before the chips are cut, bounded and packaged. The goal of diagnosis is to identify good ICs, which will undergo the following steps of manufacturing, while the faulty circuits will be discarded, thus saving the costly process of packaging.

This paper introduces a new diagnosis algorithm (NDA) for a wide class of regular systems. Section 2 presents the diagnostic model in detail. Section 3 reviews related works on diagnosis algorithms. Section 4 presents the diagnosis algorithm with some notes on its degree of correctness and completeness. Section 5 reports the outcomes of simulation experiments and the comparison between NDA and other diagnosis algorithms. We finally draw our conclusions in Sect. 6.

2 System Model

A system S is represented as an undirected *system graph* $G = (\mathsf{V}, \mathsf{L})$, where nodes represent units and edges represent interconnection links. There exists edge[1] $\{u, v\} \in \mathsf{L}$ if and only if units u and v are interconnected. The cardinality $n = |\mathsf{V}|$ is called the *size* of the system. Units u and v are said to be *adjacent*, denoted $u \longleftrightarrow v$, if $\{u, v\} \in \mathsf{L}$.

In the *PMC model* [16] diagnosis is based on a suitable set of tests between adjacent units. The *testing* unit u_i provides a test sequence to the *tested* unit u_j, which returns an output sequence to u_i. The testing unit compares the actual and the expected output sequences and generates a test outcome $a_{ij} = 1$ if it finds that the test failed; otherwise $a_{ij} = 0$. A test of v performed by unit u with outcome $\delta \in \{0, 1\}$ is denoted by $u \xrightarrow{\delta} v$. The concise notation $u \overset{\gamma}{\longleftrightarrow}{}^{\delta} v$, with $\gamma, \delta \in \{0, 1\}$, denotes both the test of u performed by v with outcome γ and the test of v performed by u with outcome δ. The PMC model assumes that tests performed by fault-free units have perfect coverage, while tests performed by faulty units have arbitrary results. This *invalidation rule* is shown in Table 1.

Table 1. Invalidation rule of the PMC model

Testing unit	Tested unit	Test outcome
Faulty-Free	Faulty-Free	0
Faulty-Free	Faulty	1
Faulty	Faulty-Free	0 or 1
Faulty	Faulty	0 or 1

The *diagnostic graph* $DG = (\mathsf{V}, \mathsf{E})$ represents the part of the system under test. Set V is composed of units involved in the diagnostic test, and edges (u, v)

[1] We denote (u, v) a directed edge from unit u to v, and $\{u, v\}$ an undirected edge.

from u to v exists if and only if unit u tests unit v. Edges in E are labeled with the binary test outcomes.

Given any set $\mathsf{V}_f \subseteq \mathsf{V}$ of faulty units (*actual fault set*), the set of all test outcomes is called *syndrome*, denoted σ. Given a syndrome σ, the task of determining the status of the units is known as *syndrome-decoding*. In the *centralized diagnosis* approach [16], the syndrome is decoded by an external, reliable computer, called *diagnoser*.

Preparata et al. introduced the concepts of *one-step* and *sequential diagnosis* [16]. In the former approach it is required that a single syndrome decoding identifies the status of all units. The latter approach consists of several *diagnosis and repair* phases, the goal of each phase is the identification of at least one faulty unit. Once identified, faulty units are immediately repaired or replaced, thus reducing the number of faulty units. The process is iterated until all the faulty units have been removed.

A system S is *one-step diagnosable*, that is, one-step diagnosis of S is always possible, if the number of faults in the system is not above a parameter, the *one-step diagnosability*, related to the structure of the diagnostic graph. A similar parameter for sequential diagnosis is called *sequential diagnosability* and it is usually far above the one-step diagnosability.

The one-step diagnosability of a system is limited above by the minimum number of the tests undergone by units in the system, that is, by the minimum of the node in-degrees in DG [9,16]. For this reason, the one-step diagnosis approach is inadequate to the case of large systems based on regular or quasi-regular interconnection structures, such as hypercubes, tori and grids.

Given a diagnostic graph $DG = (\mathsf{V}, \mathsf{E})$, and a syndrome σ resulting from an actual fault set V_f, we consider diagnosis algorithms which result is a partition of set V into subset F of units declared faulty, subset K of units declared non-faulty, and subset S of *suspect* units, i.e., the units whose status remains unidentified. The diagnosis is said to be correct if $\mathsf{F} \subseteq \mathsf{V}_f$ and $\mathsf{K} \subseteq \mathsf{V} - \mathsf{V}_f$. The diagnosis is said to be complete if $\mathsf{S} = \emptyset$. Using this viewpoint, one-step diagnosis is both *correct* and *complete*, while sequential diagnosis may leave unidentified, at every phase, the status of as many as $|\mathsf{V}| - 1$ units.

3 Related Work

One-step diagnosable systems have been characterized in [9], while sequentially diagnosable systems have been characterized in [11] and [15]. The problem of determining the one-step diagnosability of a given system was solved in [21], while the problem of determining the sequential diagnosability was shown to be co-NP complete in [17].

Lower bounds to sequential diagnosability of grids and hypercubes have been provided in [13]. However, the repeated execution of diagnosis and repair phases entailed by the sequential diagnosis approach may be time consuming and the underlying hypothesis, which excludes the occurrence of additional faults while the diagnosis is going on, may be unrealistic.

An alternative definition of systems *diagnosability* has been introduced in [4] as a possibility to achieve correct diagnosis of a large fraction of the system units. This definition of diagnosability is lower bounded [4] by a function $T(n)$ based on the edge isoperimetric inequality of the diagnostic graph.

Extensive research efforts have provided practical algorithms for the diagnosis of a system. Dahbura and Masson [8] suggested an $O(|\mathsf{E}|^{2.5})$ diagnosis algorithm for one-step diagnosis based on the concept of *disagreement set*. Sullivan [22] suggested an $O(t^3 + |\mathsf{E}|)$ fault diagnosis algorithm that uses a branch and bound procedure to build the diagnosis.

Diagnosis algorithms, which are able to diagnose in one step the state of a large fraction of the system units, thus providing an almost complete diagnosis, have been reported in the literature [3,6,7,5,18]. The almost complete diagnosis provided by the preceding algorithms is guaranteed to be correct if the number of actual faults is not above an *algorithm-specific diagnosability*.

The following section introduces a new diagnosis algorithm (NDA) inspired by the results obtained in the analysis of the diagnosability of regular systems, presented in [4]. The following two simple lemmas, from [4], are used in the definition of NDA.

Lemma 1. *From the PMC invalidation rule, for any two adjacent units u, v in set V the following statements hold:*

(a) If $u \overset{1\ 0}{\longleftrightarrow} v$, then u is faulty.
(b) If $u \overset{0}{\longrightarrow} v$ and u is fault-free, then v is also fault-free.
(c) If $v \overset{0}{\longrightarrow} u$ and u is faulty, then v is also faulty.
(d) If $u \overset{1\ 1}{\longleftrightarrow} v$, then at least one unit between u and v is faulty.

Proof. Directly from the invalidation rule of the PMC model. □

Lemma 2. *Given syndrome σ, let $\mathsf{A}_1, \mathsf{A}_2, \ldots, \mathsf{A}_r$ be the strongly connected components of $DG_0 = (\mathsf{V}, \mathsf{E}_0)$, where $\mathsf{E}_0 = \{(u, v) \in \mathsf{E} \mid u \overset{0}{\longleftrightarrow} v\}$. For every i, $1 \leq i \leq r$:*

(A) All units in A_i are in the same (faulty or non-faulty) state.
(B) If there exist units $u, v \in \mathsf{A}_i$ with $u \overset{1}{\longrightarrow} v$, then all units in A_i are faulty.
(C) If there exist units $u \in \mathsf{A}_i$, $v \in \mathsf{A}_j$ with $i \neq j$ and $u \overset{0}{\longrightarrow} v$, then all units in A_i are faulty.

Proof. Property (A) is immediate from properties (b) and (c) of Lemma 1. Property (B) is immediate from the combination of properties (a) and (c) of Lemma 1. Property (C) is immediate from properties (a) and (c) of Lemma 1, observing that must be $u \overset{1\ 0}{\longleftrightarrow} v$. □

4 A New Diagnosis Algorithm (NDA)

We use all possible interconnections to perform diagnostic tests between units. Thus, given a system graph G, the diagnostic graph $DG = (\mathsf{V}, \mathsf{E})$ is defined by replacing every undirected edge in G with a pair of directed edges with both orientations. A pseudo-code of the algorithm is reported in Algorithm 1. The algorithm takes as input the diagnostic graph DG and a syndrome σ and performs the diagnosis through two steps.

The first step performs the identification of the strongly connected components of subgraph $DG_0 = (\mathsf{V}, \mathsf{E}_0)$ of graph DG, where $\mathsf{E}_0 = \{(u, v) \in \mathsf{E} \mid u \overset{0}{\longleftrightarrow} v\}$. Intuitivelly, we partition DG in connected subsets of units that declare each other fault-free; all units into a component have the same state (faulty or fault-free) (Lemma 2.A).

Then, we build a set F as the union of those connected components A_i which satisfy condition (B) or (C) of Lemma 2; units in set F are diagnosed as faulty. The collection of remaining connected components is a partition of the subgraph $G - \mathsf{F}$ into components called Z-aggregates[2]. Two distinct Z-aggregates A_i, A_j are *adjacent* if at least a unit in A_i tests a unit in A_j.

Consider two units $u_i \in \mathsf{A}_i$ and $u_j \in \mathsf{A}_j$ with $i \neq j$, the diagnostic test between u_i and u_j must have outcome 1, thus by Lemma 1.d at least one unit, between u_i, u_j, must be faulty. This implies that if a Z-aggregate A_i is composed of fault-free units then all Z-aggregates adjacent to it must be composed of faulty units, that is, each set of fault-free Z-aggregates is an *independent set*[3].

The second step of NDA calculates two indexes, $g_0(i)$ and $g_1(i)$, for each Z-Aggregate A_i. These indexes are an approximation of indexes $G_0^\sigma(\mathsf{A}_i)$, $G_1^\sigma(\mathsf{A}_i)$ defined in [4] as the minimum number of faults implied by syndrome σ in the hypothesis that A_i is respectively fault-free ($G_0^\sigma(\mathsf{A}_i)$) or faulty ($G_1^\sigma(\mathsf{A}_i)$).

Thus, index $g_0(i)$ of Z-aggregate A_i is a lower bound to the number of faults that must be in the system if we assume A_i composed of fault-free units. In this case, from the property of independence of fault-free Z-aggregates, all Z-aggregates adjacent to A_i must be composed of faulty units. Thus, we define $g_0(i)$ as the sum of their cardinality. Similarly, index $g_1(i)$ is a lower bound to the number of faults that must be in the system if we assume A_i composed of faulty units. Since in this case there are at least $|\mathsf{A}_i|$ faulty units, we define $g_1(i) = |\mathsf{A}_i|$.

Based on these two indexes, the algorithm builds two sets: the *Fault Free Core* (FFC) composed of Z-aggregates declared fault-free; and a set of faulty Z-aggregates that are joined to set F. Units in the remaining Z-aggregates cannot be diagnosed, their union is the set S of *suspect* units.

If we take $\alpha = \max_i (g_1(i)) = \max_i |\mathsf{A}_i|$, then set FFC is composed by Z-aggregates A_i with $g_1(i) > \alpha - 1$, while Z-aggregates A_i with $g_0(i) > \alpha - 1$ are declared faulty and joined to set F. Note that, by our choice of α the inequality

[2] The Z means *Zero* and derives from the following property: all units in the same Z-aggregate test each other with outcome 0.

[3] An *independent set* or *stable set* is a set of pairwise nonadjacent vertices.

Algorithm 1 The *Diagnosis Algorithm* NDA

Input: A diagnostic graph DG and a syndrome σ over DG

{ **Initialization: all sets are empty**}

1 $\mathsf{F} = \emptyset$; $\mathsf{FFC} = \emptyset$; $\mathsf{S} = \emptyset$;

{ **Step 1: Aggregate Construction and F identification** }

{ Finding Aggregates in subgraph DG' }

2 **Let** DG' be the subgraph of DG induced by edges labeled with 0.

3 **Let** $\mathsf{A}_1, \ldots, \mathsf{A}_r$ be the strongly connected components of DG'.

{ Constructing set F of unconditionally faulty units }

4 **For each** $u \in \mathsf{A}_i$ **if** $\exists\, v \in \mathsf{A}_i$ **such that** $u \xrightarrow{1} v$ **Do** $\mathsf{F} = \mathsf{F} \cup \mathsf{A}_i$;

6 **For each** $u \in \mathsf{A}_i$ **if** $\exists\, v \notin \mathsf{A}_i$ **such that** $v \xrightarrow{0} u$ **Do** $\mathsf{F} = \mathsf{F} \cup \mathsf{A}_i$;

7 $f = |\mathsf{F}|$;

{ **Step 2: Approximation of g_0 and g_1 and diagnosis** }

Let $\mathsf{A}_1, \ldots, \mathsf{A}_h \in \mathsf{V} - \mathsf{F}$ be Z-aggregates.

8 **For** $i = 1, \ldots, h$ **Do** $g_0(i) = 0$; $g_1(i) = |\mathsf{A}_i|$;

For $i = 1, \ldots, h$ **Do**

For $j = 1, \ldots, h,\ j \neq i$ **Do**

9 **if** $\mathsf{A}_j \longleftrightarrow \mathsf{A}_i$ **then** $g_0(i) = g_0(i) + |\mathsf{A}_j|$;

{Derive the syndrome-dependent diagnosability bound for Z-aggregates}

10 $\alpha = \max\{\, g_1(1), \ldots, g_1(h) \,\}$;

{Diagnose as fault-free Z-aggregates with $g_1 > \alpha - 1$}

11 $\mathsf{FFC} = \bigcup_{i=1}^{h} \{\, \mathsf{A}_i \mid g_1(i) > \alpha - 1 \,\}$;

{Diagnose as faulty Z-aggregates with $g_0 > \alpha - 1$}

For $i = 1, \ldots, h$ **Do**

12 **if** $\mathsf{A}_i \notin \mathsf{FFC}$ **and** $g_0(i) > \alpha - 1$ **then** $\mathsf{F} = \mathsf{F} \cup \mathsf{A}_i$;

13 $\mathsf{S} = \mathsf{V}\text{-}\mathsf{F}\text{-}\mathsf{FFC}$;

14 **Output:** $< \mathsf{F}, \mathsf{FFC}, \mathsf{S}, T_\sigma = \alpha + f - 1 >$

$g_1(i) > \alpha - 1$ is equivalent to $|\mathsf{A}_i| = \alpha$, i.e., we select Z-aggregates of maximum cardinality. We employ $g_1(i) > \alpha - 1$ since it is symmetric to $g_0(i) > \alpha - 1$ and because it is more general, i.e., it is still valid if we change the definition of α.

The algorithm outputs a syndrome-dependent bound $T_\sigma = f + \alpha - 1$, where f is the cardinality of F as defined in the first step. It will be proved that, if

the number of faults in the system is not above T_σ, the diagnosis of NDA is correct. Bound T_σ is an approximation from below of the *syndrome-dependent diagnosability* t_σ defined in [4].

For the class of regular systems with edge-isoperimetric inequality we have proved in [4] a syndrome-independent worst-case bound $T(n)$ for t_σ. Thus, for this class of systems, the algorithm outputs a correct diagnosis for any possible syndrome σ, provided that the number of faults in the system does not exceed the value $T(n)$. Since the value of $T(n)$ is the result of a worst case analysis, we generally have $T(n) << T_\sigma$.

4.1 Diagnosis Correctness

Given a syndrome σ, the algorithm returns as output the quadruple (FFC, F, S, T_σ) (line 14). The correctness of the diagnosis relies on the following theorem:

Theorem 1. *Sets* FFC *and* F *are respectively fault-free and faulty provided that the number of actual faulty units t does not exceed the bound T_σ.*

Proof.

Suppose by contradiction that $t \leq T_\sigma$ and the diagnosis returned as output is not correct. This implies that some units in set FFC are not fault-free or some units in set F are not faulty.

We firstly note that units belonging to components added to set F in step one (lines 4–6) are correctly diagnosed as faulty by Lemma 2(B)(C). Therefore, after line 7 we know that there are $f = |\mathsf{F}|$ correctly diagnosed faulty units in the system. By the hypothesis $t \leq T_\sigma = \alpha + f - 1$ there must be at most $\alpha - 1$ faulty units in the remaining Z-aggregates.

If some fault-free unit u belong to set F, it must belongs to one of the Z-aggregates, say A_i joined to set F in step two (line 12). But A_i has $g_0(i) > \alpha - 1$, and the number of faulty units that must be present in the system is at least α, that is a contradiction.

Now suppose that set FFC contains some faulty units. If there is a faulty unit it must belong to one Z-aggregates, say A_k. By the properties of Z-aggregates this implies that all units in A_k are faulty. Since set FFC is constructed (line 11) as the union of Z-aggregates A_i that satisfy $g_1(i) > \alpha - 1$, we have $g_1(k) > \alpha - 1$, thus the number of faulty units is at least α, and this contradict the hypothesis.

Thus, if the number of faults does not exceed T_σ the diagnosis of NDA is correct. □

4.2 Time Complexity of NDA

The time complexity of the diagnosis algorithm is evaluated in the following theorem:

Theorem 2. *The time complexity of the diagnosis algorithm* NDA *is $O(kn)$ where k is the degree of the system graph.*

Proof.

The algorithm uses the following data structure for representing sets to perform several operations: a bit–vector of size n is used, where bit i is set to 1 if and only if unit u_i belongs to the set. With this data structure, testing if an element belongs to a set and adding or deleting an element to a set can be done in constant time. Calculating the cardinality of a set can be done in constant time by taking a counter for each set. Finally, the union of two sets requires linear time.

Consider step one: finding the connected components of the diagnostic graph DG_0 (lines 2–3) is a classical graph problem of complexity $O(n + |\mathsf{E}|)$, since the diagnostic graph is k-regular, the cardinality $|\mathsf{E}|$ is $O(kn)$ and the resulting complexity is $O(kn)$. The identification of set F (lines 4–7) can be done in time $O(kn)$.

Consider step two: it calculates indexes g_0, g_1 for each Z-aggregate. Index $g_0(i)$ is equal to the cardinality of A_i (line 8) thus can be calculated in constant time. Index $g_1(i)$ is the sum of the cardinality of Z-aggregates adjacent to A_i (line 9), and it can be calculated in time $O(kn)$ by checking at most all edges in the system. The maximum evaluated in line 10 can be done in $O(n)$ since we have $h \in O(n)$. The construction of set FFC can be done with two iterations. We identify in the first iteration which Z-aggregate belongs to set FFC in time $O(h)$, then we insert units belonging to those Z-aggregates in set FFC. Since $h \in O(n)$, set FFC can be constructed in time $O(2n)$. The Z-aggregates diagnosed faulty in line 12 require time $h \in O(n)$. The bound T_σ requires only a constant time operation. Since the dominating phases of the algorithm require time $O(kn)$, this is the resulting time complexity. □

4.3 Diagnosis Completeness

The proposed diagnosis algorithm provides a correct diagnosis, if the actual number of faulty units is not above T_σ. However, the algorithm is generally unable to achieve a complete diagnosis. The algorithm outputs an incomplete diagnosis if $\mathsf{S} \neq 0$. This is an expected result, since the value of T_σ is much higher than the one–step diagnosability t, because t in regular systems is limited from above by the degree of the diagnostic graph. However, simulative analysis shows that in the medium case the diagnosis is almost complete, and only a small fraction of the units remain undiagnosed.

5 Simulation Experiments

5.1 Simulation Model

A simulator program has been developed in order to evaluate several characteristics of NDA. The simulator takes as input a description of the diagnostic graph. Square grids (simple or toroidal) are described by the *side length* l and *node degree* d. The hypercube is described by its *dimension* d that is equal to

the degree of each node. Another parameter is the number of actual faults f, and the type of distribution to be used to distribute faults over the node set. Two distribution are available:

Uniform distribution: Each unit may fail with the same probability f/n.

Multivariate distribution: Given a parameter $h \leq 0.2f$, h *spot* faults are distributed uniformly over the node set; for every spot unit u_{xy} two random variance var_x, var_y are assigned, and a random number of faults are distributed in the rectangle $(x \pm var_x, y \pm var_y)$ using a normal bivariate distribution $N(0,1)$. This is repeated for each spot unit, with the constraint that a total of f faults are distributed. Multivariate distribution attempts to capture a frequently observed behavior of faults in VLSI wafers, i.e., the inclination of faulty chips to be clustered in some parts of the wafer.

After fault injection, a syndrome consistent with the invalidation rule of the PMC model (Table 1) is generated. The outcomes of tests performed by faulty units are defined randomly as follows: when a faulty unit tests a non-faulty unit the test outcome is 1 with probability p; when it tests a faulty unit the test outcome is 1 with probability q. The two probability p and q are input parameters of the simulator.

After the syndrome generation, the diagnosis algorithm is executed and the results are collected. A number of average measures are evaluated over samples of size at least 250; for each average the simulator reports the confidence interval[4] and the standard deviation. The confidence interval is calculated with precision 0.98 in all the simulation experiments. Every simulation experiment outputs the average $E[T_\sigma]$ of the syndrome dependent bound T_σ, the percentage of executions yielding incomplete diagnosis ($\%inc$) over the total number of executions, and the average number of suspect units $E[|\mathsf{S}|]$, averaged on those executions which yielded incomplete diagnosis. Since incomplete diagnoses may be a small fraction of all diagnoses (this situation actually occurs if the number of injected faults is below $10\%n$), the simulator runs until the target of 250 incomplete diagnosis is reached. However, a limit of 5000 executions was set for every simulation experiment, this means that the size of the sample may sometimes be less than 250.

5.2 Simulation Objectives

The simulation experiments have two main objectives: First, to evaluate NDA performance in terms of correctness and completeness of the diagnosis for simple grids, toroidal grids, and hypercubes. Second, to compare NDA with state of the art diagnosis algorithms for regular structures [7,18,13,20,14].

[4] Given a random variable X, the confidence interval calculated with precision r, is defined as the number i such that the probability $P(|E[X] - \mu[x]| \leq i)$ is greater than or equal to r, where $\mu[X]$ is the average of X calculated over the universe of all possible fault sets of given cardinality.

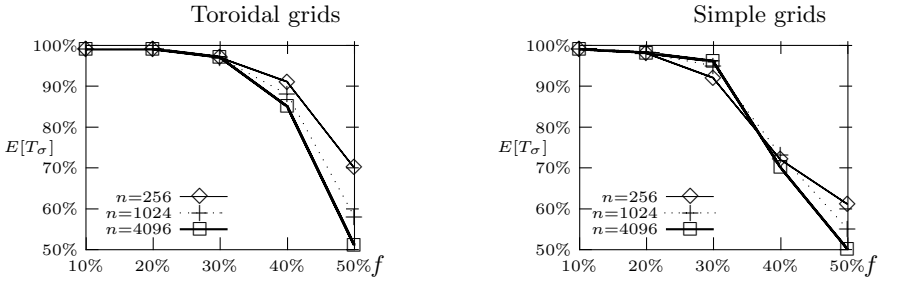


Fig. 1. $E[T_\sigma]$ as percentage of n for toroidal and simple grids. Fault sets with uniform distribution.

The rationale of evaluating the diagnosis correctness is the following: if the number of actual faults f is at most $T(n)$, the syndrome-independent bound proved in [4], then the diagnosis is known to be correct whatever is the resulting syndrome. If the number of actual faults is above $T(n)$ and the resulting syndrome is σ, the diagnosis is correct if $f \leq T_\sigma$ where T_σ is the syndrome dependent bound asserted by NDA. Since T_σ is determined under worst case conditions, the diagnosis may still be correct if $f > T_\sigma$. The main goal of simulative analysis of diagnosis correctness is the evaluation of the average value of T_σ over the set of syndromes arising from fault sets of given cardinality f, and the evaluation of the probability of incomplete diagnosis. To accomplish this analysis, simulation experiments were runs for f ranging from $0.1n$ to $0.5n$ in steps of $0.1n$.

The interest in studying the behavior of NDA with relatively large fault sets (up to $0.5n$), resides in the foreseen application to wafer-scale test of integrated circuits. This is the most promising application of system-level diagnosis, and represents a challenge to diagnosis algorithm, since the number of faults may be as large as $0.5n$.

The interest in simulative evaluation under multivariate distribution also derives from wafer scale testing applications. In fact, this model captures the clustering of faults which naturally occurs in the wafers.

5.3 Analysis of T_σ in the Average

Simulation experiments were carried out with simple and toroidal grids, and with hypercubes of size n, ranging from 256 to 4096. Faulty units are distributed over the node set with uniform and multivariate distributions. The parameters p and q, that affect the test outcomes, were initially set to 0.5, i.e., each faulty unit correctly perform the diagnostic test half of the time (in average).

The analysis of the average $E[T_\sigma]$, of the syndrome-dependent bound T_σ, gives a measure of the correctness of algorithm NDA. Figure 1 plots $E[T_\sigma]$ for simple and toroidal grids with uniform distribution of faults, and Fig. 2 plots $E[T_\sigma]$ for simple and toroidal grids with multivariate distribution of faults.

As evidenced from the plots, $E[T_\sigma]$ is generally greater than the actual number of faults; this implies a high percentage of correct diagnoses. This is particularly true in the case of multivariate distribution, where the values of T_σ were

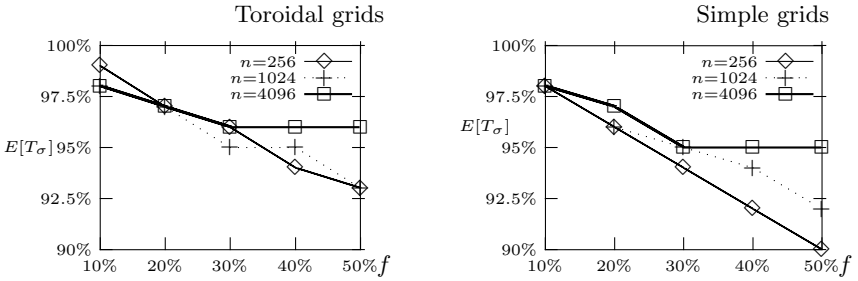


Fig. 2. $E[T_\sigma]$ as percentage of n for toroidal and simple grids. Fault sets with multivariate distribution.

always above $0.9n$. In the case of grids of size $n = 4096$ and fault sets of cardinality near $0.5n$ with uniform distribution, the average bound T_σ is slightly above $0.5n$: this implies that, although correct with high probability, there may be cases in which T_σ is smaller than the cardinality of the actual fault set, meaning that the correctness of the diagnosis algorithm cannot be guaranteed. However, comparisons between diagnoses and actual fault sets showed that the diagnosis was correct in all cases.

The value of $E[T_\sigma]$ for toroidal grids is always above the corresponding value obtained for simple grids, with both uniform and multivariate fault distributions. This is not surprising as the connectivity of toroidal grids is greater than the connectivity of simple grids.

The two distributions of faults produce different behaviors with respect to system size. With uniform distribution, faults are distributed all over the grid, increasing the number of different Z-aggregates. This has impact on the cardinality α of the largest Z-aggregate, and $E[T_\sigma]$ decreases with increasing system sizes. On the other hand, with multivariate distribution, the value of $E[T_\sigma]$ slightly increases for increasing system size. That is, algorithm NDA with multivariate faults can be considered (almost) scalable. In fact, in this case, faults are grouped, and the number of different Z-aggregates in the grid is smaller. This implies a large value of α that is reflected in the bound T_σ.

The value of $E[T_\sigma]$ for hypercubes with uniform faults is above 99.5% of the system size. This value decreases for increasing fault-set size, as in the case of grid, but it is largely above the size of the fault-set. This implies that the diagnosis of hypercubes returned by algorithm NDA is almost certainly correct. The higher correctness of NDA with hypercubes is due to the richer connectivity of hypercubes as compared with grids.

Moreover, the connectivity of hypercubes increases as $\log n$ with increasing system size. This influences $E[T_\sigma]$, which increases when the size of the system increases. This result is in accordance with the probabilistic analysis of general random graph presented by Scheinermann [19] and Blough et al. [2], where it is showed that the probability of correct diagnosis approaches to 1 as n tends to infinity if the average degree of nodes is at least $\log n + c$ where c is a small constant.

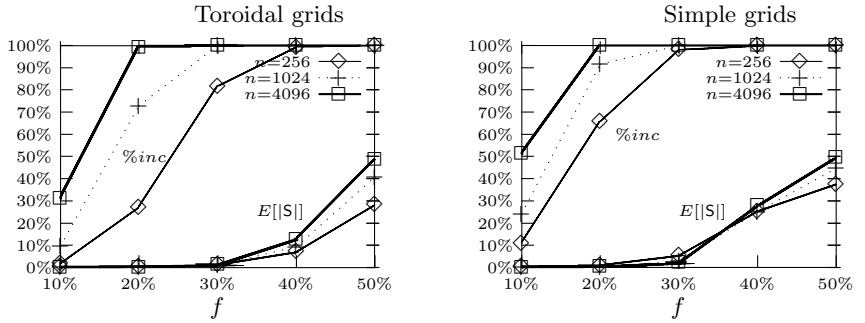


Fig. 3. $E[|\mathsf{S}|]$ as percentage of n, $\%inc$, for toroidal and simple grids. Fault sets with uniform distribution.

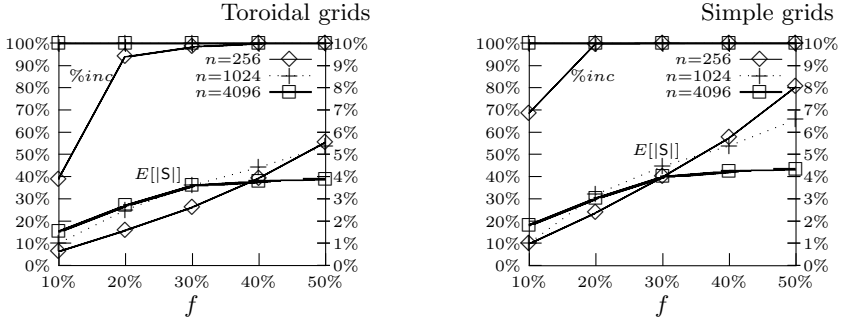


Fig. 4. $E[|\mathsf{S}|]$ as percentage of n, and percentage of incomplete diagnosis ($\%inc$). Fault sets with multivariate distribution.

5.4 Analysis of NDA Completeness

Figure 3 plots the average number of undiagnosed units $E[|\mathsf{S}|]$ as percentage of n (lower curves), and the percentage of executions yielding incomplete diagnosis $\%inc$ (higher curves) for toroidal grids (left plot) and simple grids (right plot) with uniform distribution of faults.

Even with small fault sets, diagnoses are incomplete and the percentage of incomplete diagnosis rapidly increases for increasing system size. As the fault set increases, the percentage of incomplete executions quickly reaches 100%.

On the other hand, the percentage of undiagnosed units is near to zero in presence of fault sets of cardinality below 20%, reflecting a high degree of completeness. The percentage of undiagnosed units increases for increasing fault sets; however, in most cases it remains below 50%. This implies that at least half of the units are correctly diagnosed even with fault sets of high cardinality.

Figure 4 plots the average number of undiagnosed units $E[|\mathsf{S}|]$ as percentage of n (lower curves), and the percentage of executions yielding incomplete diagnosis $\%inc$ (higher curves) for toroidal grids (left figure) and for simple grids (right figure) with multivariate distribution of faults. The scale on the right of both plots refers to the average number of undiagnosed units.

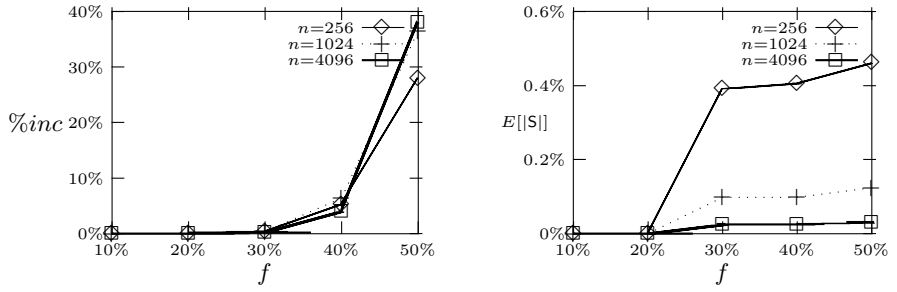


Fig. 5. $E[|\mathsf{S}|]$ as percentage of n, and percentage of incomplete diagnosis $\%inc$ for hypercubes.

The plots highlight the difference between the two faults distributions. With multivariate distribution the percentage of undiagnosed units is below 10%. The analysis of the cardinalities of the fault-free core and the number of Z-aggregates reveals that the multivariate distribution produces a small number of Z-Aggregates and a fault–free core of cardinality larger by an order of magnitude than the uniform distribution. Moreover, this behavior tends to increase with larger fault sets.

The plots of $\%inc$ and $E[|\mathsf{S}|]$ for hypercubes are shown in Fig. 5. While in grids the percentage of incomplete diagnoses rapidly approaches 100% as the relative number of faults increases, in hypercubes this percentage is always below 50%. That is, more than half of the diagnoses executions are complete. Moreover, the percentage of undiagnosed units relative to n is always below 0.5%. The actual number of undiagnosed units is always less than 2, and drops to negligible values with increasing system size. This implies that, in the case of hypercubes, algorithm NDA essentially performs like a one-step diagnosis algorithm.

5.5 Comparison between Algorithms EDARS and NDA

We compare the correctness and completeness of NDA with another diagnosis algorithm for regular structures called EDARS [3,6,7,5,18] by means of simulations. EDARS performs the diagnosis in three steps. Preliminarly, some definitions are recalled:

- The *Disagreement set* is defined as: $\Delta_1(u) = \{v \in \mathsf{V} \mid u \xrightarrow{1} v \text{ or } v \xrightarrow{1} u\}$. Given $\mathsf{U} \subseteq \mathsf{V}$, $\Delta_1(\mathsf{U}) = \bigcup_{u \in \mathsf{U}} \Delta_1(u)$. Under the PMC model, the disagreement set of a fault-free unit (or set) is a set of faulty units.
- Let $u \Rightarrow_0 v$ denote a path from u to v consisting of only 0-labelled edges. The *Zero Descendant set* is defined as: $D_0(u) = \{v \in \mathsf{V} \mid u \Rightarrow_0 v\} \cup \{u\}$. Given $\mathsf{U} \subseteq \mathsf{V}$, $D_0(\mathsf{U}) = \bigcup_{u \in \mathsf{U}} D_0(u)$. Under the PMC model, the zero descendant of a fault-free unit (or set) is a set of faulty-free units.
- The *Zero Ancestor set* is defined as: $A_0(u) = \{v \in \mathsf{V} \mid v \Rightarrow_0 u\}$, $A_0(\mathsf{U}) = \bigcup_{u \in \mathsf{U}} A_0(u)$; by the property of the PMC model, the zero ancestor of a faulty unit (or set) is a set of faulty units.

The first step, called *Local Diagnosis*, partitions the set of units into subsets F, D, and Z. Set F is initially constructed using property (a) of Lemma 1, units in set F are unconditionally identified as faulty. This set is augmented with its *zero ancestor set*. Set D is a set of disjoint pairs with the property that, for every pair, at least one unit must be faulty. The remaining units are assigned to set Z. From construction, adjacent units in set Z test each other with outcomes 0, since otherwise, they should have been put into sets F or D.

In the second step, the connected components of set Z are considered; these components are called Z-aggregates. The different behavior of EDARS and NDA stems from the properties of set D, constructed by EDARS. Although both algorithms construct Z-aggregates, comprising in both cases coalitions of mutually trusting units (i.e., the state of all units in a Z-aggregate must be the same), the aggregates are not the same in the algorithms. In fact, while Z-aggregates constructed by NDA are usually adjacent to other Z-aggregates, in EDARS they are adjacent only to units in set $(\mathsf{F} \cup \mathsf{D})$.

Set D was introduced in EDARS to enforce independence of Z-aggregates, thus overcoming serious difficulties in the study of diagnosis correctness. In fact, if Z-aggregates are independent, the state of each of them can be assigned without affecting the state of the other Z-aggregates. However, the construction of set D affects the size of the Z-aggregates defined by EDARS, which may be smaller in comparison with the Z-aggregates defined by NDA.

Letting α be the maximum cardinality of Z-aggregates, EDARS defines the fault-free core FFC as the union of Z-aggregates of cardinality equal to α. The syndrome dependent bound T_σ is defined as $T_\sigma = \alpha + |\mathsf{D}|/2 + |\mathsf{F}|$. This is the minimum number of faulty units needed to invalidate the assertion of FFC as completely fault-free. Therefore, if the actual number of faults f is below T_σ, the diagnosis of EDARS is correct.

In the last step, called augmentation, EDARS uses fault-free units in the FFC in order to diagnose other units. By the properties of the PMC model, units in set $D_0(\mathsf{FFC})$ are diagnosed as fault-free and units in set $\Delta_1(\mathsf{FFC}) \cup A_0(\Delta_1(\mathsf{FFC}))$ are diagnosed as faulty.

Simulation results for algorithm EDARS are taken from [7] for square grids, and from [18] for hypercubes. The time complexity of algorithm EDARS on k-regular systems, is $O(nk)$ where n is the system size [18]. Thus, both algorithms have linear time complexity.

It is seen that EDARS and NDA perform alike in the average. This result is not surprising, since Z-aggregates constructed by EDARS are proper subsets of those defined by NDA. As seen from simulation, the maximum cardinality Z-aggregate constructed by EDARS is unique with high probability and the remaining aggregates are much smaller. Hence, the FFC constructed by EDARS is, with high probability, a proper subset of the FFC constructed by NDA. However, the last step of EDARS augments the FFC which eventually, with high probability, becomes the same of the FFC identified by NDA. Hence, EDARS and NDA return similar diagnoses in most cases.

When faulty units are uniformly distributed, the average $E[T_\sigma]$ of the syndrome-dependent bound of algorithm EDARS is greater than the corresponding bound of algorithm NDA. With multivariate distribution, however, the average $E[T_\sigma]$ of NDA is greater than the corresponding average of EDARS. An explanation of the above fact is given in the following:

The syndrome-dependent bound of NDA is defined as $T_\sigma = f + \alpha - 1$, where f is the cardinality of the set of unconditionally faulty units, and α is the maximum cardinality of Z-aggregates, the corresponding bound in EDARS is defined as $T_\sigma = f' + \alpha' + |\mathsf{D}|/2 - 1$.

Simulation results proved that in the average the value $f' + \alpha'$ is smaller than $f + \alpha$. Thus the difference between the two bounds depends on the contribution of the value $|\mathsf{D}|/2$ to the syndrome-dependent bound of EDARS. In the case of multivariate distribution, the value of $f' + \alpha'$ is almost equal to $f + \alpha$, i.e., since fault-units are clustered, the size of the maximum Z-aggregate constructed by EDARS is almost equal to the corresponding maximum Z-aggregate of NDA, although the value of $|\mathsf{D}|/2$ is relatively small, it is sufficient to have $f' + \alpha' + |\mathsf{D}|/2 > f + \alpha$.

Note, however, that the higher syndrome-dependent bound of EDARS does not imply a higher degree of correctness with respect to algorithm NDA. In fact, both algorithms satisfy the correctness condition $f < E[T_\sigma]$ when $f < 0.5n$; thus, they are both correct. The difference between the two algorithms increases as the fault set cardinality also increases. Algorithm NDA has a slightly better degree of completeness, i.e., $E[|\mathsf{S}_{\mathsf{NDA}}|] < E[|\mathsf{S}_{\mathrm{EDARS}}|]$.

Since algorithm NDA provides a diagnosis with a slightly better degree of completeness, without compromising the diagnosis correctness, it is an improvement with respect to EDARS.

5.6 Comparison with Somani and Agarwal Diagnosis Algorithm

Somani and Agarwal [20] presented a diagnosis algorithm for regular structures (SA for short) based on the concept of local diagnosis. This algorithm classifies a given unit as locally–good (LG) if it is tested with outcome 0 by a majority of its neighbors, otherwise it is classified locally–faulty. After this classification, units are declared as *good* if they are trusted (tested with outcome 0) by a majority of locally–good units. Good units are considered as fault-free and their test outcomes are used to "expand" the diagnosis to other reachable units. If the resulting diagnosis is incomplete, the remaining units are somehow declared to be either good or faulty. Clearly, this may lead to incorrect diagnosis.

The authors reported simulation studies of diagnosis correctness and completeness taking into consideration square toroidal grids. Their results are compared with the results of NDA in Tables 2 and 3. Columns CC and INC report the percentage of correct and complete diagnoses, and the percentage of incorrect diagnoses, respectively, for both SA and NDA.

Table 2 reports results obtained for small toroidal grids with fault set ranging from 4 to 7. Distribution of faults is uniform and NDA's parameters p, q are set

Table 2. Comparison between NDA and Somani's diagnosis algorithm in small toroidal grids.

Fault Size ⟶	4			5			6			7		
$l \times l \downarrow$	CC	INC	CC NDA	CC	INC	CC NDA	CC	INC	CC NDA	CC	INC	CC NDA
4×4, $n = 16$	100%	0%	98.8%	99.63%	0%	96%	84.31%	7.2%	88.6%	49.37%	45.1%	74.7%
5×5, $n = 25$	100%	0%	99.9%	99.95%	0%	99.3%	99.35%	0.23%	97.8%	96.41%	1.94%	92.8%
6×6, $n = 36$	100%	0%	99.8%	99.97%	0%	99.4%	99.91%	0.04%	99%	99.7%	0.17%	97.8%

Table 3. Comparison between NDA and Somani's diagnosis algorithm in large toroidal grids.

Mean Fault Size ⟶	l			$2l$			$4l$		
$l \times l \downarrow$	CC	INC	CC NDA	CC	INC	CC NDA	CC	INC	CC NDA
32×32, $n = 1024$	99.9%	0%	100%	98.4%	0.11%	98.7%	77.2%	6.87%	78.9%
64×64, $n = 4096$	99.9%	0%	100%	99.6%	0%	99.8%	93.4%	0.68%	94.6%
128×128, $n = 16384$	100%	0%	100%	99.9%	0%	99.9%	98.3%	0.02%	98.8%
256×256, $n = 65536$	100%	0%	100%	99.9%	0%	100%	99.6%	0%	99.8%

to 0.5. Table 3 reports results obtained for relatively large toroidal grids with fault set ranging from l (the side of the grid) to $4l$.

With small fault sets, it is seen from Table 2 that the diagnosis provided by SA is always correct and complete, in contrast, the diagnosis of NDA may be incomplete. As the size of the fault set increases (e.g., in the case of 7 faults in a 4×4 toroidal grid), the diagnosis of NDA is always correct and almost complete, while in SA the occurrence of incorrect diagnosis becomes frequent. With large toroidal grids, it is seen from Table 3 that NDA outperforms SA in all simulation experiments. The advantage of NDA is more evident as the fault set becomes a larger fraction of the grid size.

The different performance of SA and NDA is easily explained by the different design philosophies of the algorithms. In NDA, diagnosis correctness is the strong requirement and incomplete diagnosis are allowed whenever correctness could be impaired. SA is a probabilistic algorithm providing a diagnosis of every unit, even if the correctness cannot be fully trusted.

The diagnosis of NDA is incomplete if the set of suspect Z-aggregates is not empty. Units in the suspect Z-aggregates are not diagnosed by NDA, while in SA they are guessed as good or as faulty, thus leading to a complete although possibly incorrect diagnosis.

6 Conclusions

In this paper we presented an efficient diagnosis algorithm and analyzed its correctness and completeness by means of simulations. Algorithm NDA diagnosis is correct (whatever is the syndrome) if the number of faults is not above the syndrome-independent bound $T(n)$ presented in [4]. The syndrome-dependent bound T_σ reported by NDA has been studied, in the average, by means of simulations. Simulations show that T_σ is above the actual fault-set size, thus NDA is always correct provided that $f \leq T_\sigma$.

NDA has been compared with two diagnosis algorithms for regular system, EDARS and SA. The comparisons show that NDA has a higher degree of diagnosis completeness; it is able to diagnose a large fraction of the system even with a higher number (near $0.5n$) of faults.

We believe that the higher degree of diagnosis correctness and completeness achieved by NDA are a good indication that it is well-suited for wafer-scale testing of integrated systems, one of the most promising application of system-level diagnosis.

References

1. L. Baldelli and P. Maestrini. Self-diagnosis of array processors. In *Proceedings of the* 24^{th} *International Symposium on Fault-Tolerant Computing*, pages 48–53, Austin, Texas, June 1994.
2. D. Blough, G. Sullivan, and G. M. Masson. Efficient diagnosis of multiprocessor system under probabilistic models. *IEEE Transactions on Computers*, 41(9):1126–1136, September 1992.
3. A. Caruso, S. Chessa, P. Maestrini, and P. Santi. Diagnosis of regular structures. In *IEEE Proc. Dependable Systems and Networks*, pages 213–222, New York, NY, June 2000.
4. A. Caruso, S. Chessa, P. Maestrini, and P. Santi. Diagnosability of regular structures. *Journal of Algorithms*, 45:126–143, November 2002.
5. A. Caruso, S. Chessa, P. Maestrini, and P. Santi. Evaluation of a diagnosis algorithm for regular structures. *IEEE Transactions on Computers*, 51(7):850–865, July 2002.
6. A. Caruso, S. Chessa, P. Maestrini, and P. Santi. Fault–diagnosis of grid structures. *Theoretical Computer Science, Elsevier*, January 2003.
7. Stefano Chessa. *Self-Diagnosis of Grid-Interconnected Systems with Application to Self-Test of VLSI Wafers*. PhD thesis, TD-2/99, University of Pisa, Italy, Department of Computer Science, March 1999.
8. A. T. Dahbura and G. M. Masson. An $o(n^{2.5})$ fault identification algorithm for diagnosable systems. *IEEE Transactions on Computers*, C-33(6):486–492, June 1984.
9. S. L. Hakimi and A. T. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Transactions on Computers*, C-23(1):86–88, January 1974.
10. K. Huang, V. K. Agarwal, and K. Thulasiraman. A diagnosis algorithm for constant degree structures and its application to vlsi circuit testing. *IEEE Transaction on Parallel and Distributed Systems*, 44(4):363–372, April 1995.
11. S. Huang, J. Xu, and T. Chen. Characterization and design of sequentially t-diagnosable systems. In *IEEE Proc.* 19^{th} *International Symposium on Fault-Tolerant Computing*, pages 554–559, Washington, D.C., USA, June 1989.
12. A. Kavianpur and K.H. Kim. A comparative evaluation of four basic system-level diagnosis strategies for hypercubes. *IEEE Transactions on Reliability*, 41(1):26–37, March 1992.
13. S. Khanna and W. K. Fuchs. A graph partitioning approch to sequential diagnosis. *IEEE Transactions on Computers*, 46(1):39–47, January 1997.
14. L. E. LaForge, K. Huang, and V. K. Argarwal. Almost sure diagnosis of almost every good element. *IEEE Transactions on Computers*, 43(3):295–305, March 1994.

15. P. Maestrini and C. L. Liu. On the sequential diagnosability of a class of digital systems. In *IEEE Proc.* 11^{th} *Int. Symp. on Fault-Tolerant Computing*, pages 112–115, June 1981.
16. F. P. Preparata, G. Metze, and R. T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Computers*, EC-16(12):848–854, December 1967.
17. V. Raghavan and A. Tripathi. Sequential diagnosability is co-NP complete. *IEEE Transactions on Computers*, 40(5):584–595, May 1991.
18. Paolo Santi. *Evaluation of a Self-Diagnosis Algorithm for Regular Structures.* PhD thesis, TD–6/00, University of Pisa, Italy, Department of Computer Science, March 2000.
19. E. R. Scheinerman. Almost sure fault tolerance in random graphs. *SIAM Journal on Computing*, 16(6):1124–1134, December 1987.
20. A. K. Somani and V. K . Agarwal. Distributed diagnosis algorithm for regular interconnected systems. *IEEE Transaction on Parallel and Distributed Systems*, 42(7):899–906, July 1992.
21. G. Sullivan. A polynomial time algorithm for fault diagnosability. In *Proceedings of the* 25^{th} *Int Symp. on Foundations of Computer Science*, pages 148–156, Los Angeles, Ca., USA, October 1984.
22. G. Sullivan. An $o(t^3 + |\mathsf{e}|)$ fault identification algorithm for diagnosable systems. *IEEE Transactions on Computers*, 37(2):388–397, April 1988.

A Tool for Fault Injection and Conformance Testing of Distributed Systems

Eliane Martins[1] and Maria de Fátima Mattiello-Francisco[2]

[1]Institute of Computing (IC)
State University of Campinas (UNICAMP)
eliane@ic.unicamp.br

[2]Ground Systems Division (DSS)
National Institute for Space Research (INPE)
Av. Dos Astronautas, 1758 - São Jose dos Campos -12227-010 - SP - Brazil
FAX: +55-12-345-6625,
fatima@dss.inpe.br

Abstract. This paper presents an approach for conformance testing and fault injection of distributed systems supported by a tool named FSoFIST (Ferry-clip with Software Fault Injection Support Tool). The approach extends the ferry-clip concept to cope with fault injection. The ferry-clip concept was aimed at providing a highly modular, flexible and configurable architecture for protocol conformance testing. Due to these qualities, this architecture can be used for testing different protocol implementations with reduced effort. The work presents the design issues employed to achieve the ferry-injection architecture and describe the components of the proposed architecture. The capabilities of the approach are demonstrated in a case study used to validate the FSoFIST tool.

1 Introduction

On the last decades a continuous increase in the use of distributed systems has been observed in several applications: industry, offices, research and education institutions, banks, commercial organizations, and hospitals. The Internet extended the use of these systems to the domestic environment, supporting applications such as „home banking“ and e-business among others.

Dependability becomes then an important property of such systems, as even more activities depend on their good performance. For that reason, dependability validation is more and more important, in order to guarantee that the developed system has the expected properties. In this paper our concern is with testing, aimed at revealing faults in a system's implementation.

Testing distributed systems usually requires various types of tests [2]: conformance, interoperability, quality of service testing. *Conformance testing* is aimed to determine if an implementation meets its functional specification. The *interoperability tests* aim to determine whether various components of a distributed

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 282–302, 2003.

system are able to cooperate with each other to perform the specified services. In the *quality of service (QoS) testing*, the purpose is to assess the behavior of a distributed system to determine whether attributes such as performance, reliability or availability meet the expected standards. Complimentary to these techniques, fault injection allows validating the system in the presence of a variety of faults or errors, even those not anticipated in the specification.

Distributed systems consist of a number of independent components running concurrently on different machines that are interconnected by a communication network. Testing distributed systems is a difficult and challenging task for several reasons. One problem is the generation of adequate test cases that present good potential for uncovering faults. Another problem is the generation of the expected results for each test case. Generally these tasks are performed manually, but in the case of distributed systems the number of potential input sequences that the system can handle is infinite, which means that a subset of the possible input sequence represents indeed too much testing. This makes the cost of having people generating test cases and evaluating test data unfeasible. Although an important issue, test case generation and results analysis are not discussed further in this text. These were the concerns of two other tools developed by the group and can be seen in [16, 22].

Here our concern is with test execution support. One challenge is the intrinsic non-determinism of distributed systems. Non-determinism may occur when the implementation under test (IUT) cannot be accessed directly. For example, a protocol implementation is tested via an underlying communication service, or it is embedded in other protocol layers. This limits the view that the test component has of the system and the ability to reproduce the same IUT behavior when repeating the same input conditions. To reduce non-determinism, one has to introduce various test components into the System Under Test (SUT), which increase intrusion in the original system to be tested. Those test components may alter either the system structure (e.g., some fault injectors are implemented as extra code inserted either at the IUT or into libraries it access) or the behavior of the IUT, for example, by altering its execution speed, which may be a problem when testing real-time systems. Moreover, this requires distributed test components and, consequently, some mechanism to synchronize them.

Typically, distributed systems are heterogeneous in terms of communication networks, operating systems, hardware platforms and also the programming language used to develop individual components. This represents another challenge, in that a test system must be able to run in a wide variety of platforms and has to access different kinds of interfaces.

This text presents a tool built to support conformance testing, with the aim to give an answer the question: does the system comply with its functional specification? Fault injection is used as complement, in that it allows answering questions such as: how does the system react when whenever faced with unexpected or invalid behavior of its environment? The tool design is based on the *ferry architecture* [28] proposed in the context of protocol testing. The tool, named FSoFIST (Ferry-clip with Sofware Fault Injection Support Tool) [1], extends this architecture in order to support fault

injection by software. The ferry architecture was adopted because of the following features:

- It offers the means to introduce test instrumentation inside the IUT with low intrusiveness;
- It provides the means to synchronize the test components;
- It is portable for different platforms and easy to adapt to different kinds of interfaces
- It is highly modular, which eases modifications;
- It can be easily extended to cope with testing of multiple IUT.

The paper is organized as follows: Section 2 presents basic notions about the two types of test currently supported. In Section 3 we present the proposed test architecture. Section 4 presents an implementation of the proposed test architecture, performance evaluation of the tool and also a case study for its validation. Section 5 describes some related works. Finally, section 6 concludes the paper. The appendix contains a list of abbreviations used in this text.

2 Types of Tests Supported

2.1 Conformance Testing

Conformance testing aims to determine whether an implementation meets the specification [12]. In the context of the Reference Model for Open Systems Interconnection (OSI), it was developed a particular standard for protocol conformance testing, the IS 9646: „OSI *Conformance Testing Methodology and Framework*“ (CTMF) [14]. The standard defines the methodology, structure and specification of test sequences as well as the procedures to be followed. The purpose is to improve the capability of both comparing and reproducing the tests results performed by different groups. The standard does not establish the way that the tests should be generated, rather, it defines a framework for structuring and specifying the tests.

The CTMF also defines conceptual architectures (designated as abstract test methods) to support test execution (ISO IS-9646, 1993, Parts 1 and 2). A test architecture describe how an IUT (which represents a protocol entity) is to be tested, i.e., what inputs can be controlled and what outputs can be observed. As can be seen in figure 2.1 a, the points at which inputs and outputs to/from the IUT can be controlled and observed are called the Points of Control and Observation (PCO). The specification of a protocol of the OSI reference model describes the behavior of an entity in terms of the inputs and the outputs passing by the upper and lower service access points (SAP), respectively named (N)-SAP and (N-1)-SAP.

Ideally each SAP is a PCO that is directly used for the communication with the IUT. But generally one or more SAPs are not directly accessible during testing. An

IUT may be embedded in other applications or may be only accessed via underlying services. For these reasons, test architecture might be described in terms of [23]:

(i) **the PCOs**,
(ii) **the test context**, that is, the environment in which the IUT is embedded and that is used during testing to interface the IUT (the test context, also called test interface, is supposed to be correct);
(iii) **the testers** – there is a tester associated to each PCO, they are named upper tester (UT) and lower tester (LT) respectively connected to (N)-SAP and (N-1)-SAP. Whenever both testers are used, synchronization between them during testing is required; this is achieved by the Test Coordination Procedure (TCP).

Since, in real world, the IUT and the testers can reside on different machines, the following 4 architectures were proposed [14, 23] that can be classified as local or external. Figure 2.1 illustrates these architectures.

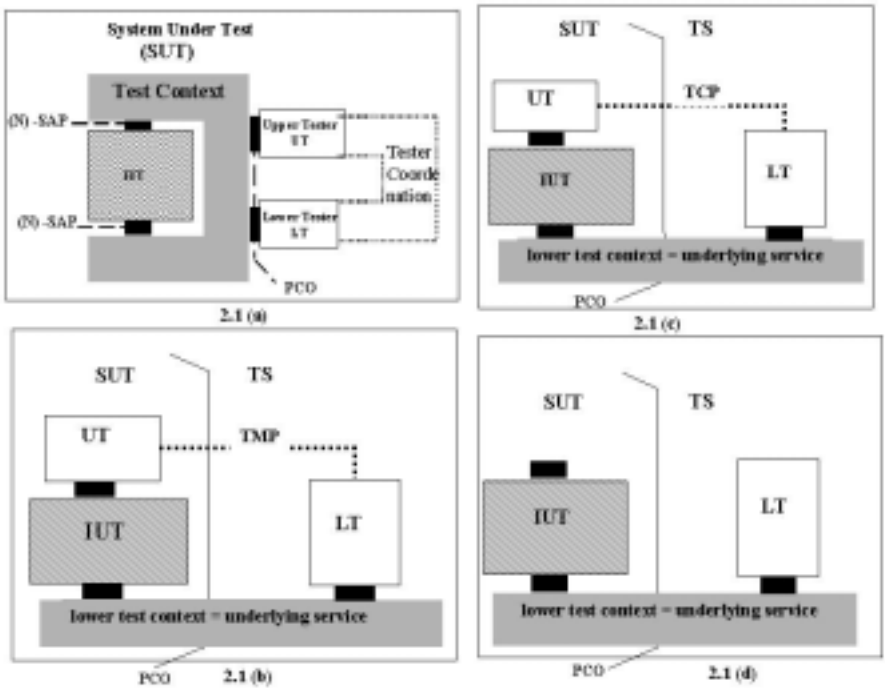


Fig. 2.1. Conformance testing architectures.

In the local architecture (figure 2.1(a)), the IUT interacts with both testers, UT and LT, intercepting the SAP (Service Access Point) in the upper and lower interfaces of IUT, which are the tests PCOs.

The other three architectures are external, in which case the LT has remote access to the IUT. The LT resides remotely in the Test System (TS) and its interaction with

the IUT is through the (N-1) layer. Test Coordination Procedure (TCP) shall implement a communication protocol between the TS and the LT. In the distributed architecture (figure 2.1(b)), the UT is physically close to IUT, directly accessing its upper interface. The Coordinated Method (Figure 2.1(c)) uses a Test Management Protocol (TMP) as TCP. In the remote architecture, (Figure 2.1(d)), the UT does not access directly the IUT upper interface

Each test architecture has three variant forms [14]: single-layer, multi-layer and embedded. In single layer-methods, the IUT represents a single layer of a protocol stack and is tested without reference to the layers above it. Multi-layer methods are designed for testing a multi-layer IUT as a whole. In the embedded method a single-layer is tested within a multi-layer IUT; the SAP of the single-layer being tested is accessed through the layers above it, which constitutes the upper test context.

The architectures described so far are aimed at testing an IUT that communicates, in a point-to-point connection, with a single peer. This is the single-party testing context. A generalization of these architectures was proposed for the case in which an IUT participates in a multi-point connection with several peers at the same time. This constitutes the so-called multi-party testing context, in which several LT and zero or more UT are used to control and observe the IUT. Since our concern in this paper is single-party testing context, we will not discuss multi-party testing issues any further. Interested readers can consult references [6, 24] for further details on this topic. Also, [27] presents a good survey on distributed test architectures for protocol conformance testing according to CTMF and beyond.

2.2 Fault Injection

Fault injection consists in the deliberated insertion of faults or errors into a system aiming at observing its behavior. This technique is very useful to validate the implementation of error recovering and exceptions mechanisms, as well as to determine the system behavior in presence of faults in the environment.

There are several approaches for fault injection [2, 7, 13]. These work addresses fault injections by software, in which changes in the state of the system under test, are performed under control of special software. In this way both hardware and software failure modes might be emulated. The mechanism consists in interrupting the IUT execution to run the fault injector code. The latter can be implemented in various forms: as a routine started by a high priority interruption; as a routine started by a trace mechanism; as an extra code inserted into the IUT or in its context (operating system, underlying communication layer, for example). A pioneering work in [21] implements some of these mechanisms.

The fault injector implemented in ATIFS aims to mimic communication faults, which represent typical faults of distributed systems like message lost, duplication, corruption and delay. Communication fault injectors are generally inserted by a probe/fault injection layer between the IUT and the underlying service [9, 10, 11, 19, 20], as shown in Figure 2.2.

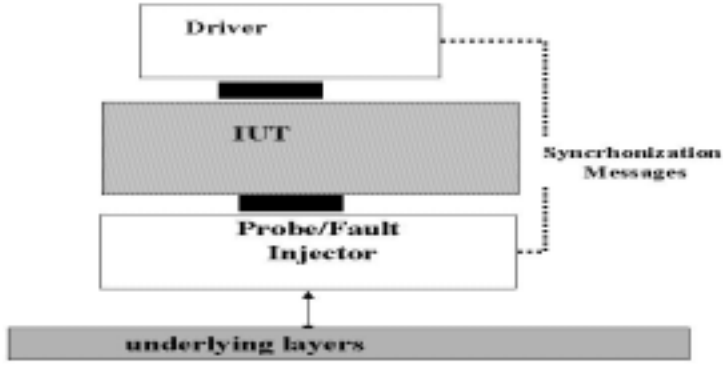


Fig. 2.2. The probe/fault injection approach

Typically, fault injection is used later in the test process, mainly to obtain dependability measures. Here we are concerned with the testing of one IUT, and fault injection is used to ensure that erroneous messages can be generated as required by the test case being run.

3 FSOFIST Design Overview

3.1 The Proposed Ferry-Injection Architecture

The *ferry* architecture was proposed in [28, 29] to support protocols test architectures defined under ISO9646 standard. This architecture has a high modular structure, which allows its use in different platforms requiring few modifications. Basically, it consists of transporting the test data from the Test System (responsible to the test coordination) to the SUT, which contains the IUT, in a transparent way to allow those both upper and lower testers reside in the Test System. This approach simplifies the synchronization between the UT and LT, minimizing the amount of test software to be aggregated to the SUT. The study in [6] has shown the flexibility of the ferry architecture in supporting other types of testing. The architecture proposed here, designated as ferry-injection, extends the ferry-clips to support fault injection.

Figure 3.1 illustrates the distributed architecture proposed. The Active Ferry (AF) and the Passive Ferry (PF) are the main elements of the ferry architecture. They are responsible for the test data transfer according to a simplified ferry-protocol. Additionally, they adapt the data into the format understandable by the IUT, so the Test Sequence Controller is not modified to each new IUT.

As shown in figure 3.1, the test data are transported from the Test System (responsible for the test coordination) to the System Under Test by the *ferry transfer medium,* which is an IUT-independent communication channel.

The test manager (TM) starts and stops a test session automatically or under a user command. It provides a user interface allowing the user to follow the executed test steps and to enter information before the test execution. It activates and deactivates the Test System components.

The Test Sequence Controller (TSC) applies a test sequence to the IUT, using the ferry clip protocol according to the available PCOs. The Fault Injection Controller (FIC) controls fault injection testing, according to the script. It sends information about the faults to be injected to the FIM and stores data collected by the latter. The Fault Injection Module (FIM) intercepts the messages received by the IUT and inserts the faults determined by FIC.

The current version of FSOFIST is aimed at single-party testing, in which there is a single IUT that participates in a point-to-point communication with its peer. In this context, fault injection objective is only to ensure that errors are generated as required by the test case being run. The IUT can represent a single layer or multiple layers of a protocol stack, considered as a whole.

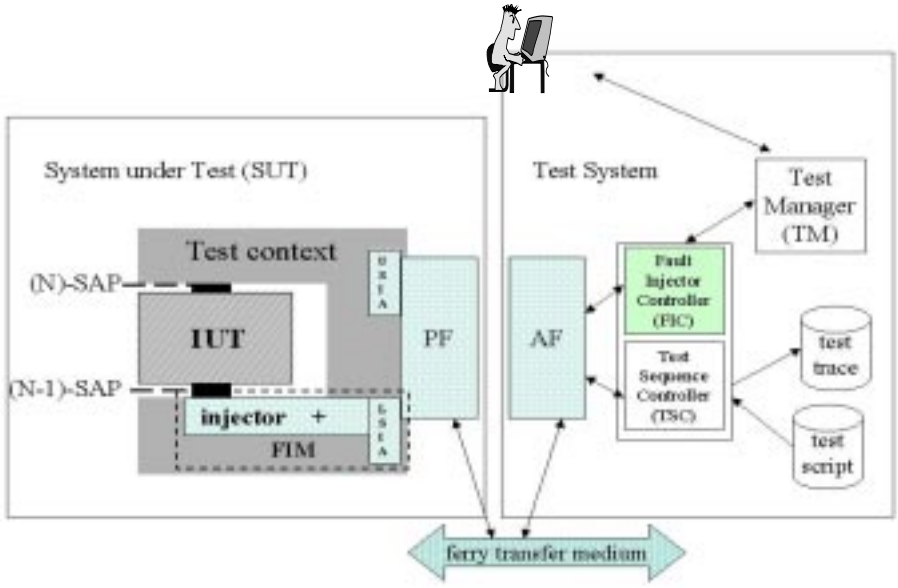


Fig. 3.1. FSoFIST architecture

An important aspect concerning software fault injection tools is intrusiveness: the instrumentation introduced for injection and monitoring purposes may affect the structure (structural intrusiveness) or disturb the execution of the target system (behavioral intrusiveness). Our approach attempts to minimize intrusiveness in the following ways: (i) only the IUT dependent part of the fault injection features resides on the System Under Test (SUT), in that way reducing space overhead on this system; (ii) the FIM is part of the PF test component, affecting messages passing through the

IUT lower interface. In this way, IUT code is not necessarily modified, which avoids structural intrusiveness.

In the following, the major components of this architecture are described. Unless otherwise stated, the components functions are identical to that indicated in the literature [5, 6, 29] which can be consulted for further details.

3.2 The Ferry Protocols

The transfer of data between the AF and PF is achieved through the Ferry Control Protocol (FCP). Since the FCP is implemented in each ferry-clip, it should be as simple as possible, to simplify their implementation. The FCP defines abstract service primitives (FCP-ASP) that allow open and close a connection and the transfer of data between the two ferries. It also defines ASPs for the communication among the test components and the ferries. They are not presented here, for sake of space. Interested readers can consult [1] for further information The FCP uses services provided by the Ferry Transfer Medium Protocol (FTMP). To simplify matters to the FCP, this protocol has the following requirements: (i) guarantees end-to-end error free delivery of data and (ii) deliver packets in the same sequence they were sent. The SUT must provide such a protocol. In the present implementation of FSoFIST, the TCP/IP is used as the FTMP.

3.3 The Ferry-Clips

The structure of both ferry-clips is quite similar. They are structured into three modules:

- Ferry Engine (FE), which implements the core functions of the ferry-clip. It contains all functions that are independent of the IUT and FTMP being used. It interacts with the other test components by means of FP-ASP.
- Lower Mapping Module (LMAP), which maps the FP-ASP into the FTMP-ASP. Thus, this is the module to change for different implementations of the FTMP.
- Service Interface Adapter (SIA) which provides interface to the IUT. Since in the implemented test architecture the PF has access to both IUT interfaces, this module was not implemented in the AF. In the Passive Ferry this module converts data received through the ferry transfer medium into a format used by the IUT, and conversely, i.e., it converts IUT outputs into a format that the Test Suite Controller (TSC) understands. The PF-SIA is then logically structured in two sub-modules: the U-SIA, associated with the IUT upper interface, and the L-SIA, associated with the lower interface, as shown in Figure 3.3. Thus, the PF-SIA module is IUT-dependent, hence, it is the only module to be replaced if a different IUT is to be tested.

An important concern when designing the ferry clips is to keep the PF as small and simple as possible. This can be achieved, on the one hand, by choosing a simple, yet reliable, ferry-transfer medium protocol. On the other hand, the conversion functions performed by the PF-SIA should be reduced to a minimum. One possible solution consists in externally specifying the IUT interface formats instead of coding the conversion into the PF-SIA. This is the solution we adopted for FSoFIST. The primitives (or PDUs), their respective parameters, as well as parameter length and the allowable range for each parameter are input to another tool, ConDado [16], that automatically generates test cases in the adequate format.

3.4 The Fault Injection Module

The Fault Injection Module (FIM) is an extension of the PF-SIA to incorporate fault injection capabilities. More precisely, it extends the L-SIA, given that communication fault injectors typically resides in some layer bellow the IUT (see section 2.2). When no fault injection is used for testing, the user can configure the PF to use the L-SIA instead of the FIM. It contains the Injection Logic, which performs fault-injection functions, and a message buffer to store messages to/from the IUT.

The FIM first processes the fault descriptor which was sent by the FIC. Then, it runs through a loop, intercepting messages from/to the IUT and checking whether a message should be injected or not, according to the fault descriptor. A message counter is used by the FIM to trigger fault injection. In this way, the FIM is able to inject various fault types such as: message delay (a message from/to the IUT is retained until a certain delay is up), message dropping (a message from/to the IUT is deleted), message duplication (a copy of a message from/to the IUT is maintained in the buffer and sent after a certain delay) and message corruption (the content of a message is modified). It can also inject faults permanently (affecting every message), intermittently (affecting certain messages according to a pre-specified frequency) or transiently (affecting only one message in a test run).

These fault types are applied only on incoming messages. When no faults are to be introduced, FIM only transfers messages directly to the IUT.

3.5 The Fault Injection Controller

The Fault Injection Controller (FIC) implements all fault injection functions that are IUT independent. Hence, changing the IUT has no effect on this module. It uses a script defined by the user determining: when to start fault injection, what faults to inject and for how long. This information is used to pass the appropriate commands to the FIM using the Ferry Protocol primitives. This module also stores data sent back by the FIM for post-mortem analysis.

3.6 Test Specification

The CTMF defines a test notation for test case specification: it is the Tree and Tabular Combined Notation (TTCN) [14, 18]. However we decided to use TCL for that purpose, for the following reasons: (i) at the time the tool was designed, TCL was a popular interpreted language, used in some fault injection tools (e.g, [9]); (ii) TCL syntax is quite similar to C, which reduces the burden of learning a new language for users already familiar with this programming language and (iii) TCL allows users to write their own extensions, in C or C++, so that the script can invoke user defined procedures.

3.7 Requirements on the System under Test

It is important to point out that there are also requirements on System Under Test (SUT) in order that our test architecture could be used. First of all, the SUT must allow the insertion of test components to control and observe the IUT interfaces. Another point is that it should allow the installation of a tester in a layer bellow the IUT to perform fault injection. Furthermore, it must provide an IUT-independent communication medium for the transfer of information between the Test System and the SUT.

4 Implementation of FSoFIST

FSoFIST was developed according to the ferry-injection architecture described in the previous Section. It presents a user interface allowing the tester to control and monitor the tests step by step, allowing the user to start, stop and continue a test session any time.

FSoFIST was developed under the Solaris 2.5 operating system, where it was first used. It was further ported to Linux. Its PF component ran initially in Solaris but was transferred to Windows platform for the case study presented in section 4.4.2. The AF is written in C++ language and the PF in Perl. The communication between them uses the socket library and the TCP/IP protocol, which guarantees portability, since socket libraries are available in various Operating Systems.

In the following we present the experiments performed to validate FSoFIST. The first experiments were aimed at evaluating the tools performance. The second one illustrates the use of FSoFIST in a simple, real world application.

4.1 Performance Measurements

The SUT used in these experiments was a dual Xeon 300MHz CPU running a Linux Operating System. The PF and the IUT resided both in the SUT. The IUT in this case does not matter, since we were only interested in measure the overhead introduced by

the test components. The Test System was implemented on an Ultra SPARC 366 MHz running Solaris. The two hosts were connected through a FastEthernet network.

The Test Manager implements a TCL Interpreter, which manages the execution of the testers (FIC and TSC). The Test System components were implemented as objects, and were written in C++.

The performance evaluation experiments are summarized bellow.

Measuring script interpretation overhead. These experiments were aimed at evaluating the tool's execution delay caused by the TCL Interpreter. For those purposes, four scripts were generated: a null script, and three scripts that executed a loop for 1000 times, 10000 times and 100000 times, respectively. The script is shown bellow and the results obtained are presented in Table 4.1.

```
set x 0
while ($x < 1000 ) { incr x }
```

The time taken to process the null script indicates the TCL Interpreter overhead. By subtracting this overhead from the times taken to execute the loops we obtain that each assignment operation takes about 7μs (e.g., 208 (the time taken to process the loop 10000 times) - 137 = 71 ms /10000 assignments).

Table 4.1. Script interpretation delays

Script	Execution time
Null script	137 ms
Loops 1000 times	141 ms
Loops 10000 times	208 ms
Loops 100000 times	684 ms

Measuring TSC-AF communication delay. These experiments were aimed at evaluating the execution delay caused by the communication internal to the Test System. For that purpose, we vary the size of the data exchanged between two modules: Test Suite Controller and AF. Three buffer sizes were considered: 10, 100 and 1000 bytes respectively. The script is shown bellow and the results obtained are presented in Table 4.2.

```
set data_buffer „xxxxxxxxxx"
time {
       senddata Lower data_buffer
     } 1000
```

Table 4.2. Internal communication delays

Number of bytes	Execution time
Empty data	13,353 s
Data with 10 bytes	13, 458 s
Data with 100 bytes	19,993 s
Data with 100 bytes	26,550 s

The internal communication is through pipes, and we considered that each pipe has the same execution speed, so only one pipe was observed. The connection with the PF was not opened, which causes the AF to respond to each request with a disconnect indication.

Measuring AF-PF communication delay. These measurements were aimed at assessing the response time in the ferry transfer medium. The PF was executed in loop back mode, that is, all data received was sent back to the AF. The script used in these experiments is presented bellow, and the results are in Table 4.3.

```
# opening the connection with AF:
openconn <PF#>  <host>  <port>
openconn 1 lua 2345

# putting PF in loopback mode
sendcmd 1 Loopback

# sending nb bytes buffer and awaiting to receive it
# back for 1000 times:
senddata <PF #> <SAP> <data>
time {
    senddata 1 Lower "xxx ...<nb bytes> ... xxx"
    recvdata 1 Lower
     } 1000

# closing the connection with the PF
closeconn 1
```

The measures in table 4.3 show the time taken by the TCL Interpreter to process the script as well as the internal communication overhead and the data transfer using TCP/IP as Ferry Transfer Management Protocol (FTMP).

The overhead in the communication between PF and IUT depends on the interfaces between these components, and for that reason it is not presented here.

Table 4.3. Ferry transfer delays.

Number of bytes (nb)	Execution time
nb = 0	20.89 s
nb = 100	57.75 s
nb = 1000	63.89 s
nb = 100000	86.10
nb = 1000000 bytes	244.6 3 s

4.2 Experiments with MASCO

The Telemetry Reception Software (TMSTATION) of the MASCO Telescope [25] implements a ground entity of the ground-board communication protocol which receives in real-time the data of the sky imaging in X-rays got by the MASCO tele-

scope (a Brazilian space project). The data are acquired during approximately 30 hours during the telescope flight on board of a balloon mission. The experiment was developed by the Astrophysics Division at the National Institute of Space Research (INPE) and will be launched yet in 2003. The imaging data acquired by a hardware detector are organized in frames, stored in files on board and transmitted in real time to the Ground Station, where they are received by the TMSTATION software [17], as illustrated in figure 4.1.

The main function of TMSTATION is to separate the frames, sequentially received through a serial channel RS422, in distinct files, like they are stored on Hard Disk on board. The separation is based on the identification of a pattern, e.g. a string of at least 5 occurrences of the hexadecimal word „AA55“, which shall be presented at the end of each frame. In the rest of the text we will refer to this word as the end-of-file (EOF) pattern. The software specification also required the TMSTATION application to report the status of each frame stored on ground according to the frame length. The TMSTATION was implemented in C language under LabWindow/ CVI [15] environment.

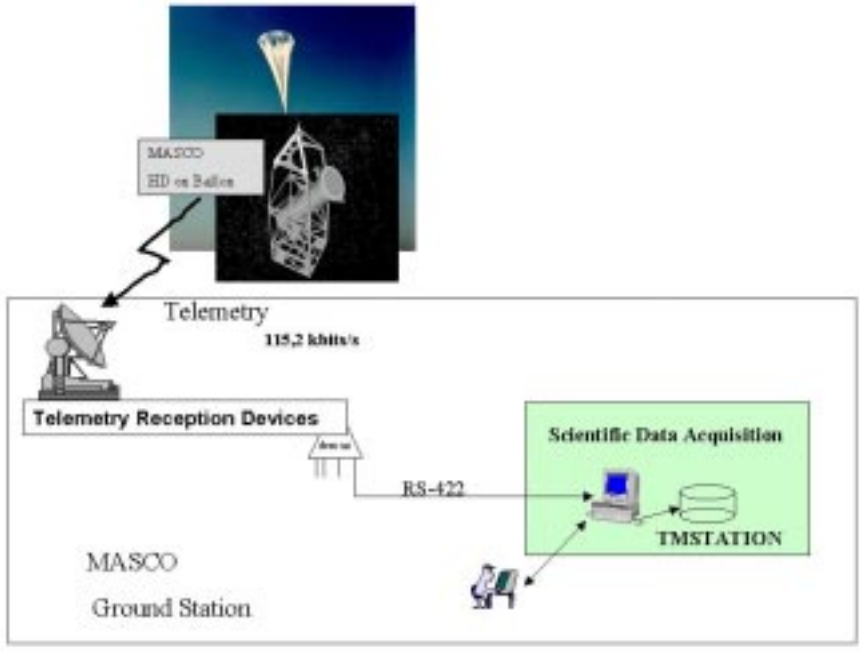


Fig. 4.1. The TMSTATION software operation context for MASCO scientific data reception on ground.

Under normal transmission conditions from board to ground, the valid frame to be recorded at ground station has the following format: first, the word „EB90“, followed by „937 x 146 words“ (data field), finally the „end of file“ pattern. In case of communication failure, part of the frame might be missed resulting:

(i) **Truncated frame**: a frame with reduced length is generated whenever part of the frame data field is lost, without affecting the EOF pattern
(ii) **Extended frame**: a frame with bigger length might occur if part of the EOF pattern is corrupted or lost. In this case, the TMSTATION aggregates the next frame to the current frame data field.

In order to validate whether the TMSTATION implementation deals correctly with the reception of valid frames, we used conformance testing. To emulate invalid frames, as described above, we used fault injection with the aim to mimic communication failures on the radio frequency downlink.

The FSoFIST architecture, presented in figure 3.1, was configured to achieve the TMSTATION testing needs as shown in figure 4.2. The physical serial channel RS-422 connects the two processes, PF and IUT, both residing on the same machine, on an Windows environment. The Lab-CVI library was used for the communication through the serial line, acting then as the test context. Faults are injected to emulate ground-board link failures. For that reason, we assumed that messages transferred to the IUT through the serial line are delivered in sequence, without modification. So, the FIM was not introduced inside the test context, as it should be (c.f. Section 3.1); instead, it uses this context to interface with the IUT. This configuration was useful given that our main interest was to validate FSOFIST module's behavior.

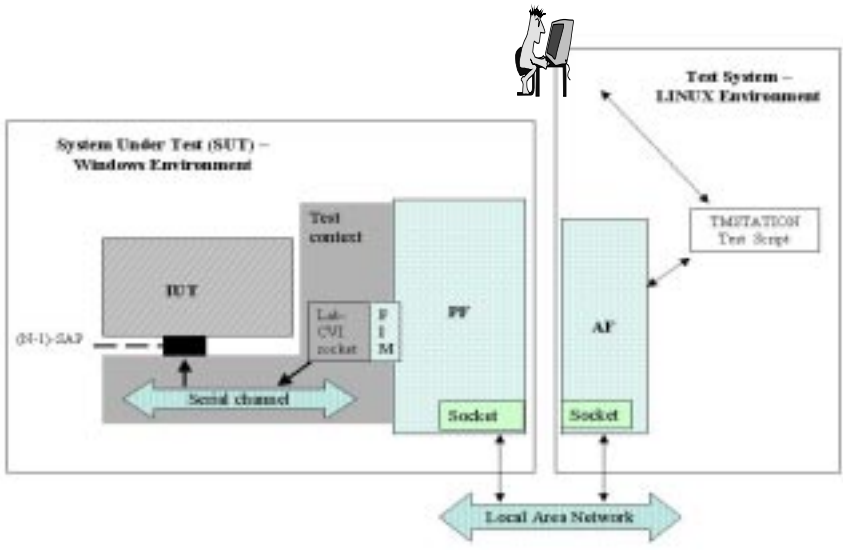


Fig. 4.2. FSoFIST configuration for TMSTATION application testing

A TELEMETRY.dat file, containing a sequence of valid frames, was used during the tests. Various scripts were developed: one for conformance testing, and the others for fault injection. The conformance testing script merely reads frames on the

TELEMETRY.dat file and passes them to the IUT through the ferry clips. For the fault injection tests, the scripts were generated according to the types of faults to be injected (fault cases). Faults were selected deterministically to meet the frame violations bellow:

(i) omission of words in the data field – to emulate the reception of truncated frames.

(ii) change of the EOFpattern - to emulate failure in the transmission of this pattern which results in extended frames.

Figure 4.3 shows an example of FSoFIST window during test execution. The upper part of this window shows the script used to inject faults according to (ii) above. In this script, the last three words of the EOF pattern are changed from (AA55) to (9A05), intermittently on the 2nd, 3rd and 4th frames of a sequence composed of five frames. The lower part of Figure 4.3 shows part of the log generated during the tests.

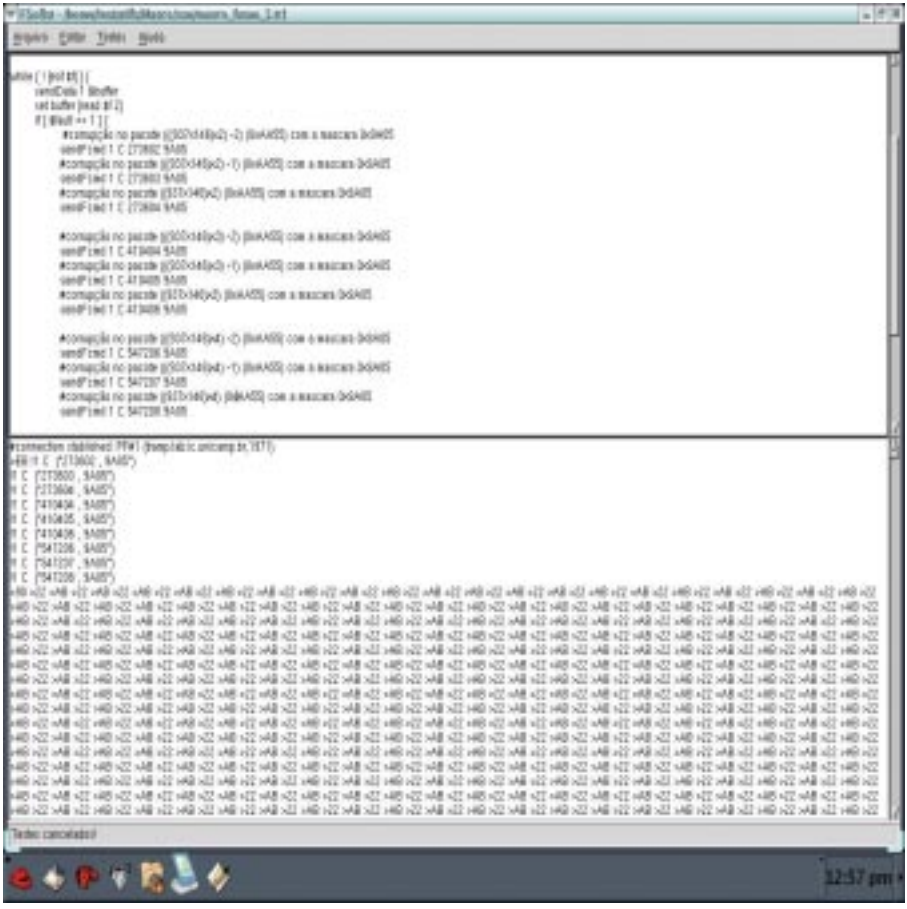


Fig. 4.3. Presents the script and test results associated to FSoFIST execution for extended frames test case.

Six scripts were created in order to validate fault injection capabilities of the FIM module. These scripts implement the prescribed error models according to (i) and (ii) above, and for each one we varied the repetition pattern (transient, intermittent and permanent). Table 4.4 describes them as well as the related results:

Table 4.4. Summary of the results of FSoFIST validation using TMSTATION software as case study.

Number of frames in input file	Error model	FSoFIST technique	Status of the stored frames (*)
6	Transient at 3rd frame	corruption at EOF pattern	1valid 2valid 3/4extended 5valid 6 valid
7	Intermittent at 2nd, 4th, 6th frames	corruption at EOF Pattern	1valid 2/3extended 4/5extended 6/7extended
5	Intermittent at 2nd, 3rd, 4th frames	omission of EOF pattern	1valid 2/3/4/5extended
3	Permanent	byte omission on the data field	1 truncated 2 truncated 3 truncated
5	Intermittent at 2nd, 4th, 5th frames	byte omission on the data field	1 valid 2 truncated 3 valid 4 truncated 5 truncated
4	Transient at 2nd, 3rd frames	byte omission on the data field	1 valid 2 truncated 3 truncated 4 valid

*Test Results: Status of the data files (frames) generated by TMSTATION

5 Related Works

Numerous approaches have been proposed for the validation of distributed systems. Our work is primarily related with studies in the areas of protocol conformance testing and fault injection. Section 2 presented the main aspects concerning test architectures used in these areas. Here we discuss some previous work.

From protocol conformance testing field we borrowed the ferry-clip approach to build our protocol and fault injection tool. The ferry-clip approach is not new. Zeng et al introduced the concept in 1985 [28], where the PF was intended to replace the UT in the System Under Test, and an enhanced UT was moved to the same machine as the LT. As a result, the amount of test code in the SUT is reduced, and also, the synchronization between the UT and LT is easier. In their approach both the test channel and the ferry channel pass through the IUT. Many refinements were applied to this architecture since then. In [29] it is proposed a reduction in the PF by removing the interface region from it. The interface region converts data to/from the IUT,

masking IUT dependent features from the testers. Part of these features were moved to a new module, called Service Interface Adapter, introduced in the Test System to convert data from to/from the upper IUT interface. Another part constitutes the encoder/decoder in the Test System, used to convert data to/from the IUT lower interface. Also, the ferry channel no longer passes through the IUT; it uses the underlying communication layer instead. Chanson et al proposed the Ferry Clip based Test System (FTCS) [5], where a Ferry Control Protocol was introduced to provide a standardized interface on top of an existing protocol which actually transfers the test data (the ferry transfer medium protocol). In this architecture, both interfaces of the IUT are controlled and observed by the PF. We borrowed this idea to build our ferry-injection tool. Besides the improvement on the control and observation of the IUT interfaces, this architecture also provides an independent communication channel for ferry connection. This allows the Ferry Transfer Medium Protocol to be as simple as possible, in order to reduce the complexity of the PF. In addition, this guarantees availability of the connection between the Test System and the System Under Test in case of crashes that can occur as consequence of fault injection.

Another related work is the one presented by A.W.Ulritch et al, where two test architectures for testing distributed systems were proposed: one based on a global tester that controls and observes all distributed components of a SUT in a central manner, another based on a distributed tester that consists of a number of distributed, concurrent tester components, each of them observing a partial behavior of the IUT [24]. The latter must provide a test coordination procedure to assure a consistent global view of the SUT, which comprises various IUT. A tool, TMT (from Test and Monitoring Tool) to support both architectures was implemented in Java. TMT can be applied to systems implemented in C++ (for Unix-like or Windows NT platforms) or Java. The distributed test components are implemented in a test library. Call for functions in this library must be inserted into the IUT source code. The source code is then necessary since TMT implements a grey-box testing approach, in which instrumentation is introduced inside the IUT to observe its internal interactions. The architecture of FSOFIST can also be easily extended to incorporate multiple ferry clips inside the SUT. Grey-box testing can then be used, for instance, for a multi-layer IUT, allowing the observation exchanges at various layer boundaries. However, the fact that FSOFIST does not take for granted that source code is available, its implementation is not dependent on any specific programming language.

Also closely related to our work is the approach called on-the-fly testing, which combines test derivation and execution in an integrated manner. Instead of deriving complete test cases (comprised of test events, each test event representing an IUT interaction), the test derivation process only derives the next test event from the specification and this test event is immediately executed [26]. In this case, besides the Driver, that controls and observes the IUT during testing, there is another component, the Primer, responsible for the generation of a valid input derived from the specification, as well as for checking whether the output generated by the IUT is valid according to the specification. TORX [23] is an example of a tool developed to support this approach. The tool also supports batch test derivation and execution. A drawback of this approach is that the Test System is dependent on the specification formalism,

since it is also responsible for test case derivation. In our case, this task can be performed either by another tool or by the user that can manually write her/his test cases using the editor available at FSoFIST interface.

The studies presented so far are aimed at conformance and interoperability testing, but do not consider fault injection.

Fault injection on distributed systems is a very active area. In the past, most common fault injection approaches were by hardware or through fault simulation. These approaches have been used even for software validation (e.g. [3; 8]). More recently, software-implemented fault injection is been used for that purpose. SFI [20] and its successor, Doctor [19], use different approaches to inject different types of faults. To inject communication faults, library routines are altered so as to provide erroneous communication services. EFA [11] introduces communication faults by inserting a fault injection layer in the protocol stack. This extra layer is introduced on top of the network link layer. Fault injection control is a program compiled into the fault injection layer. This program can be automatically generated from a formal model of the system. VirtualWire [10] also introduces fault injection and monitoring features on top of physical layer, more precisely, between the network interface card's device and the IP protocol stack. VirtualWire is aimed at testing any network protocol operating within a local area network. Fault injection control is programmed in a declarative language specially developed for that purpose. PFI and its successor, ORCHESTRA [9] use a fault injection layer between the IUT and the layer below on the network stack, as the previous works. But differently from EFA and VirtualWire, this layer moves around the protocol stack. It is based on the concept of probes, introduced to network monitoring purposes. Similar to FSoFIST, the experiments are programmed by the user in TCL and C. This work is rather close to the approach used in this paper. They also defined an abstract fault-injection architecture and develop a tool based on it. Their architecture aims at protocol-independence, although their tool is devoted to protocols that use sockets on Unix-like operating systems. Fault injection capabilities were introduced at the routines which comprise the socket interface provided by a library. The IUT source code is not modified, but it must be re-linked with the new library.

Loki [4], as ORCHESTRA, is another tool that uses probes monitoring as well as fault injection purposes. Probes are inserted in each node to monitor state changes and to inject faults as required. Fault injection on a node is triggered according to state changes information, either local or from remote nodes. Probes can be inserted into the source code, whether it is available. In case the source code is not available, monitoring and fault injection is performed from outside the IUT. The users can either select a probe among the pre-implemented ones, or can develop their own probes.

Similar to Loki, in FSoFIST the users can also develop their own PF, more specifically, the SUT-dependent part of the PF (LMAP and SIA/FIM modules), which gives the tool more flexibility and applicability. The user can choose a suitable location to the SIA/FIM modules according to her/his test purposes. These modules can also be implemented using probes, as in ORCHESTRA and Loki. For the moment we do not have pre-implemented modules as in Loki, but this is envisaged for the near future.

6 Conclusion and Future Work

This paper presents the ferry-injection architecture for conformance testing and fault injection of distributed systems. We also presents a tool, FSOFIST, that implements the proposed architecture. Some experiments were performed to validate the tool and demonstrate the capabilities of the ferry-injection architecture. Preliminary results of performance evaluation are also reported. A small and simple, but real-world application was used as case study. The advantages of the proposed architecture include protocol independence and portability to different platforms. The architecture is also ease to use, since test cases for conformance testing, as well as fault injection experiments, might be specified using user-defined scripts in TCL, whose syntax is very similar to the C language. Ongoing activities on this project are considering the following aspects: (i) use of other distributed applications in the aerospace field as case studies; (ii) extension of the ferry-injection architecture for multi-party and interoperability testing. In the long term we plan to use other commercial or prototype distributed applications to evaluate the tool's capabilities.

Acknowledgment. The authors would like to acknowledge the financial support from the National Research and Development Council (CNPq) of Brazil and the team of the Astrophysics Division at the National Institute of Space Research (INPE) headed by Dr. João Braga, responsible for the MASCO project. We also thank Anderson Nunes de Paiva Moraes for executing the experiments with FSoFIST, and Ana Maria Ambrósio for the technical review.

References

1. Araújo, M. R. R. FSoFIST- A Tool for Fault Tolerant Protocols Testing. MSc Dissertation. Institute of Computing – State University of Campinas (UNICAMP). (October/2000). (In Portuguese)
2. Arlat, J., Crouzet, Y., Laprie, J.-C., Fault Injection for Dependability Validation of Fault Tolerant Computing Systems. Proc Int'l Symposium on Fault-Tolerant Computing (FTCS-19), Chicago, USA, (1989)
3. Arlat, J., Aguera, M., Crouzet, Y., Fabre, J.-C., Martins, E., Powell, D. Experimental Evaluation of the Fault Tolerance of an Atomic Multicast System. IEEE Trans. Reliability, 39 (4), (October/1990), 455–467
4. Chandra, R., Lefever, R.M., Cukier, M., Sanders, W.H., Loki: a State-Driven Fault Injector for Distributed Systems. Proc. Int'l. Conference on Dependable Systems and Networks (DSN'00), New York, USA, (2000) 237–242
5. Chanson, S. T., Lee, B. P., Parakh, N. J., Zeng, H. X., Design and Implementation of a Ferry Clip Test System. Proc. 9th IFIP Symposium on Protocol Specification Testing & Verification, Enscchede, The Netherlands. (1989) 101–118
6. Chanson, S.T., Vuong, S., Dany, H., Multi-Party and Interoperability Testing using the Ferry Clip Approach. Computer Communications, 15 (3), (April/1992)
7. Clark, J. A., Pradhan, D. K., Fault Injection: A Method for Validating Computer System Dependability. IEEE Computer, (June/1995) 47–56

8. Czeck, E., Siewiorek, D., Effects of Transient Gate-Level Faults on Program Behavior. Proc Int'l Symposium on Fault-Tolerant Computing (FTCS-20), (1990) 236–243
9. Dawson, S., Jahanian, F., Mitton, T., ORCHESTRA: a Fault Injection Environment for Distributed Systems. Available on site: http://www.ecs.umich.edu.
10. De, P., Neogi, A., Chiueh, T-C., VirtualWire: A fault injection and analysis tool for network protocols. Proc. IEEE 23rd. International Conference oin Distributed Computing Systems, Providence, Rhode Island, USA (2003)
11. Echtle, K., Leu, M., The EFA Fault Injector for Fault-Tolerant Distributed System Testing. Proc. Workshop on Fault-Tolerant Parallel and Distributed Systems, Amherst, USA, (1992)
12. Holzmann, G.J. Design and Validation of Computer Protocols, Prentice Hall, (1991)
13. Hsueh, M., Tsai, T. K., Iyer, R. K., Fault Injection Techniques and Tools. IEEE Computer, (April/1997) 52–75
14. ISO TC97/SC21, IS 9646. OSI Conformance Testing Methodology and Framework, ISO/1991.
15. LabWINDOWS/CVI-C for Virtual Instrumentation-User Manual, *National Instruments Corporation Technical Publications*, 1996
16. Martins, E., Sabião, S.B., Ambrosio, A.M., ConData: a Tool for Automating Specification-based Test Case Generation for Communication Systems. Software Quality Journal, 8 (4), (1999) 303–319
17. Mattiello-Francisco, M.F. Software Requirements Specification for MASCO Telemetry Ground Reception. Internal Report MASCO-SRS-001, INPE, (05/2000)
18. Probert, R., Monkevic, O. , TTCN: The International Notation for Specifying Tests of Communication Systems. Computer Networks & ISDN Systems, 23 (1992) 111–126
19. Rosenberg, H.A., Shin, K.G., DOCTOR: an Integrated Software Fault Injection Environment. Technical Report, University of Michigan n° CSE-TR-192-93 (1993)
20. Rosenberg, H.A., Shin, K.G., Software Fault Injection and its Application in Distributed Systems. In Int'l Symposium on Fault-Tolerant Computing (FTCS-23), (1993), Toulouse, France
21. Segall, Z., Vrsalovic, D., Siewiorek, D.P., Yaskin, D., Kownacki, J., Barton, J., Dancey, R., Robinson, A., Lin, T., FIAT - Fault Injection Based Automated Testing Environment In Int'l Symposium on Fault-Tolerant Computing (FTCS-18), Tokyo, Japan, (1988) 102–107.
22. Stefani, M.R., Trace Analysis and Diagnosis Generation for Testing the Behavior of a Communication Protocol in the Presence of Faults. MSc. Dissertation, Institute of Computing, State University of Campinas, (May/1997)
23. Tretmans, J., Belinfante, A., Automatic Testing with Formal Methods. Proc. 7th. European Conference on Software Testing, Analysis and Review. EuroSTAR´99, (November, 1999)
24. Ulrich, A W., Zimmerer, P., Chrobok-Diening, G., Test Architectures for Testing Distributed Systems. Proc. of Software Quality Week, (1999)
25. Villela, T., Braga, J., Mejá, J., D'Amico, F., Alves, A., Silva, E., Rinke, E., Fernandes, J., Corrêa, R., Preflight Tests of the MASCO Telescope, Advances in Space Research, 26(9), (2000) 1411–1414
26. Vries, R.G., Tretmans, J., On-the-fly Conformance Testing using SPIN. In G.Holzmann, E.Najm, A.Serhrouchni, (ed.), 4th. Workshop on Automata Theoretic Verification with the SPIN Model Checker, Paris, France, (November/1998) 115–128
27. Walter, T., Shieferdecker, I., Grabowski, J., Test Architectures for Distributed Systems – State of the Art and Beyond. In, Testing of Communicating Systems, 11, A. Petrenko, N.Yevtuschenko (ed.), Kluwer Academic Publishers, (September/1998). Obtained in 2002 at URL: http://citeseer.nj.nec.com/walter98test.html

28. Zeng, H.X., Rayner, D., The Impact of the Ferry Concept on Protocol Testing. In Proc. V Protocol Specification, Testing and Verification, (1986) 533–544
29. Zeng, H.X., Li, Q., Du, X.F., He, C.S., New Advances in Ferry Testing Approaches. Computer Networks and ISDN Systems, 15 (1988) 47–54

Appendix

AF	Active Ferry	PCO	Point of Control and Observation
ASP	Abstract Service Primitive	PDU	Protocol Data Unit
CTMF	Conformance Testing Methodology and Framework	PF	Passive Ferry
FCP	Ferry Control Protocol	SAP	Service Access Point
FE	Ferry Engine	SIA	Service Interface Adapter
FIC	Fault Injection Controller	SUT	System Under Test
FIM	Fault Injection Manager	TCP	Test Coordination Procedure
FTMP	Ferry Transfer Medium Protocol	TM	Test Manager
ISO	International Standard Organization	TMP	Test Management Protocol
IUT	Implementation Under Test	TS	Test System
LMAP	Lower Mapping Module	TSC	Test Sequence Controller
LT	Lower Tester	UT	Upper Tester

A Fault-Tolerant Distributed Legacy-Based System and Its Evaluation

Andrea Bondavalli[1], Silvano Chiaradonna[2], Domenico Cotroneo[3], and Luigi Romano[3]

[1] Dipartimento di Sistemi e Informatica, Universitá di Firenze,
Via Lombroso 6/17, 50134 Firenze, Italy
a.bondavalli@dsi.unifi.it
[2] ISTI-CNR
Via A. Moruzzi 1, Loc. S. Cataldo, 56124 Pisa, Italy
Silvano.Chiaradonna@cnuce.cnr.it
[3] Dipartimento di Informatica e Sistemistica Universitá di Napoli Federico II
Via Claudio 21, 80125 Napoli, Italy
{cotroneo, lrom}@unina.it

Abstract. In this paper, we present a complete architecture for improving the dependability of complex COTS and legacy-based systems. For long-lived applications, such as most of those being constructed nowadays via integration of legacy subsystems, fault treatment is a very important part of the fault tolerance strategy. The paper advocates the need for careful diagnosis and damage assessment, and for precise and effective recovery actions, specifically tailored to the affecting fault and/or to the extent of the damage in the affected component. In our proposal, threshold-based mechanisms are exploited to trigger alternative actions. The design and implementation of the resulting solution is illustrated with respect to a case study. This consists of a distributed architectural framework, handling replicated legacy-based subsystems. Replication and voting are used for error detection and masking. An experimental prototype deployed over a COTS-based LAN is described and has allowed a dependability analysis, via combined use of direct measurements and analytical modeling.

Keywords: Software Implemented Fault Tolerance, Fault Diagnosis, Fault Treatment, Legacy systems, Threshold-based mechanisms, CORBA Architectures

1 Introduction

We are witnessing the construction of complex distributed systems, which are the result of the integration of a large number of components, including COTS (Commercial Off-The-Shelf) and legacy systems. The resulting systems are being used to provide services, which have become critical in our everyday life. It is thus paramount that such systems be able to survive failures of individual components, as well as attacks and intrusions, i.e. they must provide some level

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 303–320, 2003.

of (reduced) functionality also in the event of faults. The integration of COTS components and legacy subsystems in a wider infrastructure is a challenging task, for a variety of reasons, and in particular:

- COTS components are relatively unstable and unreliable [3]. Nevertheless, to reduce system development time, the use of unmodified COTS components is mandatory in modern distributed systems;
- Legacy systems were designed as disjoint components and they were not supposed to interoperate. As a consequence, the integration environment might stimulate legacy components in unexpected ways, thus leading to potential failures. This is a well-known problem to integration test professionals. Indeed, the task of the integration test is to check that components or software applications, e.g. components in a software system or - one step up - software applications at the company level - interact without error. Many examples from field experiences can be found at `http://www.sqs.de/english/casestudies/index.htm`.

For these reasons, without effective architectural solutions, survivable infrastructures consisting of COTS and legacy-based applications, are virtually impossible to obtain. In order to be effective, fault tolerance strategies in such systems need to be revisited. More precisely, mechanisms and strategies to implement fault tolerance functions have to be tuned, to account for the many differences between COTS and legacy-based systems and traditional safety-critical systems. Recently some proposals have been made, which allow to build dependable systems by integrating COTS and legacy components [4], [5], [6], [7], [8], [9], [10]. These proposals mainly concentrate on error processing, either by NMR-style replication or by organizing distributed membership of replicas. However, little attention has been paid to the problem of maintaining system health and preserving tolerance capabilities.

This paper focuses on diagnosis and fault treatment and proposes their integration with mechanisms for error detection and processing. Fault treatment consists of fault diagnosis, and recovery/reconfiguration [11]. Although diagnosis has been extensively studied, distributed COTS and large legacy-based systems raise a variety of issues which have not been addressed before. Such issues stem from a number of factors, which are briefly described in the following. First, the designer (system integrator) has limited knowledge and control over the system as a whole, as well as over individual components, since for most components and subsystems the internal design is not known. Second, COTS and legacy components are heterogeneous, whereas the targets of traditional diagnosis are – to a large extent – homogeneous. Third, diagnostic activities must be conducted with respect to components which are large grained, whereas traditional applications (such as safety critical control systems) typically consist of relatively fine grained components. It is thus not practical (or possible at all), as soon as an error is observed, to declare the entire component failed and proceed to repair and replacement (repair implies at least very costly recovery and re-integration, replacement may not be possible at all).

For the above discussed reasons, in a COTS and legacy-based infrastructure diagnosis must be able to assess the status or the extent of the damage in individual components, so to carefully identify the most appropriate fault treatment and system reconfiguration actions and when they have to be applied. To this aim, it is foremost that data about error symptoms and failure modes be carefully gathered and processed. One shot-diagnosis is thus inadequate: an approach is needed, which collects streams of data and filters them by observing component behavior over time. Several heuristics based on the notion of threshold have been proposed and have effectively been applied to many fields (e.g. diagnosis and telecommunications). All these mechanisms take their decisions based on the analysis of streams of data, consisting of sequences of observations.

We present a complete architecture for improving the dependability of complex COTS and legacy-based systems. We emphasize aspects related to fault treatment. We had to face several key issues, and in particular:

- Limiting the probability of common mode failures - The replicated legacy systems were identical and thus they could exhibit common mode failures. Since we could not modify the legacy systems, we enforced diversity at different architectural levels when we integrated the legacy systems in our replication framework. More details about the specific measures that we have taken are provided in subsection 3.2.
- Limiting the probability of interaction faults - These are typically caused by mismatches or incompatibilities between the legacy applications and the COTS platforms and software components. Research has demonstrated that in order for an error detection process to work properly, data from individual replicas must be conditioned before comparisons are made [19].

We detail the description of the architectural solutions we have implemented with respect to a case study. We also evaluate the effectiveness of the suggested approach via combined use of direct measurements on the system prototype and analytical modeling. We use a real prototype to populate the model with realistic parameter values. The rest of the paper is organized as follows. Section 2 introduces previous relevant work and illustrates the system we use as a case study. Section 3 details our complete fault tolerance strategy highlighting fault treatment and describes our implementation. Section 4 describes our approach to the analysis, and the models we built. Section 5 reports the results we obtained so far, while section 6 concludes the paper.

2 Preliminaries

2.1 Background

A great deal of research has been conducted on providing support for dependability to existing applications via distributed architectural frameworks. These projects differ from one another under many aspects, including the nature of the fault tolerance mechanisms (hardware, software, or a combination of the two),

and the level of transparency to the application level (application aware/unaware approach). Among the others, it is worth mentioning [6], [7], [9], [10], [13], as well as the FT CORBA initiative [14]. All these projects focus on error processing, but they address fault-treatment only to a limited extent. There are also several commercial products which claim to incorporate fault tolerant facilities (such as J2EE [2] and Microsoft .NET [1], to name a few). As to our personal experience, in [9] we presented a middle-tier based architectural framework for leveraging the dependability of legacy applications in a way transparent to the clients. Such a framework can be deployed on top of any CORBA infrastructure. The fault tolerance mechanisms used include error detection, different forms of error processing, graceful degradation, and re-integration. A system prototype was developed and tested over a distributed heterogeneous platform. A preliminary analysis of such a prototype clearly indicated that a more effective fault treatment support was needed to significantly improve the dependability level attained by the system. Thus, we started investigating diagnosis in COTS- and legacy-based applications. More precisely, we addressed issues related to goals and constraints of a diagnostic sub-system based on the concept of threshold, which must be able to *i*) understand the nature of errors occurring in the system, *ii*) judge whether and when some action is necessary, and *iii*) trigger the recovery/repair mechanisms/staff to perform the adequate actions to maintain the system in good health [15]. This led to the definition of a diagnostic subsystem scheme which consists of two distinct components. One, the Threshold-Based Mechanism (TBM), implements a threshold-based algorithm, the other, the Data Processing & Exchange (DPE), is in charge of conveying to the former the data from the detection sub-system, possibly performing some processing. The two blocks (TBM and DPE) might well be distributed in actual implementations of the system. Typically, DPE functions would be co-located with the error detection sub-system. The specific algorithm implemented by the TBM is called alpha-count, and is described in [12]. Among the heuristics based on the concept of threshold, the alpha-count family of mechanisms appears to be particularly interesting for our purposes due to the clear and simple mathematical characterization and to the thorough analysis already conducted. In the following, we briefly describe how the alpha-count mechanism works. The alpha-count processes information about erroneous behavior of each system component, giving a smaller and smaller weight to error signals as they get older. A score variable α_i is associated to each not-yet-removed component i to record information about the errors experienced by that component. α_i is initially set to 0, and accounts for the L-th judgement as follows:

- $\alpha_i(L) = \alpha_i(L-1) + 1$ *if channel i is perceived as faulty during execution L*
- $\alpha_i(L) = K * \alpha_i(L-1)$ $(0 < K < 1)$ *if channel i is perceived as correct during execution L*

When $\alpha_i(L)$ becomes greater than or equal to a given threshold α_T, component i is diagnosed as failed and a signal is raised to trigger further actions (error processing or fault treatment). The effectiveness of the mechanism depends on the parameters K and α_T. The optimal tuning of these parameters depend on the

expected frequency of errors and on the probability c of correct judgements of the error signaling mechanism. The analysis performed in [12] has clearly shown the trade-offs between delay and accuracy of the diagnosis and thoroughly discussed ways to tune these parameters for maximizing performance.

2.2 The Application Used as a Case Study

By legacy application, we mean a software program for which re-engineering or maintenance actions are either impossible or prohibitively costly. The application used as a case study is a multi-tier application, consisting of legacy code, written in C, which uses a COTS DBMS, namely PostgreSQL for stable storage facilities. The application runs on a Unix-like kernel, on top of a commodity PC or a workstation. The data base physical files are stored on disk. Services can be grouped in two main categories, namely *Queries* and *Updates*. The legacy applications reads data from the database to the program dynamic memory. The application allocates new memory when it needs some. Records are stored in a linked list like data structure. The DBMS also caches data.

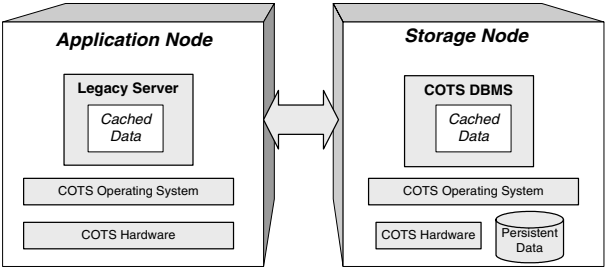


Fig. 1. Case-study legacy application

Since we could introduce a satisfactory degree of diversity (more details about this in section 3.2) in the running environments of our legacy subsystems to make the probability of common mode failures very low, our objective is to contrast the effects of independent faults hitting the back-end servers. The application was made more dependable by replicating the legacy application according to a TMR scheme. In particular we consider the following type of faults:

- Hardware-induced faults - These are faults stemming from instabilities of the underlying hardware platform [16]. We limit our attention to intermittent faults, since these are by far the predominant cause of system errors [17];
- Software errors - These are faults related to flaws in the application and/or in the software infrastructure. Examples are errors in the design and/or implementation process of the application and/or of the Operating System, or to process and/or host crashes or hangs due to inconsistent application

and/or OS level state, or to unavailability of resources due to overload conditions. We distinguish between "naive" software faults, such as trivial bugs, and "'subtle"' software faults, such as memory leaking problems in the application or system resource exhaustion. We limit our attention to the latter kind of faults, since we assume that the former kind of faults has been detected and fixed during the years long operation of the legacy system. This is not a simplistic assumption. In fact, in the typical scenario, legacy software has been thoroughly tested and debugged, and the vast majority of naive bugs has thus been detected and fixed over the years. However, more subtle software errors may still be present in the code base. As an example, errors related to incorrect use of memory may well have gone undetected (it is likely that the programmer has requested the allocation of a certain amount of memory, but he has failed to deallocate it). As a result, the legacy program may exhibit memory leakage problems. A memory leak can go undetected for years if the application and/or the system is restarted relatively often (which might well have been the case of the legacy application). However, if the new deployment scenario requires that the legacy system be launched as a long running application, these subtle problems would eventually manifest.

– Faults in the physical data-base - Corrupted data has been written to the system stable storage.

We do not consider communication errors. Such errors occur when data gets corrupted while traversing the network infrastructure. Indeed, unless "leaky" communication protocols are adopted, this kind of errors is fairly unlikely to happen (a leaky protocol is a protocol that allows corrupted information to be delivered to the destination).

The above discussed faults manifest as failures of the legacy application as follows. Hardware faults may corrupt the linked list structure of the application cache. This may happen in two ways, which result in errors of different severity. A first problem occurs when a data item in the list gets corrupted. This is the less severe kind of error, since at the application level it results in a corrupted entry being written to the database or exposed to the system external interface. A second problem occurs when a pointer gets corrupted. This is a more severe error, since it manifests as a multiple error. One possibility is that the faulty pointer points to a null value. At the application level, this error results in a truncated list being written to the physical database (i.e. in possibly several items being deleted from the database). Another possibility is that the faulty pointer points to a wrong item in the list. This error would result in possibly several items being added/deleted to/from the database. Yet another possibility is that the the faulty pointer points to an invalid address. This error typically results in a signal being generated by the Operating System and in the application being killed.

Software faults may lead to resource exhaustion. As an example, memory leaks may result in a huge amount of memory being allocated to the faulty application. At first, this would result in an overload condition for the hosting node, with potential performance faults which would manifest as timeout failures

of the application. Eventually, the Operating System would deny the allocation of further resources to the application. Again, this would typically result in a signal being generated by the Operating System and in the application being aborted.

3 Fault Tolerance and System Prototype

In this section, we describe the complete fault tolerance strategy that we propose. In particular, we highlight the fault treatment logic (i.e. alpha count mechanisms, and data exchange techniques) and describe its implementation applied to our case study (i.e. the architecture of the distributed legacy based system and the roles played by individual components).

3.1 Fault Tolerance Approach

Our architecture is based on a middle-tier to manage three replicas of the legacy database and COTS DBMS running on the back-end nodes (channels), used in a TMR fashion. Thus, a voter in the middle tier assures masking of one replica failure and conveys to our diagnostic subsystem information on errors detected. As we already mentioned, excluding replicas or performing heavy and costly recovery actions, at the very first occurrence of an error is too a simplistic approach, which may well have a negative impact on the dependability of the overall system. Thus, the middle-tier is also in charge of diagnosing the status of the channels, and of deciding i) when to perform recovery actions, and ii) which recovery action is the most appropriate. Based on the fault assumptions made, we identified three recovery actions for the legacy subsystems:

1. Restart of the application - This action can fix inconsistent application level states, but it is inefficient against errors in the system stable storage;
2. Reboot of the host computer - This action restarts the operating system and all the service software; it can thus fix inconsistent OS states as well as service software states;
3. Restoration of the data base - This action aims at correcting errors in the physical data stored on disks. Restoration is conducted as follows: the recovery manager reads binary data from the two other replicas and compares the flows. If comparison is successful, bits are copied to the recovering instance. If a disagreement is detected, a third value is read from the recovering replica. Hopefully, it is possible to determine the correct value via majority voting. If this is not the case, we assume the system has failed, since no valid data is available.

The restoration procedure that we have described is of course just one of many possible algorithms, and we do not claim it is the best alternative (the focus of this paper is not on database restoration techniques). Two instances of alpha-count monitor each channel. These are used to choose among the three alternative recovery actions. Fault-treatment logic is as follows. Any error detected

by the voter triggers the first recovery action (application restart) and is sent to the first alpha-count which increases the score, whereas each success is used to decrement the score. When the threshold is reached, recovery action number two (host restart) is executed, the score is reset to zero and an error signal is sent to the second alpha-count, which increments its score. The score of the second alpha-count is never decremented (assuming that normal operation does not correct corrupted data). Finally, when the threshold in the second alpha-count is reached, the third recovery action is performed (database restoration) and the scores of both alpha-counts are reset to zero.

3.2 Prototype

The overall organization of the system prototype is depicted in Figure 2 and its components are described in the following. This architecture consists of a three-tier system. The first tier consists of a Client which uses the services provided by the third tier, consisting of a replicated COTS and Legacy-based application (channel).

Since the final effects of faults (i.e. the application failure modes) depend on several factors, which include the specific characteristics of the computing node which hosts the application, when we integrated the legacy system replicas of individual channels in the TMR architectural framework, we enforced diversity at various architectural levels of the deployment environment. By doing so, we were able to lower the occurrence of common mode failures. More precisely, the nodes in the third tier run different versions of the Linux Kernel (2.2, 2.4.18, and 2.4.19), have different amounts of RAM (0.5 GB, 1 GB, and 256 MB), and are configured so to launch a different set of services at startup.

The second tier mediates the service by hiding replication to the users and providing proper management facilities (distribution of service requests, voting upon individual results, and redundancy management). In particular, the Middle Tier handles updates by broadcasting them to all active replicas. The connectivity between the individual components is provided by CORBA [14], specifically, VisiBroker version 4.1.

Gateway – In order to replicate the COTS-based application, we had to wrap it with a multicast enabled component. To this goal, we integrated in the Gateway the reliable multicast support provided by the Ensemble group communication facility [23]. We had to attach Ensemble to the `Gateway` object (at one end) and to the server replicas (at the other end). The former task was straightforward: we just had to link the Ensemble library to the Gateway code. The latter task was quite more complex. In fact, the legacy application came with a TCP/IP socket based interface. We had to intercept TCP calls, and redirect them to Ensemble. In order to do so, we developed a virtual device driver within the kernel of the node which hosted the legacy server. For a thorough description of this technique, please refer to [24]. The Gateway is in charge of all data exchanges between the Voter and the specific COTS-based application. This entails addressing all synchronization-related and format-related issues, since research has demonstrated that, in order for the voting process to work properly, data

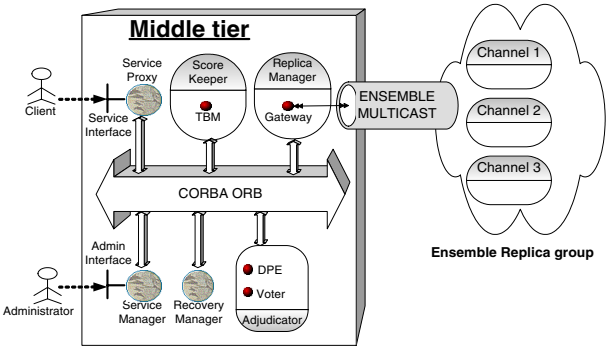


Fig. 2. Overall System Architecture

must be conditioned before comparisons are made [19]. The Gateway purifies the data from application-specific and platform-related dependencies, thus avoiding that system failures occur due to interaction problems.

Service Manager – The Service Manager component is in charge of configuring all other components. It provides functions to customize the behaviour of individual objects (such as the specific adjudication strategy which must be used for building the reply to be sent to the client), and to set system configuration parameters (such as the number of threads in the thread pools). System configuration is performed via the Admin Interface.

Service Proxy – The Service Proxy encapsulates the services provided by the legacy application and exports them via the Service Interface, thus making such (enhanced) services available to the clients.

Adjudicator – The Adjudicator component incorporates both the voter and the DPE. The voter is in charge of selecting a presumably correct result out of those provided by the channels. It may support several adjudication strategies but here it is configured to perform TMR-based majority voting. Based on the results of voting, the adjudicator provides error detection information to the DPE. The DPE is in charge of packing and delivering error detection data to the threshold based mechanisms (TBM) component.

Score Keeper – The Score Keeper computes the scores. It updates the alpha-count pair which is associated to each channel. It receives data produced by the Voter and filtered by the DPE.

Recovery Manager – The Recovery Manager is in charge of performing the recovery actions triggered by the Score Keeper.

4 Dependability Analysis

In order to analyze and evaluate our proposal and to tune the relevant parameters for the alpha-counts, we adopted an approach based on combined use of modelling and prototype-based measurements. This approach appears as the most promising one for large complex systems [20].

4.1 Prototype Settings

Fault injection experiments and performance measurements were performed on the prototype to populate the analytic model with realistic parameter values. Since our focus is faults which affect the channels, both the network and the CORBA infrastructure and services were considered reliable. Thus, we only injected faults to the channels. Fault injection was conducted using the NFTAPE tool [18]. The details of the fault injection campaign are not discussed here, due to lack of space. We used different machines for the channels and different workloads. From the analysis of the experimental data, the time needed to perform the recovery actions and to service a request, summarized in Table 1, were derived.

Table 1. Parameter values obtained from measurements on the prototype [sec]

Parameter	Description	Range
T_1	Time to restart the application	$0.1 - 0.5$
T_2	Time to restart the machine	$130 - 460$
T_3	Time to reconstruct the database (1GB)	$1000 - 3000$
T_4	Time to serve a request (at sustained rate)	$0.05 - 0.15$

4.2 System Models

The system has been modeled using Stochastic Activity Networks (SAN) [22]. We did not develop a detailed model of the channels,i.e., a model consisting as a large set of fine-grain components (e.g. processes, data structures, OS layer, HW layer, etc.), since this would have resulted in an unnecessarily large size for the model. Instead, we considered a channel as a relatively simple component, consisting of an application object, an OS component, and a database object, as detailed later in this section. Also, instead of modeling the whole variety of faults that we have discussed in section 2.2, we considered fault effects as they manifest as application failures. Conceptually, this is equivalent to using a fault dictionary to abstract from the component level to the application level. We assume intermittent application failures with an increasing failure rate according to a lognormal distribution [21] , since this hypothesis is consistent with the fact that the extent of the damage of the channels increases with time (if no recovery action is taken). The channel failure modes which we considered are:

- Timing Errors - The channel returns no value (before the Voter timeout expires);
- Value Errors - The channel returns a wrong value. More precisely: *i*) it returns a value different from what is actually stored in the physical database;

ii) it stores to the physical database a value different from the input ; *iii*) it does not perform the requested operation.

The composed model of Figure 3 represents the hierarchical model of the system behavior. It consists of ten logically separate SANs (Recovery, Client, GTDBServer1, GTDBServer2, GTDBServer3, FailModelDBServer1, FailModelDBServer2, FailModelDBServer3, ResultProviderVoter, and DiagnosticBlock), joined through common places by the Join1 operation. The SAN Recovery mimics system behavior as different types of recovery actions are taken. During recovery, the system does not serve requests. The SAN 'Client' represents the service requests, the status of the replies to the clients (correct, detected erroneous, undetected erroneous) and the number of replicated servers which are still online. The SANs GTDBServer1, GTDBServer2, GTDBServer3 represent the

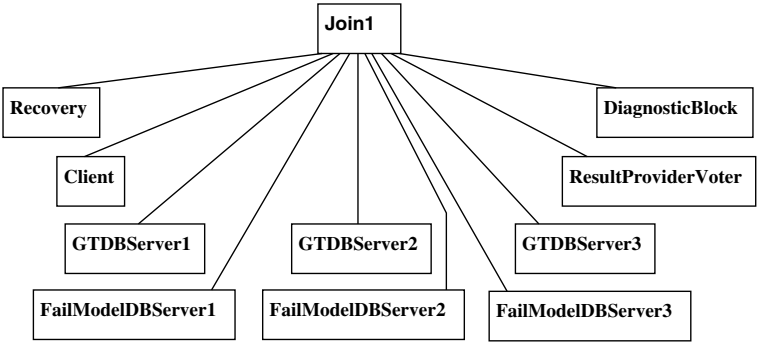


Fig. 3. Composed Model of the System

three replicas of the DB server and the GatewayThread processes (used to parallelize the activity of the Gateway. The SANs FailModelDBServer1, FailModelDBServer2, FailModelDBServer3 represent the failure behaviour of each channel. The SAN ResultProviderVoter represents the time and the actions of the ResultProvider (which receives the result sets from the GatewayThreads and delivers them to the Voter), and the Voter. The SAN DiagnosticBlock represents the behavior of the TBM.

For the sake of brevity, in the following only two of the sub-models are described in detail. Figure 4 depicts the SAN 'Recovery'. The activity Recovery represents the deterministically distributed time of recovery, depending on the type of recovery action. For example, the time for reconstructing the database depends on the number of records in the database, represented by the number of tokens in the place nRecords1, nRecords2, nRecords3. The activity Recovery is enabled by the DiagnosticBlock (which triggers the recovery by putting a token in the place recovery) and the Client models (which remove a token by the token busyServer when the current request has been served). The C code in the output gate Recovered enables the marking changes due to the restoration

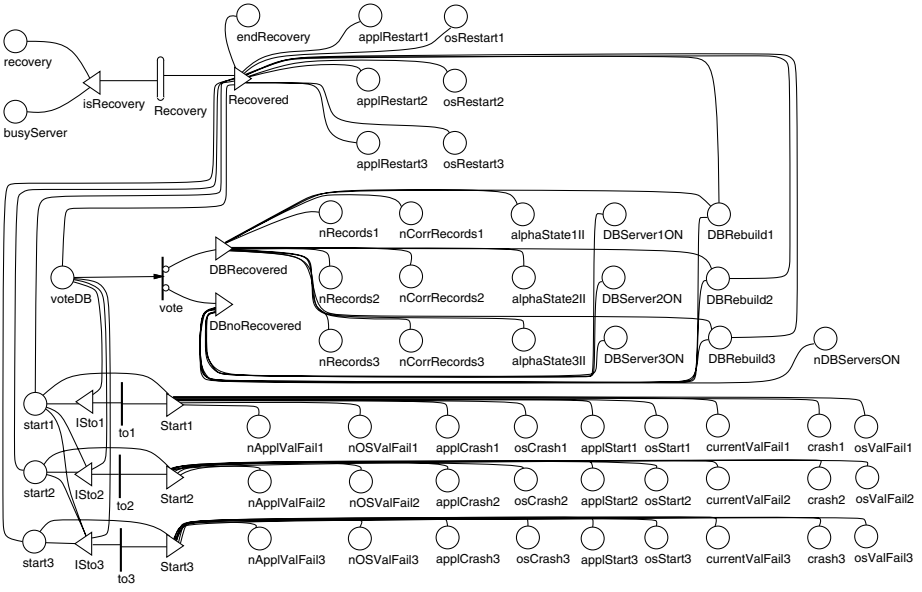


Fig. 4. SAN 'Recovery'

of the DataBase or due to the restarting of the application and the OS of the three channels (after restoration all the OS of the channels are restarted). The output gates DBRecovered and DBnoRecovered represent the marking changes for modeling the success or the failure of the restoring of the DataBase, respectively. The output gates Start1, Start2, Start3 represent the marking changes for modeling the restarting of the application and the OS of the three DB servers. Figure 5 depicts the SAN 'FailModelDBServer1'. The Lognormal (or Weibull) distributed activities ApplFail and OSFail represent the times to failure of the application and of the OS, respectively. The two associated cases represent the probability of crash (case 1) and of value failure (case 2). After the i-th failure, the time to next failure is reduced by using a distribution with a mean equal to the original one divided by i. ApplFail and OSFail are restarted with the original distributions after each restart of the application or of the OS, respectively.

5 System Evaluation

This section describes the results obtained so far on parameter tuning and overall system dependability via combined use of direct measurements on the system prototype and analytical modeling. Table 2 reports the main parameters of the SAN model and their default values. These, together with the ones reported in Table 1, which were extracted from direct measurements, were used to populate the analytical model. For all parameters, the higher extreme of the range measured has been used as the reference value for that parameter in the evaluation

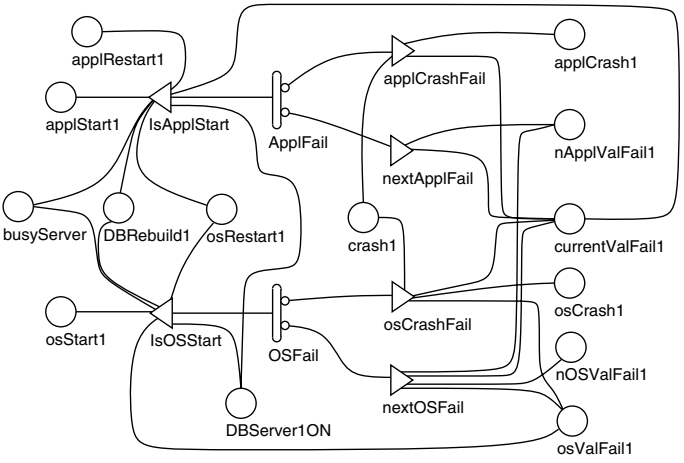


Fig. 5. SAN 'FailModelDBServer1'

of the SAN model. Our analysis consists of two main parts. The first part shows how to derive indications about tuning the parameters of the alpha-counts in order to reach good performance of the fault treatment strategy. The second part reports some measures of system availability obtained using the SAN model.

5.1 Tuning of the Parameters

A good performance of the system can be reached if one properly understands how frequently and under which system conditions the restoration procedure should be scheduled. The procedure is triggered by the second alpha-count and since the records can be corrupted, but cannot be corrected by a service request, it must be K_{II}=1. The recovery follows a majority voting approach and, if the database is not correctly recovered (upon completion there are still erroneous records) the system halts with a failure. The probability that this recovery procedure reaches its goals, i.e., that a correct version of the database can be reconstructed is evaluated as a function of the amount of corrupted records existing in the three replicas. We assume: *i*) a uniform distribution of erroneous data, *ii*) that corrupted replicas of the same record are perceived as different, and *iii*) (conservatively) that all the replicas contain the same number of corrupted records. Under these assumptions, a good approximation of the probability p_k that there are "k" erroneous records after executing the recovery procedure is obtained using a binomial distribution:

$p_k = \binom{N}{k} q^k (1-q)^{N-k}$, where:
$q = q_1q_2q_3 + q_1q_2(1-q_3) + q_1(1-q_2)q_3 + (1-q_1)q_2q_3$
$q_i = \frac{N_i^e}{N}$
N_i^e=number of erroneus records of the Channel_i

Table 2. Parameters and their default values used in the evaluation

Parameter	Default value
KI (1st alpha-count)	0.99
TI (1st alpha-count)	2, 3
KII (2nd alpha-count)	1
TII (2nd alpha-count)	3
Probability of Application Crash	0.75
Probability of OS Crash	0.75
N: nr. of records in the Database	10000
Request rate $[s^{-1}]$	0.005
Probability of an Update Request	0.2
Probability of a Query Request	0.8
μ_A, μ_O (par. of the Lognormal)	14
α_A, α_O (par. of the Lognormal)	0.6
mean (of the Lognormal) [days]	16.7
variance (of the Lognormal) [days]	12651129

Figure 6(a) plots p_0, i.e. the probability of correctly restoring all the records of the database, for $N = 10000$. Having in mind the desired probability of successfully restoring one replica of the database, one can now relate the number of corrupted records to the threshold of the second alpha-count which is used to trigger the recovery procedure. Figure 6(b) reports the number of erroneous records of a replica of the database as a function of T_{II} as estimated with our SAN model of the system. Figure 6(b) shows that, in the settings chosen, the number of corrupted records is always very low ranging from about 2 for low values of T_{II} to 5 when TII goes to 10. In more details, for the same T_{II} the

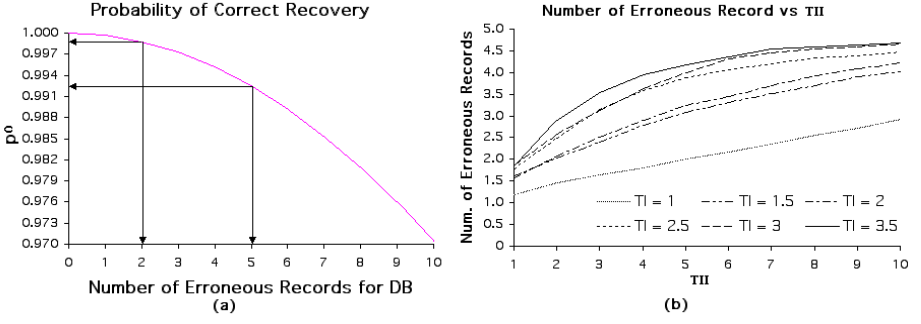


Fig. 6. (a) Probability of success of the recovery procedure as a function of the number of corrupted records, (b)Number of corrupted records as a function of the T_{II} of the second alpha-count for different values of T_I (KI=0.995)

higher is TI the higher the number of corrupted records. Figure 6(a) shows that in the range 2-5 of the number of corrupted records the probability of correct recovery varies from .999 to .992. However this should not be a concern: chances of correct recovery are higher than .99.

5.2 Availability

We considered a time frame of one year, and we measured the cumulated time in which the system is available for providing services i.e., the availability. Due to our will to account for many non exponential events (e.g., the failure process modeled as a lognormal and the duration of the recovery procedures modeled as constants), the SAN models were solved by simulation with UltraSAN [22]. Figure 7 reports the expected availability of the system in a one year period as a function of the value of the parameter KI of the first Alpha-count for several values of T_I the threshold of the first alpha-count. T_{II} the threshold of the second alpha-count is fixed to 3. The availability is computed accounting both for the time spent in doing recovery and for a potential complete failure of the system.

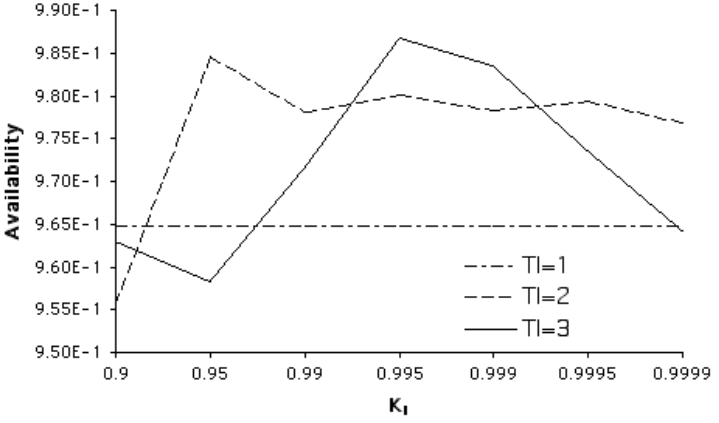


Fig. 7. Expected availability of the system in one year, as a function of KI the parameter of the first alpha-count for several values of TI (TII=3)

The figure shows that setting T_I =1 gives a constant availability: in this case the value of K_I does not play any role. Instead when T_I assumes values greater than 1 the impact of K_I on the availability can be noticed. Given a value for T_I one can notice that for increasing values of K_I the availability first increases until it reaches a maximum and then becomes worse. The utility of the first alpha-count and of the distinction between the first two recovery actions can be noticed by considering that the best combination of T_I and K_I gives better availability than the case with T_I =1.

Figure 8 reports the expected unavailability of the system in a one year period as a function of the value of the second threshold T_{II} for three different values (1, 2 and 3) of T_I the first threshold and K_I fixed to 0.99. The unavailability is computed accounting for both the time spent in doing recovery and the outage due to complete failure of the system.

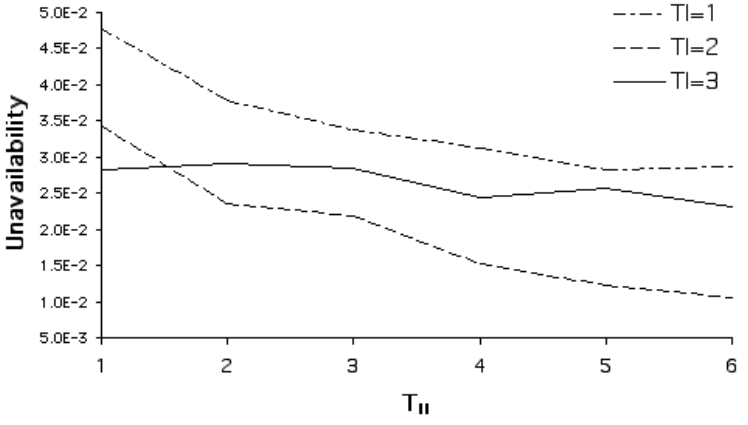


Fig. 8. Expected availability of the system in one year, as a function of TII (the threshold of the second alpha-count) for TI=1, 2 and 3 assuming KI=0.99

The point T_{II}=1, T_I=1 shows the unavailability of the system without our fault treatment mechanisms: all the recovery actions are performed at each error detection. The figure shows that the system improves for growing values of T_{II}. The best availability is obtained with $T_I = 2$. In this scenario setting $T_I = 3$ appears to be not very rewarding, the unavailability remains approximately constant, probably because too many errors affect the database before restoration is performed. Overall the figure shows that with a good tuning of the parameters using our fault treatment strategy one can reduce 5 times the unavailability from .05 to .01.

6 Conclusions and Future Work

This paper described a fault-tolerant distributed legacy-based system which has been implemented as a middle-tier based architectural framework to leverage the dependability of an existing application. The application used as a case study consists of legacy code, written in C, which uses a COTS DBMS, for persistent storage facilities. The application runs on Linux, on top of a commodity personal computer. Three different kinds of faults in the application have been considered: i) hardware-induced software errors in the application; ii) software errors (i.e., process and/or host crashes or hangs); and iii) errors in the physical database (i.e., corrupted data in the system stable storage). The system is organized

as a TMR and failure data collected by the voter is provided to the diagnostic subsystem, consisting of a filtering unit and a threshold-based component (TBM). The TBM uses filtered failure data to update three alpha-count pairs, one for each replica of the back-end servers. The two alpha-counts composing individual pairs are used to discriminate among alternative recovery actions. We considered three possible recovery actions: Action 1 (application restart), Action 2 (host restart), and Action 3 (data base restoration). In order to evaluate our system i.e., the effectiveness of the proposed fault-treatment strategy and to tune the parameters of its mechanisms, we developed a SAN model of the overall system consisting of ten sub-models joined together. We performed several evaluations by simulation in order to account for non exponential events. Values for the model parameters were extracted from direct measurements on the system prototype. The analysis, although still preliminary, shows: i) how to set proper values for the parameters, and ii) the efficacy of the system which calibrates different recovery actions.

Acknowledgements. This work has been partially supported by the Italian Ministry for Education, University and Research (MIUR) within the projects: FIRB "WEB-MINDS: Wide-scalE, Broadband, MIddleware for Network Distributed Services", "Strumenti, Ambienti e Applicazioni Innovative per la Societá dell'Informazione" , SOTTOPROGETTO 4, and Legge 449/97 Progetto "SP1 Reti Internet: efficienza, integrazione e sicurezza".

References

1. Microsoft Corporation (2002), NET Framework Reference, http://msdn.microsoft.com/netframework/techinfo/documentation/default.asp
2. B. Shannon (2002), Java 2 Platform Enterprise Edition Specification, v1.4, http://java.sun.com/j2ee
3. J. Arlat, J.-C. Fabre, M. Rodríguez, F. Salles, Dependability of COTS Microkernel-Based Systems, IEEE Transactions on Computers, 2002 (Vol. 51, No. 2)
4. P. Narasimhan, and P.M. Melliar-Smith, State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects, in proc. of The 2001 International Conference on Dependable Systems and Networks, 2001.
5. C. Sabnis, W.H. Sanders, D.E. Bakken, M.E. Berman, D.A. Karr, M. Cukier, AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects, in proc. of The IEEE 17th Symposium on Reliable Distributed Systems, 1998.
6. Z.T. Kalbarczyk, R.K. Iyer, S. Bagchi, K. Whisnant, Chameleon: a Software Infrastructure for Adaptive Fult Tolerance, IEEE Trans. on Parallel and Distributed Systems, vol. 10, pp. 560–579, 1999.
7. R. Baldoni, C. Marchetti, M. Mecella, A. Virgillito, An Interoperable Replication Logic for CORBA Systems, in proc. of The 2nd International Symposium on Distributed Object Applications 2000 (DOA00), 2000.
8. B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt, DOORS: TowardsHigh-performance Fault-tolerant CORBA, in proc. of International Symposium on Distributed Objects and Applications (DOA'00), 2000.

9. D. Cotroneo, N. Mazzocca, L. Romano, S. Russo, Building a Dependable System from a Legacy Application with CORBA, Journal of Systems Architecture, vol. 48, pp. 81–98, 2002.
10. J.C. Fabre, T. Perennou, A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach, IEEE Transactions on Computers, vol. 47, pp. 78–95, 1998.
11. A. Avizienis, J.C. Laprie, and B. Randell, Fundamental Concepts of Dependability, LAAS, Technical Report n.ro 01145, Tolosa (France), Technical Report n.ro 01145 2001.
12. A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, F. Grandoni, Threshold-Based Mechanisms to Discriminate Transient from Intermittent Faults, IEEE Transactions on Computers, vol. 49, pp. 230–245, 2000.
13. D. Powell, G. Bonn, D. Seaton, P. Verissimo, F. Waeselynck, The delta-4 approach to dependability in open distributed computing systems, in Proc. of the 18th International Symposium on Fault Tolerant Computing Systems (FTCS 18), 1988.
14. O.M. Group, Fault-Tolerant CORBA Specification, v1.0, OMG, http://www.omg.org, document ptc/00-04-04 2001.
15. L. Romano, S. Chiaradonna, A. Bondavalli, D. Cotroneo, Implementation of Threshold-based Diagnostic Mechanisms for COTS-based Applications, in proc. of The 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002), Osaka, Japan, 2002.
16. K.K. Goswami, R.K. Iyer,Simulation of Software Behavior Under Hardware Faults,in Proc. of the 23rd Annual International Symposium on Fault-Tolerant Computing, 1993.
17. R.K. Iyer, D. Tang, Experimental Analysis of Computer System Fault tolerance", in chapter 5 of Fault-Tolerant Computer System Design, D.K. Pradhan, Prentice Hall Inc., 1996.
18. D. Stott, P. H. Jones, M. Hamman, Z. Kalbarczyk, R. K. Iyer, NFTAPE: networked fault tolerance and performance evaluator, in proc. of International Conference on Dependable Systems and Networks, 2002.
19. D.E. Bakken, Z. Zhan, C.C. Jones, D.A. Karr, Middleware support for voting and data fusion, presented at DSN01- IEEE International Conference on Dependable Systems and Networks, Gotenburg, Sweden, 2001, pp. 453–462.
20. DBench Consortium, Measurements, Deliverable ETIE1, IST-2000-25425 Dependability Benchmarking (DBench), 2002.
21. R. Mullen, The Lognormal Distribution of Software Failure Rates: Origin and Evidence, in proc. of The Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998.
22. W. H. Sanders and J. F. Meyer, A Unified Approach for Specifying Measures of Performance, in Dependable Computing for Critical Applications, vol. 4 of Dependable Computing and Fault-Tolerant Systems, A. Avizienis, H. Kopetz, and J. C. Laprie, Eds.: Springer Verlag, 1991, pp. 215–237.
23. Ken Birman, Robert Constable, Mark Hayden, Christopher Kreitz, Ohad Rodeh, Robbert van Renesse, Werner Vogels,The Horus and Ensemble Projects: Accomplishments and Limitations, in Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00), 2000.
24. D. Cotroneo, A. Mazzeo, L. Romano, S. Russo, Implementing a CORBA-based architecture for leveraging the security level of existing applications, 8th International Symposium on Distributed Objects and Applications (DOA 2002), Lecture Notes in Computer Science Series, LNCS 2519, Springer Verlag, 2002.

An Architectural-Level Exception-Handling System for Component-Based Applications

Fernando Castor Filho, Paulo Asterio de C. Guerra, and
Cecília Mary F. Rubira

Instituto de Computação
Universidade Estadual de Campinas
P.O. Box 6176. CEP 13083-970, Campinas, SP, Brazil.
{fernando, asterio, cmrubira}@ic.unicamp.br
+55 (19) 3788-5842 (phone/fax)

Abstract. Component-based software systems built out of reusable software components are being used in a wide range of applications which have high dependability requirements. In order to accomplish the required levels of dependability, it is necessary to incorporate into these complex systems means for to cope with software faults. Exception handling is a well-known technique for adding forward error recovery to software systems supported by various mainstream programming languages. However, exception handling for component-based applications at the architectural level introduces new challenges which are not addressed by traditional exception handling systems, such as unavailability of source code, specially when off-the-shelf components are employed. In this paper, we present an exception handling system which adds fault tolerance to component-based systems at the architectural level. Our solution considers issues which are specific to component-based applications, such as unavailability of source code. We also present a framework which implements the proposed exception handling system for applications built using the C2 architectural style.

1 Introduction

Modern computing systems require evolving software that is built from existing software components, in general developed by independent sources [2]. Hence, the construction of component-based systems with high dependability requirements out of existing software components represents a major challenge, since few assumptions can generally be made about the level of confidence of off-the-shelf components. In this context, an approach for the provision of fault tolerance based on the system's software architecture[3] is necessary in order to build dependable software systems assembled from untrustworthy components [13].

Exception handling[10] is a well-known technique for incorporating fault tolerance into software systems. An exception handling system (EHS) offers control structures which allow developers to define actions that should be executed when

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 321–340, 2003.

an error is detected. The user of such a system should be able to signal exceptions when an error is detected. The EHS should be able to find and activate an exception handler to recover from errors and put the system back in a coherent state. Even though many well-known programming languages provide EHSs [11, 17,27], exception handling for component-based systems addressed at the architectural level remains an open issue. Component-based systems introduce new challenges which are not addressed by traditional EHSs. Some of these challenges include:

- exception handlers should be attached to higher-level abstractions specific to component-based systems, such as components and connectors, not only to implementation language constructs, such as methods and classes;
- the source code for the system components may not be available, specially when off-the-shelf components are employed. Hence, it is not possible to modify these components in order to introduce exception handling;

In this paper, we present ALEx, an architectural-level EHS which addresses the concerns presented above. ALEx is based on the work of Guerra et al [12], which describes a structuring concept for building fault-tolerant component-based systems based on the concept of idealised fault-tolerant component [1]. An idealised fault-tolerant component promotes separation of concerns between the abnormal activity (fault tolerance measures) of a system and its normal activity. Upon the receipt of a service request, an idealised fault-tolerant component produces three types of responses: *normal responses* in case the request is successfully processed, *interface exceptions* in case the request is not valid, and failure exceptions, which are produced when a valid request is received but cannot be successfully processed. Idealised fault-tolerant components may be organized into layers, so that components may handle exceptions raised by components located in other layers.

We also describe an object-oriented framework, called FaTC2, which implements our proposed EHS. FaTC2 is an extension of C2.FW [20], an OO framework which provides an infrastructure for building applications using the C2 architectural style [28]. The C2 style is a component-based architectural style [9] which supports large grain reuse and flexible system composition, emphasizing weak bindings between components. This style was chosen due to its ability to compose heterogeneous off-the-shelf components [20].

The rest of this paper is organized as follows. Section 2 provides some background information. Section 3 gives a motivation for the construction of an EHS for component-based applications. Section 4 presents ALEx, our approach for architectural-level exception handling. Section 5 provides a brief description of FaTC2, a Java implementation of ALEx based on the C2 architectural style. Section 7 gives a brief comparison with related work. Finally, Section 8 summarizes our conclusions and suggests directions for future work.

2 Background

2.1 Exception Handling

Following the terminology adopted by Lee and Anderson [1], a system consists of a set of components that interact under the control of a design. A fault in a component may cause an error in the internal state of the system which eventually leads to the failure of the system. Two techniques are available for errors: (i) *forward error recovery* and (ii) *backward error recovery*. The first technique attempts to return the system to an error-free state by applying corrections to the damaged state. The second technique attempts to restore a previous state which is presumed to be free from errors. Exceptions and exception handling constitute a common mechanism applied to the provision of forward error recovery.

An exception handling system (EHS) allows software developers to define exceptional conditions and to structure the abnormal activity of software components. When an exception is signalled by a component, the EHS is responsible for changing the normal control flow of the computation within a component to its exceptional control flow. Therefore, raising an exception results in the interruption of the normal activity of the component, followed by the search for an appropriate *exception handler* (or simply *handler*) to deal with the signaled exception. The set of handlers of a component constitutes its *abnormal activity* part. For any exception mechanism, *handling contexts* associate exceptions and handlers. Handling contexts are defined as regions in which the same exceptions are treated in the same way. Each context should have a set of associated handlers, which are executed when the corresponding exception is raised.

2.2 C2 Architectural Style

In the C2 architectural style [28], components communicate by exchanging asynchronous messages sent through connectors, which are responsible for the routing, filtering, and broadcast of messages. Figure 1 shows a software architecture using the C2 style where the elements A, B, and D are components, and C is a connector. The thin vertical lines correspond to connections, ports for connecting the architectural elements. A configuration is a set of components, connectors, and connections between these elements. Components and connectors have a *top interface* and a *bottom interface* (Figure 1). Systems are composed in a layered style, where the top interface of a component may be connected to the bottom interface of a connector and its bottom interface may be connected to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors. Two types of messages are defined by the C2 style: requests, which are sent upwards through the architecture, and notifications, which are sent downwards. Requests ask components in upper layers of the architecture for some service to be provided, while notifications signal a change in the internal state of a component.

The C2.FW framework [20,29] provides an infrastructure for building C2 applications. The C2.FW Java [11] framework comprises a set of classes and

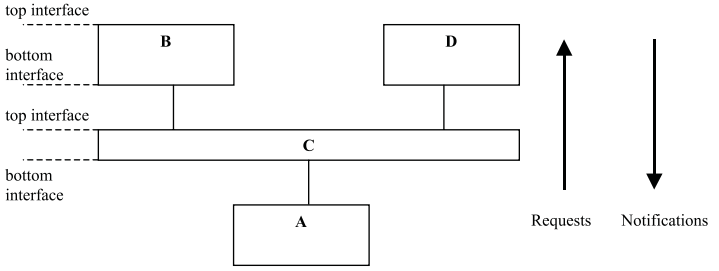


Fig. 1. An example of software architecture based on the C2 style.

interfaces which implement the abstractions of the C2 style, such as components, connectors, messages, and connections. C2.FW has been implemented in C++, Java, Python and Ada.

2.3 Idealised C2 Component

The work of Guerra et al.[12] uses the concept of Idealised Fault-Tolerant Component to structure the architecture of component-based software systems compliant with the C2 architectural style. It introduces the Idealized C2 Component (iC2C), which is equivalent, in structure and behavior, to the idealised fault-tolerant component. Service requests and normal responses of an idealised fault-tolerant component are mapped as requests and notifications in the C2 architectural style. Interface and failure exceptions of an idealised fault-tolerant component are considered subtypes of notifications.

The iC2C is composed of five elements: NormalActivity and AbnormalActivity components, and iC2C_top, iC2C_internal, and iC2C_bottom connectors. Its internal structure is presented in Figure 2. The NormalActivity component processes service requests and answers them through notifications. It also implements the error detection mechanisms of the iC2C. Internally, a NormalActivity component may be either a stateless or stateful component. However, since an iC2C should guarantee that each request received is processed in the initial state of the iC2C[13], stateless components are easier to deal with. In order to employ stateful components, some means for guaranteeing that requests are processed atomically, such as atomic transactions, should be provided.

The AbnormalActivity component encapsulates the exception handlers (error recovery) of the iC2C. While an iC2C is in its normal state, the AbnormalActivity component remains inactive. When an exceptional condition is detected, it is activated to handle the exception. In case the exception is successfully handled, the iC2C returns to its normal state and the NormalActivity component resumes processing. Otherwise, a failure exception is signalled to components in lower layers of the architecture, which become responsible for handling it.

The iC2C_bottom connector is responsible for filtering and serializing requests received by the iC2C. This conservative policy aims at guaranteeing that

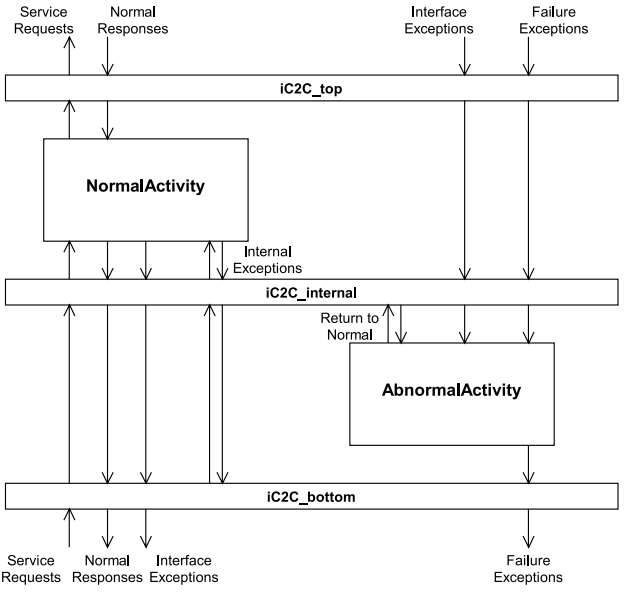


Fig. 2. Internal structure of an iC2C.

requests are always received by the NormalActivity component in its initial state, to avoid possible side-effects of an exceptional condition caused by a concurrent service request. The iC2C_internal connector is responsible for message routing inside the iC2C. The destination of the messages sent by the internal elements of the iC2C depends on its type and whether the iC2C is in a normal or abnormal state.

The iC2C_top connector encapsulates the interaction between the iC2C and components located in upper levels of the architecture. It is responsible for guaranteeing that service requests sent by the NormalActivity and AbnormalActivity components to other components located in upper levels of the architecture are processed synchronously (request/response). The iC2C_top connector also performs domain translation, converting incoming notifications to a format which the iC2C understands and outgoing requests to a format which the application understands.

The structure of the iC2C makes it compatible with the constraints imposed by the C2 architectural style. Hence, an iC2C may be incorporated into an existing C2 configuration. Previous experiments [12,13] with the iC2C model have shown its adequacy for the construction of component-based systems, including systems built from off-the-shelf components [14].

3 Exception Handling at the Architectural Level

Exception handling for component-based software systems introduces some difficulties which are not addressed by traditional EHSs. These particularities are a consequence of the differences between the abstractions supported by programming languages and component-based software development. In a traditional EHS defined by a programming language, exception handlers are attached to the constructs the language supports. For example, Guide [18] is an object-oriented language which supports the attachment of handlers to statements, methods and classes. Similarly, an EHS which supports the construction of component-based systems should allow exception handlers to be attached to their basic elements, namely, components, connectors, and configurations.

In traditional EHSs, when an exception is signaled, a handler is searched locally, depending on the handling context. If none is found, the exception is propagated to the enclosing method invocation. However, this scheme for exception propagation may not be adequate for component-based systems. In software architectures, connectors are represented as first-class design entities, due to the potential complexity of the interactions among components. This feature allows the support of more sophisticated communication elements than simple method calls, and separates explicitly the execution flow of an application from its chain of method invocations. An EHS for component-based applications should consider these requirements during its design and support the propagation of exceptions according to the flow of information between components. In the C2 style, for example, components communicate by means of asynchronous message passing. Service requests are sent from components in lower layers of an architecture to components in upper layers, and response notifications flow in the opposite direction. Hence, in a C2 architecture, if a component issues a request which triggers the raising of an exception, the latter should be propagated to the component which issued the request. In other words, it should be propagated downwards the architecture, and not along the method invocation chain.

Finally, an architectural-level exception handling system should support the attachment of handlers to components without requiring modifications on them. This requirement is very important because, in component-based software development, the source code for the system components may not be available, specially if off-the-shelf components are employed. In order to leverage exception handling for component-based applications, we have devised an architectural-level EHS, called ALEx, which addresses the concerns stated above.

4 ALEx: An Exception Handling System for Component-Based Applications

In this section, we present the main characteristics of our exception handling system and discuss our design decisions to address the requirements discussed in Section 3. We follow the guidelines established by Garcia et al [7], and describe ALEx in terms of the following features: exception representation, han-

dler definition, handler attachment, exception propagation, handler search, and continuation of the control flow.

In our approach, an iC2C (Section 2.3) can represent a component, connector or configuration which has an architectural-level exception handler attached. We also employ the C2 architectural style for describing examples of architecture configurations, since it is leveraged by the concept of iC2C. Exceptions are wrapped by C2 notifications in order to be propagated along the layers of the system's architecture. Notifications representing exceptions are called *exception notifications*, and they are distinguished from normal notifications. Furthermore, we define regular C2 components as C2 components that: (i) do not send exception notifications, and (ii) will not recognize an exception notification received as being the signalling of an exception. In other words, a regular C2 component will recognize an exception notification as an ordinary notification sent in response to a request issued prior to the receipt of the notification. We will not consider special cases, such as regular C2 components capable of sending exception notifications.

Hereafter, we use the term "exception" to designate both *exceptions* and *exception notifications*, since the latter is simply an implementation concept which is not directly related to the EHS. For situations where implementation issues are relevant, we highlight the difference between the two terms.

4.1 Exception Representation

According to the taxonomy defined by Garcia et al [8], ALEx represents exceptions as *data objects*, that is, objects which are used for holding context information which is relevant for their handling. More specifically, we define exceptions as messages, which constitute a suitable abstraction for component-based systems. Exceptions are raised by calling a specific keyword (in our Java implementation, the `throw` keyword).

Different types of exceptions are organized hierarchically as classes. The class ArchitecturalException is the root of this hierarchy. ALEx defines two subclasses of ArchitecturalException, FailureArchitecturalException and InterfaceArchitecturalException, which correspond to the failure exceptions and interface exceptions of an idealised fault-tolerant component, respectively. Any raised exception which is not of type InterfaceArchitecturalException is treated as a failure exception. Interface exceptions must be explicitly raised.

The signatures of the operations in component interfaces should explicitly indicate the exceptions they raise. According to the work of Garcia et al.[8], this practice leads to better readability, modularity, maintainability, reusability and testability. Furthermore, explicitly declaring the exceptions raised by a component allows static checks to be performed by means of analysis tools, increasing dependability.

4.2 Handler Attachment and Definition

Our EHS supports multi-level attachment of handlers, that is, handlers may be associated with: (i) a component, (ii) a connector, (iii) a configuration. Figure 2 shows a set of exception handlers, represented by the AbnormalActivity component, attached to a component, represented by the NormalActivity component. The abnormal activity of a component is defined by a class which implements a set of methods, each one representing an exception handler. Furthermore, each of these classes should define at least a *default handler*, capable of handling an exception of the ArchitecturalException type. Figure 3 shows a partial definition of an exception handler which handles exceptions of types IOException and ArchitecturalException.

```
public class AbnormalActivity (…) {

  public Message handleException(IOException e) {
    // Body of a handler for IOException.
  }

  public Message handleException(Exception e) {
    // Body of generic handler. Must be implemented!
  }

  (…)
}
```

Fig. 3. Definition of handlers in ALEx.

Exception handlers may also be attached to configurations, as well as to components and connectors. The possibility of attaching handlers to specific configurations allows that different exception handling contexts be defined within a single application. This is achieved by nesting exception handlers, as illustrated by Figure 4, where an iC2C is used as the NormalActivity component of another iC2C. This feature is specially useful for systems which have critical regions that require a higher level of fault tolerance.

4.3 Exception Propagation and Handler Search

Our exception handling model adopts explicit propagation of exceptions. The benefits of this approach are discussed in the work of Garcia et al.[8]. In ALEx, the handling and propagation of an exception depends on whether it is a *failure exception* or an *interface exception*. When an iC2C raises a failure exception, its handling is limited to the handlers within the same iC2C (that is, its AbnormalActivity component). If the exception can not be handled, then the predefined exception FailureArchitecturalException is further propagated. A handler may also explicitly resignal the exception to a component in a lower layer of the

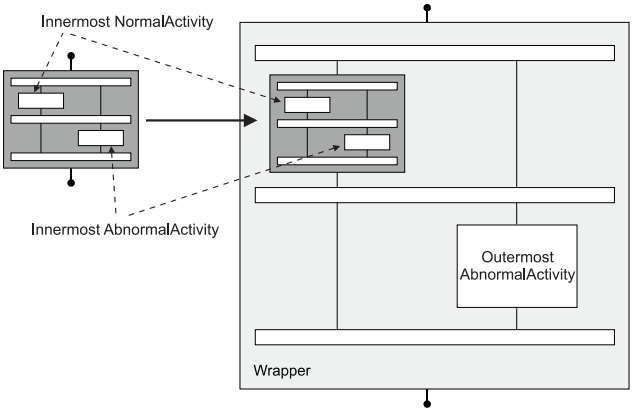


Fig. 4. Nesting exception handling contexts.

architecture. In Figure 4, exceptions raised by the NormalActivity component of the innermost iC2C are handled by the AbnormalActivity component within it. If the internal handling fails, the handler may resignal the exception to the AbnormalActivity component of the outermost iC2C. In case an exception reaches the lowest level of an architecture, an exception handler for the entire system should be executed.

An interface exception raised upon the receipt of a service request is not handled by the component, since it does not indicate that the component is faulty. In this case, the exception is propagated to the client component which issued the request. The client component handles this exception in the same manner as a failure exception, since it possibly indicates a fault within the client component.

The search for handlers for a failure exception raised by the NormalActivity component of an iC2C is defined as follows:

1. The EHS tries to find a specific handler for the exception in the AbnormalActivity component within the same iC2C. Handlers are searched according to the class hierarchy defined for exceptions:
   – if an exception of class C is raised in a given system, a handler for exceptions of class C will be searched;
   – if none is found, the EHS proceeds searching for a handler for an immediate superclass S of C;
   – if none is found, the EHS searchers for a handler for an immediate superclass of S, and so on.
2. If no specific handler is found, the default handler is selected. This handler may actually handle the exception or simply resignal it.
3. In the latter case, if the iC2C which raised the exception is wrapped by another iC2C (as is the case in Figure 4), step 1 is repeated for the outermost AbnormalActivity component. Otherwise, the exception is propagated

downwards the architecture, to the client component which issued the request that triggered its signaling. Step 1 is then repeated for the AbnormalActivity component of the client component.

The search for handlers for an interface exception raised by the NormalActivity component of an iC2C proceeds in a similar manner, except that steps 1 and 2 are not initially performed.

4.4 Continuation of the Control Flow

ALEx adopts the *termination* model[5], that is, if during the processing of a service request a component raises an exception and it is successfully handled by another component, execution is resumed by the latter.

Figure 5 presents an example of an architecture composed by two idealised C2 components (ClientComponent and ServerComponent), and a connector linking them. Figure 6 shows a UML seq. diagram illustrating a scenario where the control flow continues after an exception is successfully handled. The following steps describe the diagram:

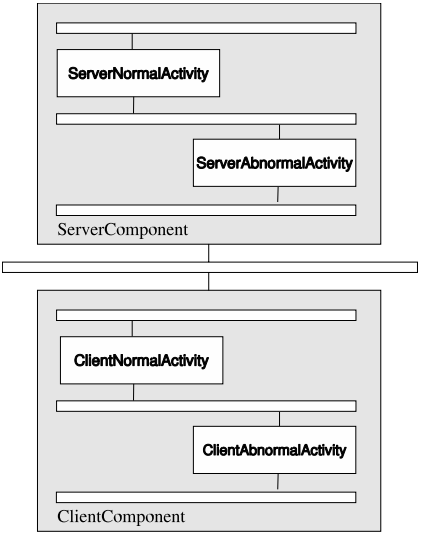


Fig. 5. A C2 architecture composed by a client component and a server iC2C.

1. ClientComponent requests a service from ServerComponent;
2. ServerNormalActivity tries to handle the service request;
3. ServerNormalActivity raises an exception which is received by ServerAbnormalActivity;

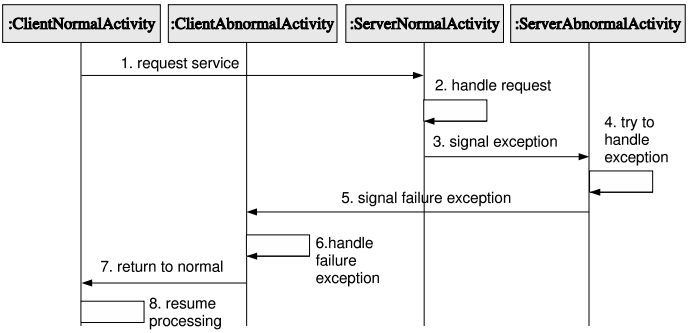


Fig. 6. A scenario illustrating the termination model adopted by ALEx.

4. ServerAbnormalActivity tries to handle the exception;
5. ServerAbnormalActivity fails to handle the exception and raises a failure exception;
6. ClientComponent receives the exception and routes it to ClientAbnormalActivity;
7. ClientAbnormalActivity handles the exception and sends a *return to normal* request to ClientNormalActivity, indicating that processing should be resumed;
8. ClientNormalActivity resumes processing.

5 Exception Handling System in the Framework FaTC2

In this section, we describe FaTC2, an object-oriented framework which implements our architectural-level exception handling approach. FaTC2 is an extension of the Java [11] version of the C2.FW framework. The original C2.FW framework does not provide adequate support for the construction of fault-tolerant systems. FaTC2 extends C2.FW with the concept of iC2C, in order to provide the support for forward error recovery, by means of the EHS described in Section 4.

Figure 7 presents a partial class hierarchy for FaTC2, and its intersection with C2.FW. In the following sections, we describe the framework FaTC2, based on the notions described in Figure 2.

5.1 IC2C

The iC2C is the basic unit provided by FaTC2 for attaching handlers to components or configurations. The creation of an iC2C is encapsulated by the IC2C class. In order to create an instance of IC2C, objects representing the NormalActivity and AbnormalActivity components (Figure 2) must be supplied. Optionally, the developer may chose to also supply objects representing the iC2C_top and iC2C_bottom connectors, in case filtering or domain-translation are required. Otherwise, default implementations are employed.

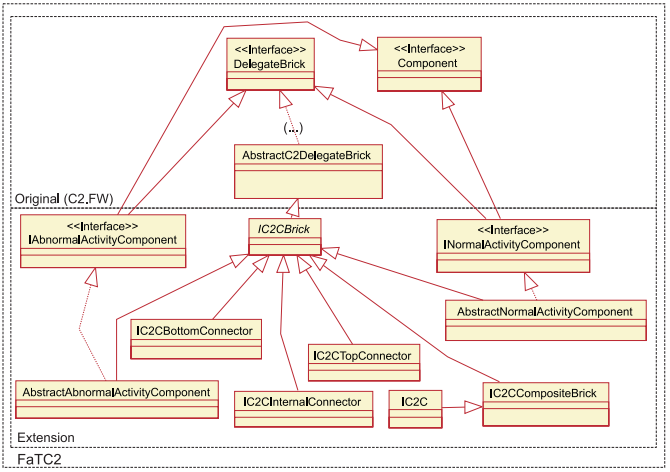


Fig. 7. A partial class hierarchy for C2.FW and FaTC2.

Although the IC2C class may be used directly in an application, it is recommended that developers create subclasses of it, specifying the NormalActivity and AbnormalActivity components, and iC2C_top and iC2C_bottom connectors which are to be used.

An analogous structuring may be used for representing fault-tolerant connectors, as long as the semantic differences between components and connectors are taken into account.

5.2 NormalActivity Component

The NormalActivity component encapsulates the functionality (normal activity) of an iC2C. It may represent both a single component and a configuration. In this work, we will only address the case where an iC2C represents a single component.

In order to define a NormalActivity component, a developer must provide a class that implements the INormalActivity interface. This interface declares three operations which define the application-dependent behavior of the component: `handleRequest()`, `returnToNormal()`, and `reset()`.

The `handleRequest()` method is responsible for (i) processing service requests and (ii) detecting errors. It takes as argument the request message to be processed, and returns a response notification to be delivered to the client component. If an error is detected during the processing of a service request, this method signals an exception, which may be a FailureArchitecturalException or an InterfaceArchitecturalException. Exceptions are caught by the framework and packaged as exception notifications, which are sent to the AbnormalActivity component.

The `returnToNormal()` and `reset()` methods are related to the abnormal activity of the iC2C. The former is called when the iC2C has successfully handled

an exception, and should resume processing. The latter is called when the iC2C is unable to handle an exception, and should perform some cleanup actions before handling new requests.

FaTC2 provides developers with an abstract class which implements the application-independent behavior of the NormalActivity component, as defined by ALEx (Section 4). This class is called AbstractNormalActivityComponent, and should be extended by the class which implements the NormalActivity component for a given iC2C. In the situations described above, the tasks of delivering requests to the `handleRequest()` method, sending response notifications to client components, and packaging and sending exception notifications to the AbnormalActivity component are performed by AbstractNormalActivityComponent.

In case the handling of a request demands the NormalActivity component to request services from components located in upper layers of the architecture, the AbstractNormalActivityComponent class provides a utility method, `requestService()`, which may be used to send synchronous (request/response) requests transparently, upwards the architecture.

5.3 AbnormalActivity Component

The AbnormalActivity component encapsulates the exception handlers of an iC2C. In order to implement it, a developer must provide a class that implements the IAbnormalActivityComponent interface. This interface declares a single method, `handleException()`, which defines the default exception handler of the component. This scheme enforces the policy defined by ALEx, that at least the default exception handler must be implemented by every AbnormalActivity component.

Additional handlers are defined by *handler methods* that are declared in the same class which implements the IAbnormalActivityComponent interface. The order in which they are declared is not important. Handler methods present the following structure:

```
public Message handleException(<exception> e, Request m) raises
Exception {
   // Body of a handler for <exception>.
}
```

In the code snippet above, *<exception>* stands for the exception type (a Java class) which the handler is capable of handling. The `Request r` parameter refers to the request which was being processed when the exception was signaled. If an exception is successfully handled, the handler method returns an object of type Message. This represents a C2 message which is delivered to the NormalActivity component in order for processing to be resumed. If an exception can not be handled, the handler method should resignal it or raise another exception. Either way, the exception is propagated to the enclosing exception handling context (Section 4.3).

FaTC2 provides developers with an abstract class which implements the application-independent behavior of the AbnormalActivity component, as defined by ALEx (Section 4). This class is called AbstractAbnormalActivityComponent, and should be extended by the class which implements the AbnormalActivity component for a given iC2C.

In case the handling of an exception requires the AbnormalActivityComponent to request services from other components, or from the NormalActivityComponent in the same iC2C, class AbstractAbnormalActivityComponent provides methods which allow synchronous requests to be carried transparently, similarly to the AbstractNormalActivityComponent class.

5.4 iC2C_Top, iC2C_Bottom, and iC2C_Internal Connectors

The IC2CTopConnector, IC2CBottomConnector, and IC2CInternalConnector classes are default implementations for the iC2C_top, iC2C_bottom, and iC2C_internal connectors, respectively.

IC2CTopConnector and IC2CBottomConnector may be extended in order to implement filtering of notifications in the top domain of an iC2C or requests in its bottom domain, respectively. A filtering scheme is defined by implementing the `accept()` method in a subclass of IC2CTopConnector or IC2CBottomConnector. A message m is processed only if `accept(`m`) == true`.

Subclasses of IC2CTopConnector may also implement domain translation in the top domain of the iC2C. The methods `translateIncomingMessage()` and `processOutgoingMessage()` are responsible for this task and are called by FaTC2, respectively, immediately after a message has been *accepted* by the iC2C_top connector, and immediately before a given message is sent by it. The iC2C_bottom connector is not expected to perform domain translation. In the C2 architectural style, an element placed in an upper layer of an architecture should make no assumptions about elements in the lower layers [28].

In case no filtering or domain translation is necessary, the default implementations for the iC2C_top and iC2C_bottom connectors may be used.

The IC2CInternalConnector class is reused without needing any specialization, since its only task is to route messages inside an iC2C.

6 An Application Example

In order to show the usability of FaTC2, we present a small example extracted from the Mine Pump Control System[23]. The problem is to control the amount of water that collects at the mine sump, switching on a pump when the water level rises above a certain limit and switching it off when the water has been sufficiently reduced. In this section, we describe an implementation for the example application which uses the infrastructure provided by FaTC2.

6.1 Description of the Architecture

The C2 architecture of our example is shown in Figure 8. The Pump component commands the physical pump to be turned on/off. Component LowWaterSensor signals a notification when the water level is low. WaterFlowSensor checks whether water is flowing out of the sump. The IdealPumpControlStation component controls the draining of the sump by turning on/off the pump, according to the level of the water in the sump. It includes an exception handler which is executed when the pump is turned on but no water flow is detected. The error handler is implemented by the AbnormalPumpControlStation component. The Pump, LowWaterSensor and WaterFlowSensor components have been implemented as simple C2 components, while IdealPumpControlStation is an iC2C. In order to build the IdealPumpControlStation, five classes are implemented: NormlPumpControlStation, AbnormalPumpControlStation, PumpControlStationTop, IdealPumpControlStation and TranslationConnector.

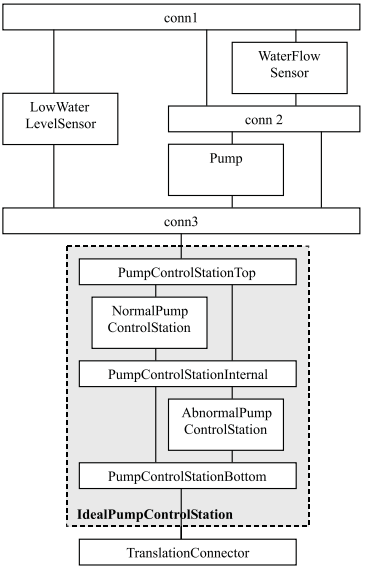


Fig. 8. C2 configuration for the fault-tolerant Mine Pump Control System.

Class NormalPumpControlStation implements the NormalActivity component of IdealPumpControlStation, that is, the methods defined by the INormalActivityComponent interface(Section 5.2). Due to the support provided by FaTC2, no messages need to be explicitly sent by any of the methods in NormalPumpControlStation; that is, the architect does not need to understand the internal protocol of the iC2C or the way it is implemented.

The AbnormalPumpControlStation class implements the exception handler of the IdealPumpControlStation. When an exception message is received by the `handleException()` method, the latter keeps sending new requests to Pump until either water flow is detected or the maximum number of retries permitted is reached. In the former case, normal activity is resumed(the method simply returns). In the latter, a FailureArchitecturalException message is sent downwards the architecture(the method throws an IC2CFailureException). The following code snippet partially illustrates this situation.

```
public Message handleException(Exception e, Request m)
 throws Exception {
(...)
 if(this.retries >= this.MAX_RETRIES) {
    throw new IC2CFailureException(e);
 }
(...)
```

In order to send an exception message downwards the architecture, the architect should throw a Java exception. In the example above, an exception of type IC2CFailureException, a subtype of Exception, is thrown.

The PumpControlStationTop class provides the IdealPumpControlStation component with an extension of the IC2CTopConnector class which performs filtering. When a request is issued by the IdealPumpControlStation, PumpControlStationTop records the type of the request sent, so that only a notification which is a response to that request is allowed to be processed. To build this filtering scheme, two methods had to be implemented: `accept()` and `processOutgoingMessage()`(Section 5.4).

IdealPumpControlStation is a subclass of IC2C. The IdealPumpControlStation class defines a public constructor which takes as argument the name of the IdealPumpControlStation instance to be created. TranslationConnector translates requests and notifications at the bottom interface of the IdealPumpControlStation (Figure 8).

The implementation of this architecture using FaTC2 was relatively easy, although a large amount of glue code had to be implemented in order for the NormalPumpControlStation component to work with FaTC2. This was due to the fact that the selected example application worked in an asynchronous manner while FaTC2 implements an iC2C which was modelled as a synchronous (request/response) entity. This highlighted some of the limitations of this synchronous approach and pointed out some directions we should follow in our future research, discussed in Section 8.

7 Related Work

Software fault-tolerance at the architectural level is a young research area that has recently gained considerable attention [14]. Most of the existing works in

this area emphasize the creation of fault-tolerance mechanisms [4,15,21] and description of software architectures with respect to their dependability properties [22,25]. Some approaches based on the idea of design diversity [1] have been developed in the context of the reliable evolution of component-based distributed systems. Both the Hercules framework [4] and the concept of Multi-Versioning Connectors [21] maintain old and new versions of components working concurrently, in order to guarantee that the expected service is provided, even if there are faults in the new versions. Both approaches are orthogonal to ours and could be used in conjunction.

Stavridou and Riemenscheneider [25], and Saridakis and Issarny [22] emphasize the formal description of architectures in order to prove they are reliable. By employing these specifications together with refinement laws which guarantee the preservation of the reliability property, both approaches intend on producing concrete architectural descriptions which are easily translated to code. None of the two works addresses the problem of incorporating error recovery into existing components.

The concept of iC2C[12] defines a structure for incorporating fault tolerance into component-based systems at the architectural level. It defines an internal protocol followed by its elements in order to enforce damage confinement [1]. Our work refines the concept of iC2C by introducing elements which are not addressed by its definition, such as the the representation of exception handlers and the enforcement of explicit exception propagation.

The work by Guerra et al.[14] deals with the problem of integrating COTS components in systems with high reliability requirements. It presents a case study where the concept of iC2C is used, together with protective wrappers [14]. The goal of this approach is to make non-reliable COTS components which represent a critical regions of systems reliable.

The work by Issarny and Banâtre [15] describes an extension to existing architecture description languages for specifying architectural-level exceptions (configuration exceptions). This work differs from ours because it emphasizes fault treatment [1] at the architectural level, by means of architecture reconfiguration. Our work, on the other hand, emphasizes architectural-level error recovery. Furthermore, it defines exceptions which should not be handled by any component in the architecture, that is, exception handlers are defined for the whole architecture and are activated under specific situations. In our approach, a component raises an architectural-level exception because it is unable to handle it, and other components in the architecture or exception handlers attached to enclosing configurations (as shown in Figure 4) may try to handle it as well.

8 Conclusions and Future Work

According to Sprott [24], there is a general consensus in the industry that software components will bring profound changes to the way software is built. Even now, software systems built out of reusable software components are used in a wide range of applications. Many of these systems have high dependability requirements and, in order to achieve the required levels of dependability, it is necessary to incorporate into them means for coping with software faults.

In this paper, we have presented ALEx, an architectural-level exception handling system which leverages the construction of fault-tolerant component-based applications. The use of traditional language-based exception handling systems for building fault-tolerant component-based systems presents some challenges which have been discussed in Section 3. ALEx addresses these issues by instituting exception handling at the architectural level. We have also briefly described FaTC2, an object-oriented framework for the construction of fault-tolerant component-based systems which implements ALEx.

It is important to note that architectural-level exception handling is not a replacement for language-level exception handling. In our view, exception handling in the language level should be the main technique for achieving fault tolerance internally to components (intra-components). Architectural-level exception handling (inter-components) should be employed when (i) an exception can not be handled by the component which raised it, and some other component in the architecture might be able to handle it, and (ii) mechanisms for error detection and recovery must be introduced in a component in order to make it trustworthy (or *more* trustworthy), but the component should not be modified or its source code is not available.

Until the present moment, the iC2C has been modeled as a synchronous (request/response) entity and the implementation of FaTC2 conforms to this model. That means that an iC2C is unable to handle asynchronous notifications and that requests are issued under the assumption that a response will be eventually received. This restriction might be too conservative for some applications, since a large amount of glue code may be necessary if a synchronous iC2C needs to interact with asynchronous components (Section 6). Hence, a future work for is the definition of an iC2C for which some of these restrictions are relaxed.

In order to better assess our approach, we are currently working on the implementation of a real-world application in which ALEx is employed. This experiment will make possible to evaluate the adequacy of our approach to a real application in which a commercial component model, such as Enterprise Javabeans[26], is employed. We believe that ALEx fits nicely into this kind of system, since previous experiments with the C2 architectural style have shown how different off-the-shelf middleware platforms can be integrated almost seamlessly in a C2 architecture[6].

FaTC2 still does not implement all the features defined by ALEx. Some features, such as support for attaching handlers to arbitrary configurations and hierarchical handler search upon the receipt of an exception (Section 4.3) have not been implemented yet. Hence, another future work for FaTC2 is the implementation of the remaining features described by the specification of ALEx.

Finally, we believe that the design and implementation of ALEx are independent of the C2 architectural style and the iC2C. Hence, we plan to evaluate our approach on generic layered architectures by means of an implementation which does not rely on the concept of iC2C. In this case, new means for associating exception handlers to components would need to be established, in order to maintain the features described in Section 4. Computational Reflection [19] and Aspect-Oriented Programming [16] are good candidates for this task.

Acknowledgements. We would like to thank the anonymous referees, who helped to improve this paper. F. Castor Filho is partially supported by FAPESP/Brazil under grant no. 02/13996-2. P. Guerra is partially supported by CAPES/Brazil. C. Rubira is partially supported by CNPq/Brazil under grant no. 351592/97-0.

References

1. T. Anderson and P. A. Lee: *Fault Tolerance: Principles and Practice*, 2nd Edition. Prentice-Hall (1990).
2. A.W. Brown and K.C. Wallnau: The Current State of CBSE. In: *IEEE Software* **15**(5). IEEE Computer Society Press (1998) 37–46.
3. Paul C. Clements and Linda Northrop: *Software Architecture: An Executive Overview*. Technical Report CMU/SEI-96-TR-003. Software Engineering Institute, Carnegie Mellon University (1996).
4. J. E. Cook and J. A. Dage: Highly Reliable Upgrading of Components. In: *Proceedings of the 21st International Conference on Software Engineering* (1999) 203–212.
5. Flaviu Cristian: Exception Handling. In: T. Anderson (ed.): *Dependability of Resilient Computers*. BSP Professional Books (1989).
6. Eric M. Dashofy and Nenad Medvidovic and Richard N. Taylor. Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)* (1999) 3–12.
7. A. Garcia and D. Beder and C. Rubira: An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach. In : *Proceedings of the 10th International Symposium on Software Reliability Engineering - ISSRE'99*. IEEE Computer Society Press (1999) 52–61.
8. A. Garcia and C. Rubira and A. Romanovsky and J. Xu: A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. In: *Journal of Systems and Software* **59**(2). Elsevier (2001) 197–222.
9. David Garlan and Robert T. Monroe and David Wile: Acme: Architectural Description of Component-Based Systems. In: Gary T. Levens and Murali Sitaraman (eds.): *Foundations of Component -Based Systems*. Cambridge University Press (2000) 47–67.
10. J. B. Goodenough: Exception Handling: Issues and a Proposed Notation. In: *Communications of the ACM* **18**(12). ACM Press, New York, NY (1975) 683–696.
11. James Gosling and Bill Joy and Guy Steele: *The Java Language Specification*. Addison-Wesley (1996).
12. Paulo Guerra and Cecília Rubira and Rogério de Lemos: An Idealized Fault-Tolerant Architectural Component. In: *Proceedings of the 24th International Conference on Software Engineering – Workshop on Architecting Dependable Systems* (2002).
13. Paulo Asterio C. Guerra and Cecília Mary F. Rubira and Rogério de Lemos: A Fault-Tolerant Architecture for Component-Based Software Systems. In: R. de Lemos and C. Gracek and A. Romanosvsky (eds.): *Architecting Dependable Systems*. Lecture Notes in Computer Science, to appear. Springer-Verlag, Berlin Heidelberg New York (2003).

14. Paulo Guerra and Cecília Rubira and Alexander Romanovsky and Rogério de Lemos: Integrating COTS Software Componentes Into Dependable Software Architectures. In: *Proceedings of the 6th International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Computer Society Press (2003).
15. V. Issarny and J. P. Banatre: Architecture-Based Exception Handling. *In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE Society Press (2001).
16. G. Kiczales and J. Lamping and A. Mendhekar and C. Maeda and C. Lopes and J. Lingtier and J. Irwin: Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, Vol 1241. Springer-Verlag, Berlin Heidelberg New York (1997).
17. A. Koening and B. Stroustrup: Exception Handling for C++. In *Jornal of Object-Oriented Programmings* **3**(2) (1990) 16–33.
18. S. Lacourte: Exceptions in Guide, an Object-Oriented Language for Distributed Applications. Lecture Notes in Computer Science, Vol 512. Springer-Verlag, Berlin Heidelberg New York (1991) 268–287.
19. P. Maes: Concepts and Experiments in Computational Reflection. In *ACM SIGPLAN Notices* **22**(12). ACM Press (1987) 147–155.
20. N. Medvidovic and P. Oreizy and R. N. Taylor: Reuse of Off-the-Shelf Components in C2-Style Architectures. In: *Proceedings of the 1997 Symposium on Software Reusability* (1997) 190–198.
21. Marija Rakic and Nenad Medvidovic: Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. In: *Proceedings of the 2001 Symposium on Software Reusability* (2001) 11–18.
22. T. Saridakis and V. Issarny: *Fault-Tolerant Software Architectures*. Technical Report 3350. INRIA (1999).
23. M. Sloman and J. Kramer: *Distributed Systems and Computer Networks*. Prentice Hall (1987).
24. David Sprott: Componentizing the Enterprise Application Packages. In: *Communications of the ACM* **43**(4). ACM Press, New York, NY (2000) 63–69.
25. V. Stavridou and A. Riemenschneider: Provably Dependable Software Architectures. In *Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop*. ACM Press (1998) 133–136.
26. Sun Microsystems: Enterprise Javabeans Specification v2.1 - Proposed Final Draft (2002). Avaliable at `http://java.sun.com/products/ejb/`.
27. S.T. Taft and R.A. Duff: *Ada 95 Reference Manual: Language and Standard Libraries, International Standard Iso/Iec 8652:1995(e)*. Lecture Notes in Computer Science, Vol 1246. Springer-Verlag, Berlin Heidelberg New York (1997).
28. R. N. Taylor and N. Medvidovic and K.M. Anderson and E. J. Whitehead, Jr. and J.E. Robbins: A Component- and Message- Based Architectural Style for GUI Software. In *Proceedings of the 17th International Conference on Software Engineering* (1995) 295–304.
29. UCI: *ArchStudio 3.0 Homepage*. Available at `http://www.isr.uci.edu/projects/archstudio`.

On the Use of Formal Specifications to Analyze Fault Behaviors of Distributed Systems*

Fernando L. Dotti, Osmar M. dos Santos**, and Eduardo T. Rödel

Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
ZIP 90619-900 - Porto Alegre, RS - Brazil
{fldotti, osantos, eduardot}@inf.pucrs.br

Abstract. The development of distributed systems is considered a complex task. The process of assuring the correctness of such systems is even more difficult if we consider open environments (e.g. Internet), where faults may occur. To help such process we make use of formal methods and tools as means to specify and reason about the behavior of distributed systems in the presence of faults. We use a graphical and declarative formal specification language, called Object Based Graph Grammars, to model asynchronous distributed systems. The approach used to specify and analyze the behavior of distributed systems in the presence of faults is based on the observation that a fault behavior can be modeled as an unwanted but possible state transition of a system. Following this approach we can define a fault model, like crash for example, as being a transformation of a model. Thus, a model M_1 of a distributed system can be transformed into a model M_2, that comprehends the behavior of some kind of fault model. To show these methods and tools we model a pull-based failure detector as a case study.

1 Introduction

The development of distributed systems is considered a complex task. In particular, guaranteeing the correctness of distributed systems is far from trivial if we consider the characteristics of open systems, like: massive geographical distribution; high dynamics (appearance of new nodes and services); no global control; faults; lack of security; and high heterogeneity [22]. Among other barriers, in open environments (e.g. Internet) it is hard to assure the correctness of distributed systems because we cannot be sure whether a failure is caused by a fault in the system under construction itself or by the environment in which it runs.

It is therefore necessary that developers of distributed systems have a higher degree of confidence in their solutions during the development phase. To achieve

* This work is partially supported by HP Brasil – PUCRS agreement CASCO (24° TA.), and ForMOS Research Project - FAPERGS (Brazil) grant 01/0759.1 and CNPq (Brazil) grant 520269/98-5.

** This author is partially supported by CNPq (Brazil).

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 341–360, 2003.

that, we have developed methods and tools to assist developers in the construction of distributed systems. In this context, the use of formal methods becomes a good approach to provide a way to precisely describe the system. More specifically, it was developed [7] a formal specification language, called Object Based Graph Grammars (OBGG), suitable for the specification of asynchronous distributed systems. Currently, models defined in this formal specification language can be analyzed through simulation [1] [4]. Moreover, in our activities, we are also working on an approach to verify, using model checking tools, OBGG specifications. Besides the analysis of distributed systems specified in OBGG, it is also possible to generate code for execution in a real environment, following a straightforward mapping from an OBGG specification to Java code [8].

By using the methods and tools described above we have defined a framework to assist the development of distributed systems. The innovative aspect of this framework is the use of the same formal specification language (OBGG) as the underlying unifying formalism [5]. The natural steps to create distributed systems using this framework consists on: specification of the desired system using OBGG, behavior analysis of this specification (at present time such analysis is carried out through simulation), and code generation for execution in a real environment (when satisfied with the behavior of the specification).

The results achieved so far, briefly described above, have addressed the development of distributed systems without considering faults. To address the development of distributed systems for open environments, we present a technique to reason about distributed systems in the presence of selected fault behavior. Thus, we revise our framework to consider some fault models found in the literature [11]. The rationale is to bring the fault behavior to the system model and reason about the desired system in the presence of faults. To achieve that, this paper has two main contributions: i) represent classical fault models in terms of OBGG, and ii) show how to stepwise introduce these fault models in a distributed system model.

In order to reason about distributed systems considering the presence of faults, the developer must first specify a model M_1 of the desired system, and choose a fault behavior to represent in this model M_1. Based on the selected fault behavior, a transformation over M_1 is performed and a new model M_2 is obtained. The model M_2 comprehends the system and the selected fault behavior. As stated in [12], a fault can be modeled as an unwanted but possible state transition of a system. Moreover, it states that the transformation process consists on the insertion of virtual variables and messages that define the fault behavior. Due to the declarative characteristics of OBGG, such transformation process can be carried automatically. In this paper we explain this transformation process. Once the transformed model M_2 is created, the developer can reason about its behavior (currently through simulation, but verification is being integrated to the framework) in the presence of the selected fault behavior.

When the developer is satisfied with the results of the original model M_1, analyzed using the fault behavior introduced in model M_2, then code for execution in a real environment can be generated using the model M_1. While running in a

real environment, if the environment exhibits the fault behavior corresponding to the one introduced in M_2, then the system should behave as expected during the analysis phase.

This paper is organized as follows: Section 2 presents the formal specification language OBGG and introduces the specification of a pull-based failure detector, used as a case study throughout this paper; Section 3 discusses the specification of some classical fault models for distributed systems using OBGG; Section 4 shows an analysis (through simulation) of a transformed specification of the pull-based failure detector; Section 5 presents related works found in the literature; finally, Section 6 brings us to the conclusions and future works.

2 Object Based Graph Grammars

The use of a formal specification language allows the creation of a precise description of a system with a well-defined syntax and semantics. Since this semantics is based on a mathematical model, it is possible to apply formal verification techniques [3] to show that a system specified in the formal language has some desired properties. The formal specification language Object Based Graph Grammars (OBGG) [7], used in this work, is based on a restricted form of graph grammars [9].

The basic notions of graph grammars are that the state of a system can be represented by a graph (the system state graph), and from the initial state of the system (the initial graph), the application of rules successively changes the system state. A rule is composed by a name, a left side, a right side, and a condition. A rule can be applied whenever its left side is a sub-graph of the current system state graph. This is called a match or occurrence of the rule. When applied, the rule brings the system to a new state defined as: items in the left side not present in the right side are deleted; items in the right side not present in the left side are created; and items present in both sides of the rule are preserved [7]. Multiple rules can be applied in parallel if there is no conflict between them, i.e. two or more rules can not modify the same item(ns). When running into a conflict situation, one of the candidate rules to be applied is chosen in a non-deterministic manner.

OBGG is a restricted form of graph grammars with respect to the kinds of vertices, as well as the configuration of rules to represent object-based concepts. This is done representing entities (objects types in object-based systems) that aggregate attributes and react to messages. Messages themselves may have attributes, called parameters. The behavior of an entity when reacting to a message is given by a (set of) rule(s). Therefore the left side of a rule always specifies a message being received by a specific entity. At the right side we have that message consumed and the effect of applying the rule, which may be: changing attributes; creating instances of entities, called objects; and generating new messages. This way, the application of a rule may leave the system state graph in a configuration where various other matches may occur. The system evolves until there is none possible match.

Analogously to an object-based system, which is composed by object types and objects, an OBGG specification is composed by the specification of different entities and its instances (objects). Each entity is defined separately, having a type graph (specifying attributes of the entity and messages it may receive), and a (set of) rule(s) that specify the entities behavior upon reception of a message. The specification of a system where various entities are involved is given by the specification of each separate entity complemented by the system initial graph that may have different objects and messages, addressed to these objects, used to start the model.

In OBGG there is no embedded notion of local or remote communications. However, a developer may specify minimum delays for messages. This notion allows us to express some differentiation between remote and local communications by assigning different minimum delay times. By default, all messages that do not have minimum delays specified are treated as having the lowest minimum delay possible. In the case study of Section 2.1 we have explicitly specified only the minimum remote communication times (*mrct*).

2.1 Pull-Based Failure Detector

Now we introduce a pull-based failure detector modeled with OBGG, as means to illustrate some described concepts. We also make this model the basis of our case study in Section 4. Every pull-based failure detector of the specification is modeled as a *PullDetector* entity. The type graph of this entity is depicted in Fig. 1. As it can be seen, some internal attributes of the *PullDetector* entity are abstract data types: *Map* is used to handle a collection of tuples; *List* is the implementation of a chained list. The *GS* entity is the group server for a particular group being monitored. Through *GS* the pull-based failure detectors can obtain references (via a *Search* message) to communicate with other participants of the monitored group. A detailed definition of the abstract data types *Map* and *List*, as well as the *GS* entity can be found in [20].

Fig. 2 shows part of the rules that define the behavior of the *PullDetector* entity. A *PullDetector* entity monitors the activity of a known group of *PullDetector* entities (attribute *group_server*) using liveness requisitions. The monitoring behavior is started at the beginning of the system, where a *Qliveness* message is sent to each *PullDetector* entity of the system (rule *ReqLiveness1*). A *Qliveness* message indicates the beginning of an interval of *d1* units of time. During this interval the other entities will be asked it they are alive. In order to do that, a *PullDetector* has an associated *GS* (group server) which records all the other *PullDetectors* of the group. A *PullDetector* then asks *GS* for all the other *PullDetectors* in the group (rule *ReqLiveness2*), and send messages *AYAlive* for each one (rule *AreYouAlive*). Note that this message *AYAlive* has an attribute *mrct*, used to indicate the minimum delay that a message for remote communication may have. In this example we have set *mrct* to 10 units of time. Responses to *AYAlive* messages, via *ImAlive* messages (rule *IamAlive*), are accepted until the interval *d1* has not expired. When an *ImAlive* message is received during

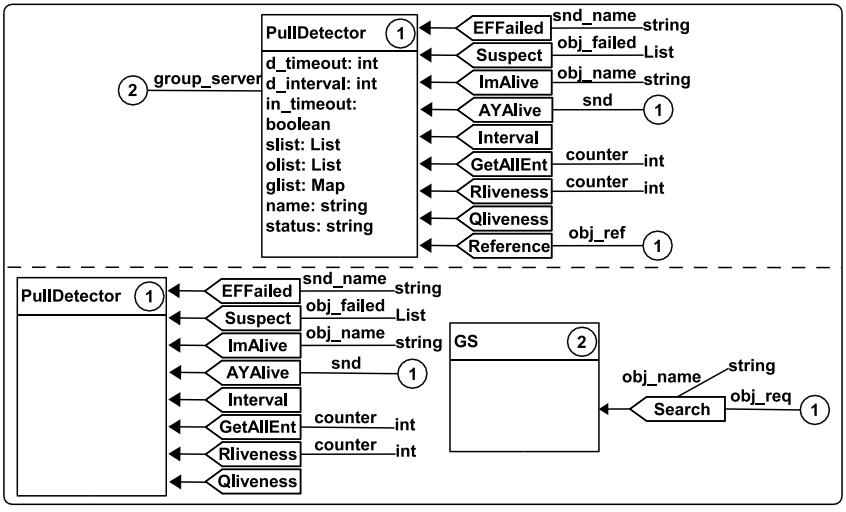


Fig. 1. Type graph of the *PullDetector* entity

the valid interval time, a *PullDetector* entity updates the situation of every entity that confirms its situation (rule *AliveReturn1*). This is done using a table (attribute *glist*) that contains the identification of every entity in the monitored group, and its current situation. Thus, the situation of each originator of the *ImAlive* message is updated to "ALIVE" in this table. Initially, every entity has the situation "UNKNOWN", meaning that there is no knowledge about their current state of activity. If the period for the response of monitored entities has finished, the *PullDetector* entity does nothing (rule *AliveReturn2*).

Fig. 3 presents the rest of the rules that define the behavior of the *PullDetector* entity. When the time interval for receiving *ImAlive* responses has finished a *PullDetector* starts a new monitoring cycle, in an interval of *d1* units of time, with a *Qliveness* message (rule *MntSuspList*). At the beginning of this pause the *PullDetector* entity mounts a local list of suspected entities, located in the *slist* attribute (rule *MntSuspList*). This list is composed by the identification of every entity that has the situation "UNKNOWN" in the *glist* table. After this local suspect list is mounted, the *PullDetector* entity sends this list to every entity on the monitored group, using a *Suspect* message (rules *SuspBroadcast* and *SendSuspect*). Besides, it sends a message *EFFailed* to every member of the monitored group.

Doing this, every entity in the monitored group receives (rule *ReceiveSuspect*) the local suspect list of the *PullDetector* that originated the *Suspect* message, and raises the union of this list with its own local suspect list, originating a unified list (without repeated entries). The message *EFFailed* is sent to every member of the monitored group in order to exclude the originator from the unified suspect list, located in the *olist* attribute (rule *ExFromFailed*). Since

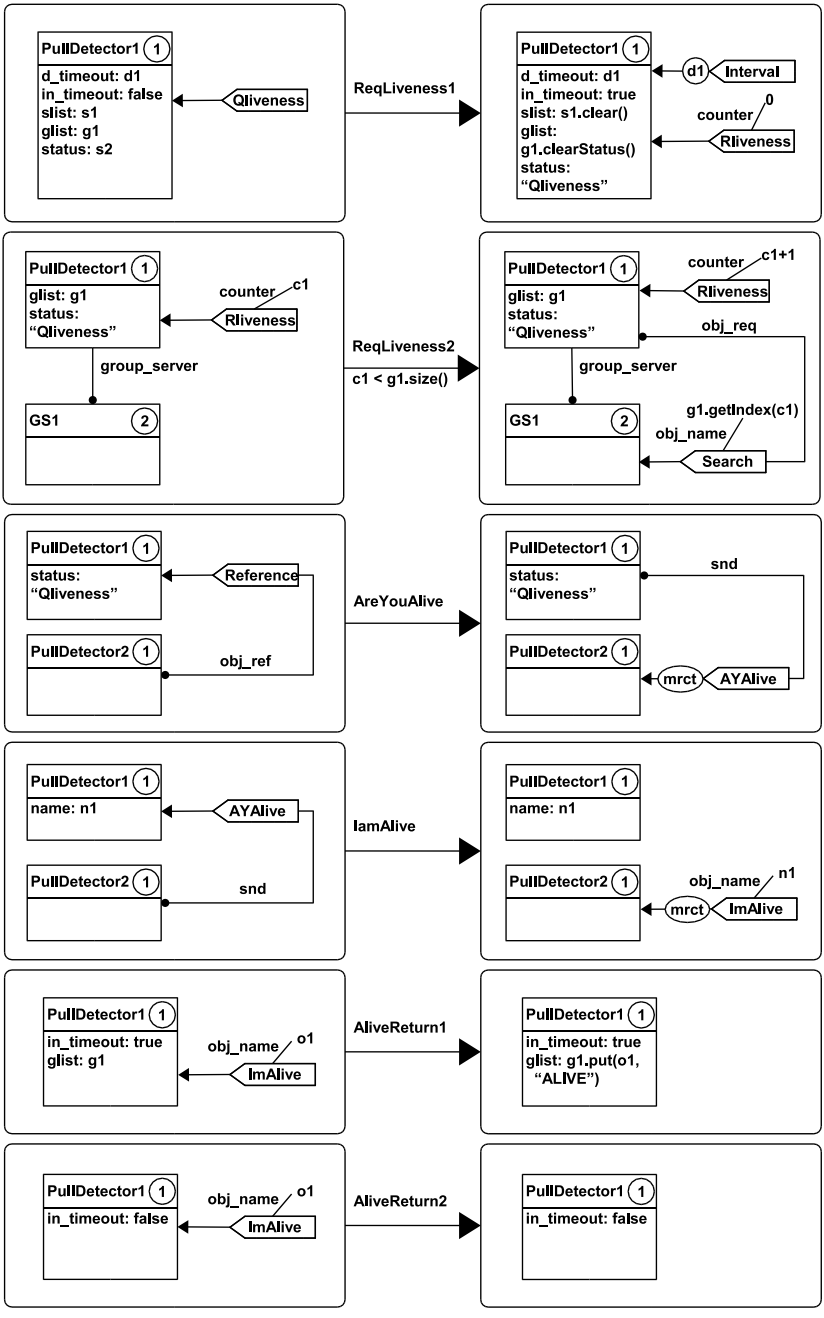


Fig. 2. Rules of the *PullDetector* entity

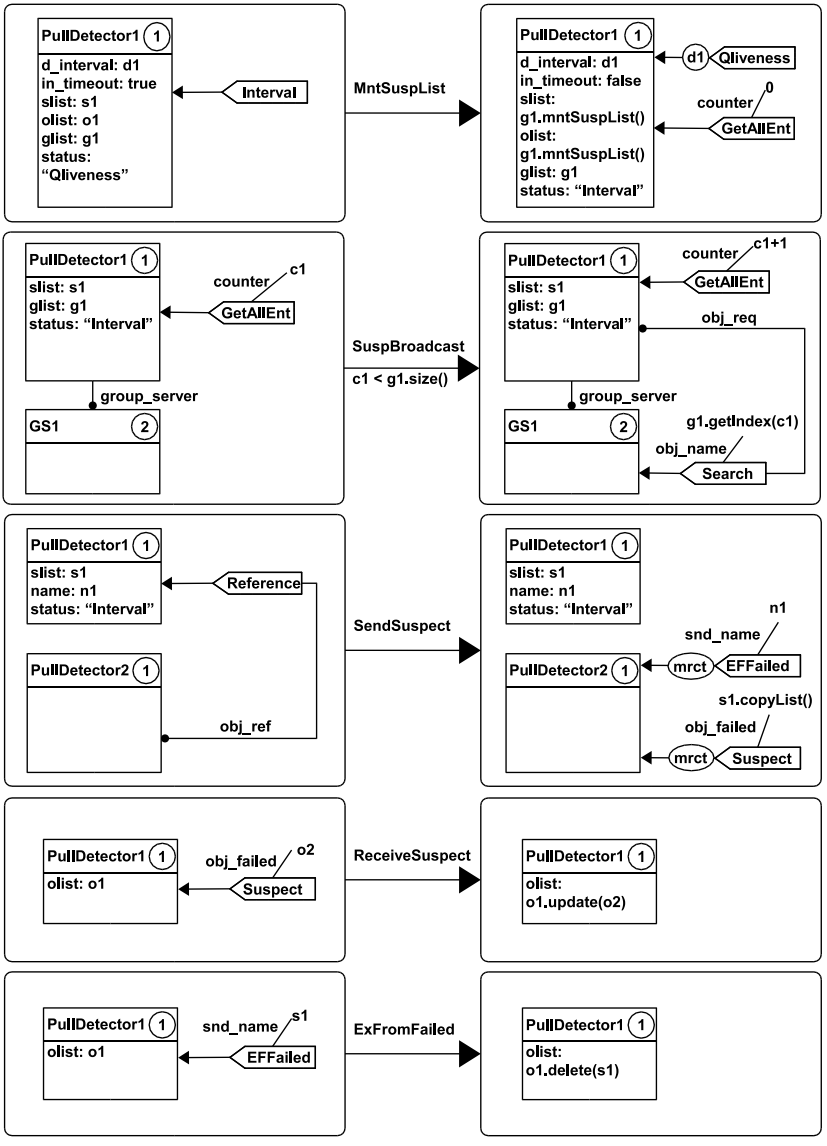


Fig. 3. Rules of the *PullDetector* entity (continued)

this monitoring process is realized by every member in the monitored group, the broadcast of the local suspect list by every member tends to form a unified suspect list. Once the delay period is over, a new liveness requisition is made, creating a new detection cycle.

In Fig. 4 is shown the initial graph of a scenario consisting of three instances of *PullDetector* entities. Every instance has a message *Qliveness* that starts the failure detection cycle. Instances of *GS* entities, used by *PullDetector* entities, are not fully described, due to space constraints and because they are not part of the main ideas of this paper (the full description of this entity is presented in [20]). Faults are not considered in this scenario, since the crash fault model is inserted and analyzed over this scenario in Section 4.

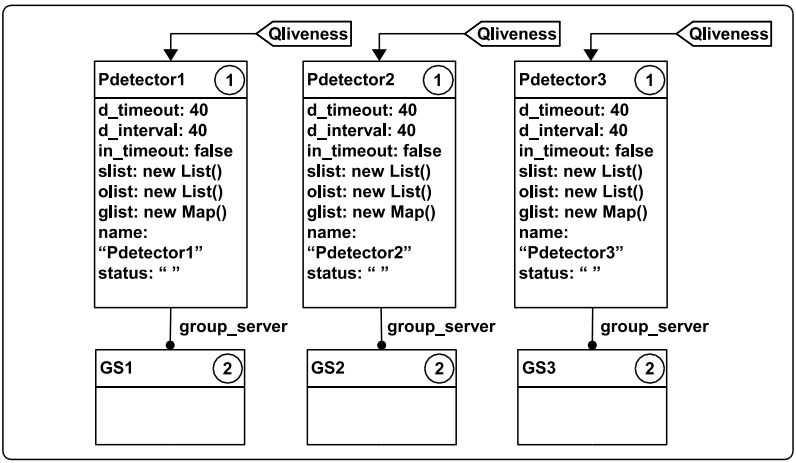


Fig. 4. Initial graph for PullDetector entities

3 Representation of Fault Models in Object Based Graph Grammars

In this Section we explain the methodology used to represent some classical fault models, found in the literature, using the OBGG formalism. After that, we present the fault models we have modeled in OBGG and how to use (insert) these selected models in an already defined specification of a desired system.

Traditionally terms like fault, error and failure are defined as in [17]. Here we try to avoid the terms error and failure, and adopt a formal definition, given in [12], for fault. As stated in [2] a system may change its state based on two event classes: normal system operation and fault occurrences. Based on this observation a fault can be modeled an unwanted (but possible) state transition of a system [12]. Thus a fault behavior of a system is just another kind of (programmable) behavior. These unwanted state transitions can be modeled through

the use of additional virtual[1] variables, acting like guards to activate specific commands (guarded commands). In this case, a group of guarded commands represents a specific fault model (the manner in which the component of the system will exhibit the fault behavior), being activated whenever its associated guard is satisfied, by the assignment of a true value to it [11].

The addition of virtual variables and the notion of guarded commands (presented above) can be viewed as a transformation of a model M_1 into a model M_2, that contains in its state space the behavior of a selected fault model [12]. As stated before, this transformation is done through the insertion of virtual variables and guarded commands, where each of these guarded commands, when activated, adds some particular fault behavior to the original model M_1. In a practical way, each guard corresponds to an additional virtual logical variable in the model M_1 that, when set to true, activates its respective guarded command, starting a specific fault behavior. When its value is set to false, the system acts exactly as the expected original behavior, that is, without the modeled fault behavior. The fault behavior is provided by additional code (guarded command) that disables some specific original function of the model M_1 and/or enables additional ones, in order to represent the intended fault model.

Here, we adopted the same concepts to transform a model M_1 into a model M_2 that represents a system with a selected fault model. Since our specification formalism supports implicit parallelism and is declarative, it is very suitable to represent guarded commands: the left side of a rule corresponds to the guard of the command; applying the rule transformation (according to the right side) corresponds to executing the guarded command. Thus we can model fault behaviors for an OBGG specification inserting virtual variables (used for guards) and messages (used to activate a fault behavior) in every entity of the model. Besides, we need to create and change the rules defined for entities that appear in the model. Depending on the fault model, different rule transformations occur. These transformations are explained in more detail in the next Sections.

For the selection of fault models to be modeled using our formalism we have adopted the fault classification found in [15]. There, fault models are classified in: crash, omission, timing, and Byzantine. More specifically, with respect to the omission model, we have used the classification found in [11], where the omission model is divided in: send omission, receive omission, and general omission. Thus, in the next Sections we present the modeling of the following fault models: crash, send omission, receive omission, and general omission. Besides, as already said, we show how to introduce a modeled fault behavior in an existing model of the system.

However, we have not modeled the timing and Byzantine fault models, which appear in the classification of [15]. In the timing fault model a process might respond too early or too late [15]. In the Byzantine fault model a process may assume an arbitrary behavior, even malicious [16]. In the future we intend to model the timing and Byzantine fault models, but due to their complexity we

[1] The term virtual is used to qualify variables that are not part of the desired system itself, but part of the introduced fault behavior.

have first concentrated our efforts in the modeling of the previous cited fault models, which are very often used in the fault tolerance literature (e.g. the crash fault model is commonly considered for distributed systems).

3.1 Crash Fault Model

In the crash fault model a process fails by halting. The processes that maintain communication with the halted process are not warned about the fault. In Fig. 5 is shown, in an algorithmic way, how to transform a model M_1 (without fault behavior) into a model M_2 that incorporates the behavior of a crash fault model.

```
1   For every entity in the model {
2     insert a boolean "down" variable in the type graph
3     set "down" to "false" in the initial graph (used as the guard)
4   }
5
6   For every entity in the model {
7     create a new rule called "EntCrash"
8       insert as the activation message a "Crash" message
9       this rule sets the guard "down" to "true"
10  }
11
12  For every entity in the model {
13    For all rules in the entity definition {
14      replicate the current rule and for each replica
15        insert a guard "down: true"
16        modify leaving only the participating entities whithout
17          any change to internal state or message generation
18    }
19  }
20
21  For every entity in the model {
22    For all original rules (not replicas) in the entity definition {
23      insert a guard "down: false"
24    }
25  }
```

Fig. 5. Algorithmic transformation over a model to represent the crash fault model

To add the behavior of a crash fault model (Fig. 5) the transformation procedure inserts (Lines 1 to 4) a virtual logical variable in every entity type graph of the model. Depending on the value of this variable, the entity may exhibit the fault behavior (*down* is true) or not (*down* is false). Besides, there is the creation of a new rule (Lines 6 to 10) that activates the fault behavior of an entity. This rule (*EntCrash*) is presented in Fig. 6.

As defined in Fig. 5, we have the creation of new rules (Lines 12 to 19), that are modified replicas of already defined rules, in order to represent the fault behavior once their guard is true (*down* is true). Besides, all rules of an entity are modified by the insertion of a guard (Lines 21 to 25). An example (using the rule *IamAlive* previously defined for the *PullDetector* entity) of these rule transformations is presented in Fig. 7.

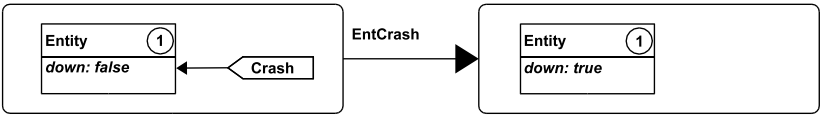


Fig. 6. Rule *EntCrash* used for the activation of the crash fault model

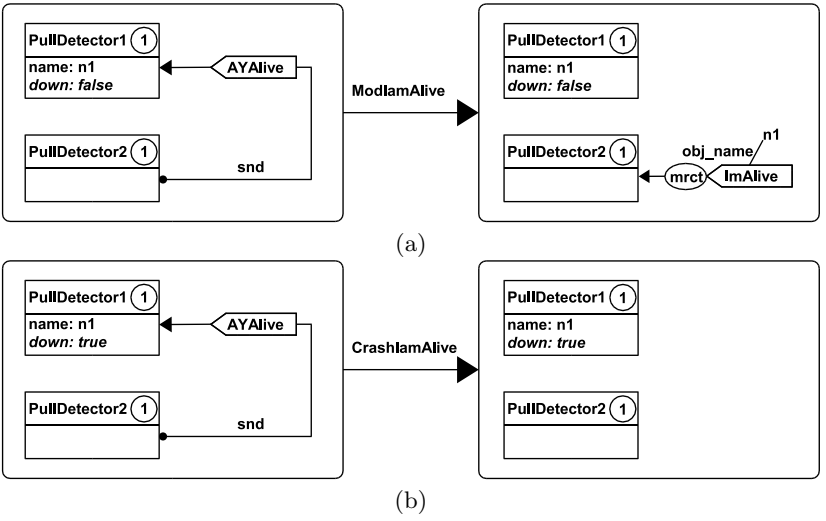


Fig. 7. Rule *IamAlive* without (a) and with (b) fault behavior

Fig. 7 (a) shows the normal behavior of the entity when the variable *down* is set to *false* (the fault behavior is not active). Once the fault behavior is activated, the rule defined in Fig. 7 (b) will occur. This rule simply consumes the message used to trigger the rule but does nothing (neither change the state of the entity nor create new messages). This way, we incorporate the behavior of a crash fault model.

3.2 Send Omission Fault Model

A process in a send omission fault may exhibit the same behavior as in a crash model. Furthermore, a process may fail to transmit a subset of the total messages it was supposed to send ([13] apud [11]). In Fig. 8 is shown, in an algorithmic way, how to transform a model M_1 (without fault behavior) into a model M_2 that incorporates the behavior of a send omission fault model.

To add the behavior of a send omission fault model (Fig. 8) the transformation procedure inserts (Lines 1 to 4) a virtual logical variable in every entity type graph of the model. Depending on the value of this variable, the entity may exhibit the fault behavior (*snd_omitted* is true) or not (*snd_omitted* is false).

```
1   For every entity in the model {
2     insert a boolean "snd_omitted" variable in the type graph
3     set "snd_omitted" to "false" in the initial graph (used as the guard)
4   }
5
6   For every entity in the model {
7     create a new rule called "EntSndOmission"
8       insert as the activation message a "SndOmit" message
9       this rule sets the guard "snd_omitted" to "true"
10  }
11
12  For every entity in the model {
13    For all rules in the entity definition {
14      replicate the current rule and for each replica
15        insert a guard "snd_omitted: true"
16        modify leaving only the participating entities whithout any
17          generation of messages but with the modification of the internal state
18    }
19  }
```

Fig. 8. Algorithmic transformation over a model to represent the send omission fault model

Besides, there is the creation of a new rule (Lines 6 to 10) that activates the fault behavior of an entity. This rule (*EntSndOmit*) is presented in Fig. 9.

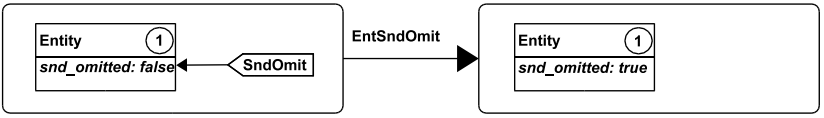


Fig. 9. Rule *EntSndOmit* used for the activation of the send omission fault model

As defined in Fig. 8, we have the creation of new rules (Lines 12 to 19), that are modified replicas of already defined rules, in order to represent the fault behavior once their guard is true (*snd_omitted* is true). The rules of an entity are not modified (guards are not inserted), since in the send omission fault model a process may fail to transmit a subset of messages. That is why we do not insert a guard on the original rules, leaving the choice of a rule to apply in a non-deterministic way (once the guard is *true*). An example (using the rule *IamAlive* previously defined for the *PullDetector* entity) of these rule transformations is presented in Fig. 10.

Fig. 10 (a) shows the normal behavior of the entity. Once the fault behavior is activated (*snd_omitted* is *true*), the rule defined in Fig. 10 (b) may also occur. Both rules of Fig. 10 (a) and (b) are enabled and one of them is non-deterministic chosen (according to the semantics of the formal specification language). The rule in Fig. 10 (b) simply consumes the message used to trigger the rule (changing

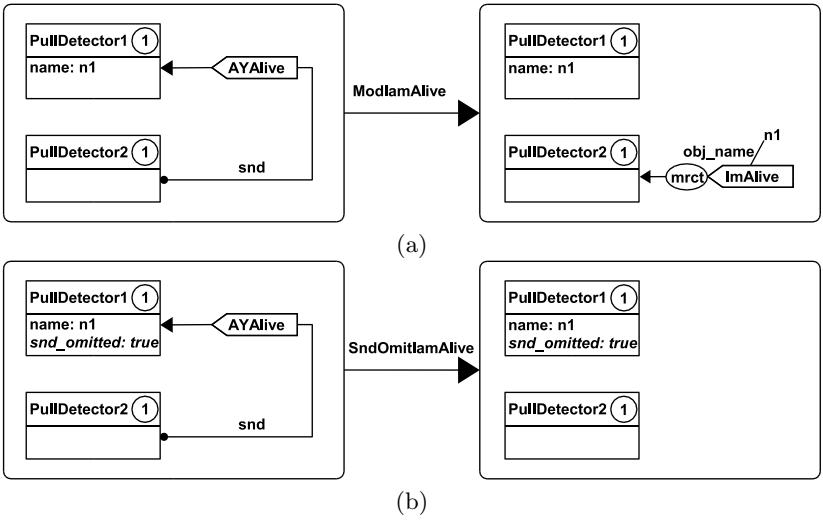


Fig. 10. Rule *IamAlive* without (a) and with (b) fault behavior

the internal state of the entity, if the rule specifies it) but does not create new messages, in this way we incorporate the behavior of a send omission fault model.

3.3 Receive Omission Fault Model

The receive omission fault model is analogous to the send omission fault model. The main difference is that in this fault model only a subset of the total messages sent to the fault process are received, differently from the send omission where only a subset of the total messages sent by the process are sent ([13] apud [11]). In Fig. 11 is shown, in an algorithmic way, how to transform a model M_1 (without fault behavior) into a model M_2 that incorporates the behavior of a receive omission fault model.

To add the behavior of a receive omission fault model (Fig. 11) the transformation procedure inserts (Lines 1 to 4) a virtual logical variable in every entity of the model. Depending on the value of this variable, the entity may exhibit the fault behavior (*rcv_omitted* is true) or not (*rcv_omitted* is false). Besides, there is the creation of a new rule (Lines 6 to 10) that activates the fault behavior of an entity. This rule (*EntRcvOmit*) is presented in Fig. 12.

As defined in Fig. 11, we have the creation of new rules (Lines 12 to 19), that are modified replicas of the already defined rules, in order to represent the fault behavior once their guard is true (*rcv_omitted* is true). The rules of an entity are not modified (guards are not inserted), since in the receive omission fault model a process may fail to receive a subset of messages. That is why we do not insert a guard on the original rules, leaving the choice of a rule to apply in a non-deterministic way (once the guard is *true*). An example (using the rule *IamAlive*

```
1   For every entity in the model {
2     insert a boolean "rcv_omitted" variable in the type graph
3     set "rcv_omitted" to "false" in the initial graph (used as the guard)
4   }
5
6   For every entity in the model {
7     create a new rule called "EntRcvOmission"
8       insert as the activation message a "RcvOmit" message
9       this rule sets the guard "rcv_omitted" to "true"
10  }
11
12  For every entity in the model {
13    For all rules in the entity definition {
14      replicate the current rule and for each replica
15        insert a guard "rcv_omitted: true"
16        modify leaving only the participating entities whithout
17          any change to internal state or message generation
18    }
19  }
```

Fig. 11. Algorithmic transformation over a model to represent the receive omission fault model

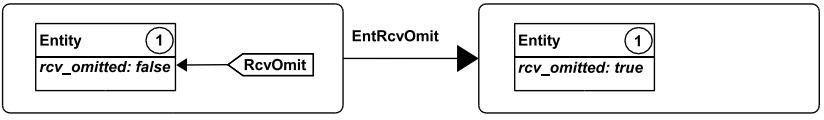


Fig. 12. Rule *EntRcvOmit* used for the activation of the receive omission fault model

previously defined for the *PullDetector* entity) of these rule transformations is presented in Fig. 13.

Fig. 13 (a) shows the normal behavior of the entity. Once the fault behavior is activated (*rcv_omitted* is *true*), the rule defined in Fig. 13 (b) may also occur. Both rules of Fig. 13 (a) and (b) are enabled and one of them is non-deterministic chosen (according to the semantics of OBGG). The rule in Fig. 13 (b) simply consumes the message used to trigger the rule but does nothing to the entity (neither change the state of the entity nor create new messages), in this way we incorporate the behavior of a receive omission fault model.

3.4 General Omission Fault Model

The general omission fault model specifies that a process may experience both send and receive omissions ([18] apud [11]). Using these concepts we model the general omission model as a merge of the previous defined send and receive omission fault models. Therefore, we have to apply both algorithmic transformations for send and receive omission models. By using this mechanism we have semantically composition of both defined models.

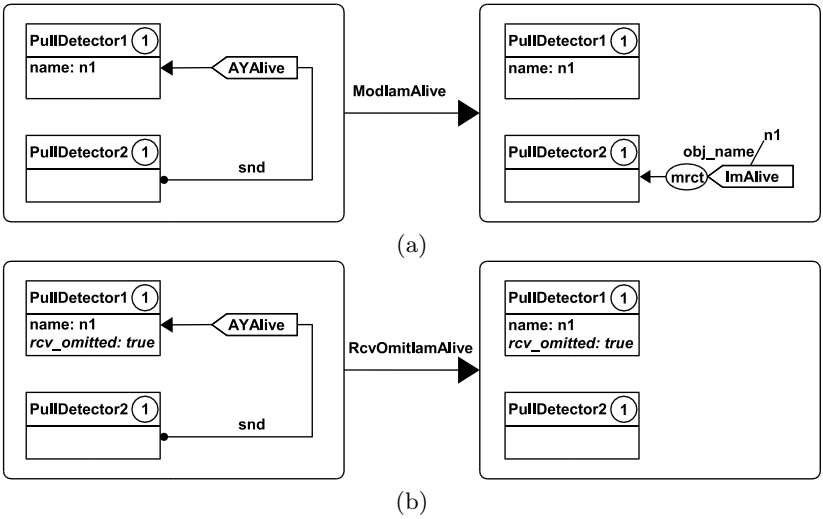


Fig. 13. Rule *IamAlive* without (a) and with (b) fault behavior

4 Analysis of the Pull-Based Failure Detector

In this Section we exemplify the introduction of fault models described in previous Sections using the example of a pull-based failure detector defined in Section 2.1. We have applied the algorithmic transformation for the crash fault model, defined in Section 3.1, in order to reason about the example in the presence of crash faults. The initial graph of the system was changed, in order to include a temporized *Crash* message, used for the activation of the crash fault behavior in the *Pdetector2* entity of the scenario. Two of three *Qliveness* messages (used for the activation of entities) in the initial graph have also been delayed, such that entities do not start the detection cycle at the same time (just for the purposes of graphical analysis in Fig. 15).

The transformation of the *PullDetector* model, presented in Section 2.1 considering the crash fault model, generated 23 rules out of the 11 original ones. From these 23 rules, 11 describes the behavior of the model without crash behavior, other 11 rules describe the behavior of a crashed pull detector, and 1 rule serves for the activation of the crash behavior. We do not show in this Section the modified rules of the *PullDetector* entity due to space constraints, and because we have exemplified how to transform rules in previous Sections.

As shown in Fig. 14, after 110 units of time the *PullDetector2* entity will receive a *Crash* message, that will activate the crash fault behavior of it. This means that the *Pdetector2* entity will halt, assuming a behavior where neither messages will be sent nor received, and its state will not change.

In the execution of this scenario both entities *Pdetector1* and *Pdetector3* will detect that *Pdetector2* has failed, and will put *Pdetector2* in their unified suspect

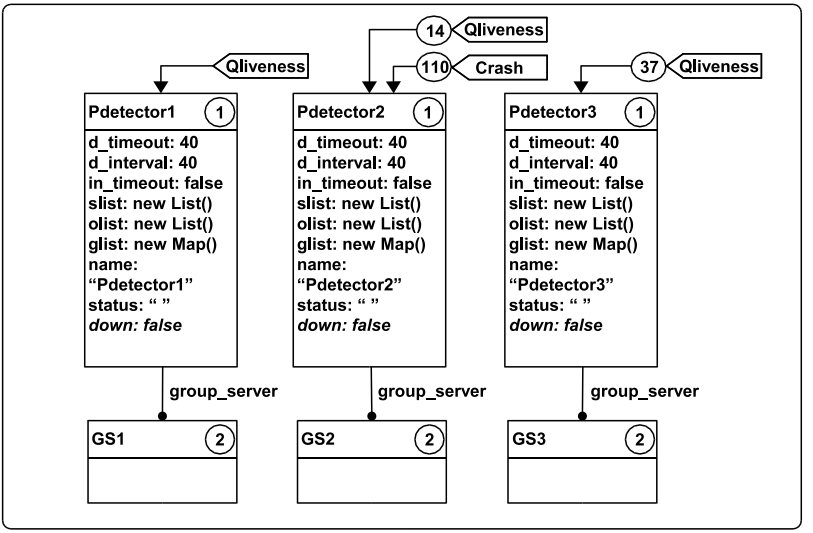


Fig. 14. Initial graph for *PullDetector* entities considering a crash fault model

list (*olist* variable). As presented in Section 1, currently we have, as means to analyze the behavior of OBGG specifications, a simulation tool. We use this tool to generate a log of an execution of the model. Based on this log we can plot a graphic (Fig. 15) that shows the messages sent between the *PullDetector* entities, that is, messages used for remote communication.

For the better understanding of Fig. 15 the reader should remember that the names of the generated messages are plotted near their originating entities. Every time that a *PullDetector* entity starts a new detection cycle, the contents of its unified suspect list is printed on the right side of the Figure. The numbers printed at the left side of the Figure correspond to the sending time of messages. The time of the reception of messages are not printed.

As it can be seen in Fig. 15 from time 0 to 110, each *PullDetector* monitors other entities and receive responses, resulting in empty unified suspect lists, as expected in a situation without faults. At time 110 *Pdetector2* crashes and stops processing any incomming messages. This can be seen in the further received messages which are marked as lost. As a result, both *Pdetector1* and *Pdetector3* eventually consider *Pdetector2* as a suspect in their unified suspect lists.

5 Related Works

Concerning related works, we have surveyed the literature trying to identify approaches that allow developers to reason about distributed systems in the presence of faults.

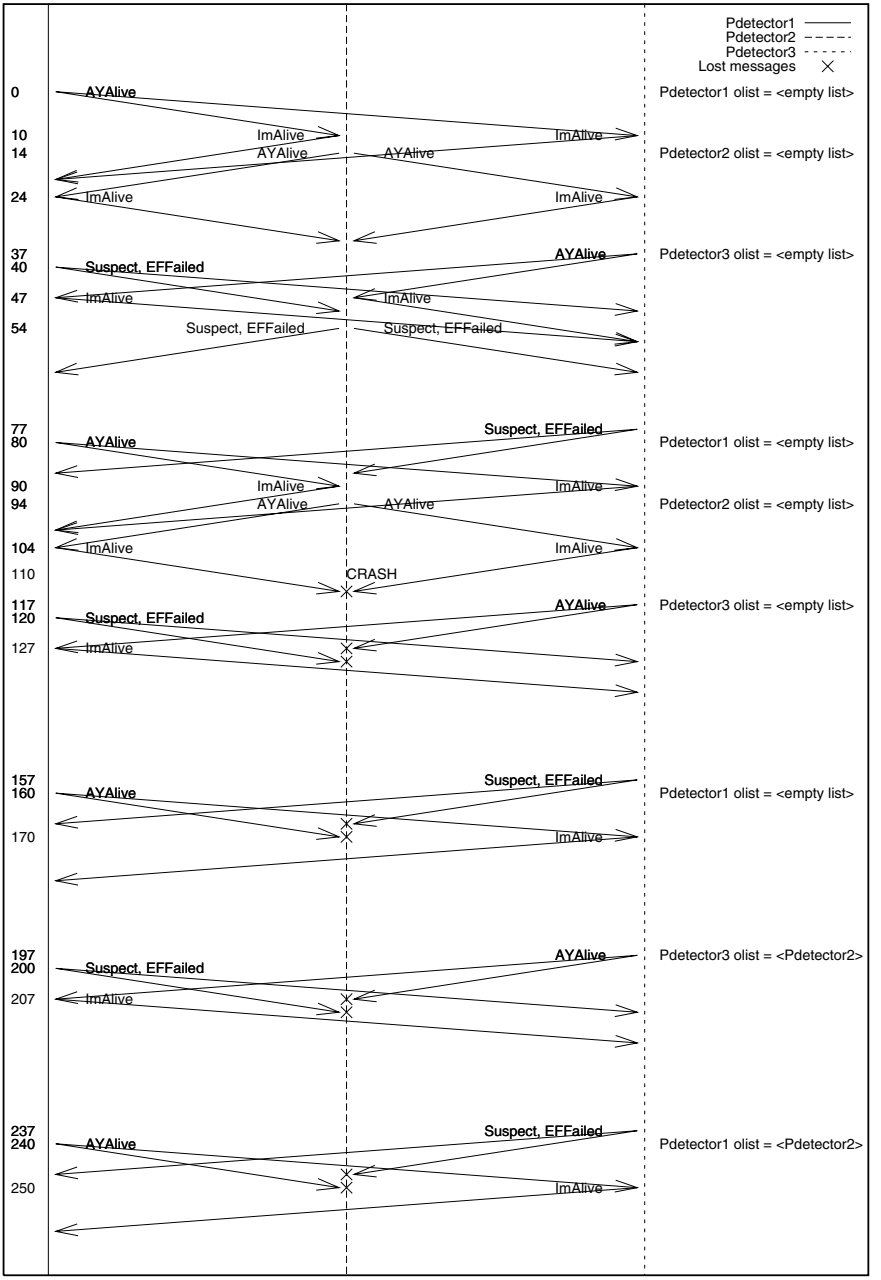


Fig. 15. Graphic generated from the log of an execution

In [19] an approach dealing with fault injection at the testing phase is proposed. In order to do that, an implementation of the system must be available. Code representing the desired fault is then inserted in the system, which is then executed and the results analyzed. Another work [21] proposes a technique, called script-driven probing and fault injection, for studying the behavior of distributed systems. There, some classical fault models can be used to probe the behavior of a protocol being developed. Again, an implementation of the system must be available in order to use the technique.

The main difference of the previous two works referred, compared to this one, is that here we allow the developer to reason about the system in an earlier phase, during specification. Another difference is that here we use a formal specification language to represent the system and the fault behavior.

The SPIN (Simple Promela INterpreter) model checker [14] enables a developer to specify a system using the formal specification language PROMELA (PROcess/PROtocol MEta LAnguage), and specify properties using linear temporal logic (LTL). The similarity is that SPIN allows one to verify models using channel abstractions that may loose messages. This feature is provided by the supporting tool. Other fault behaviors are not provided. SPIN thus provides message losses through the run-time environment. Faults are not represented as part of the system model. Since in this work we embed the fault behavior in the system model, we can use different tools to analyze the effects of the fault in the model. In this paper we used a simulation tool. Currently we are also mapping the formal specification language to a model checking environment [6] and this shall allow one to verify models in the presence of faults too.

Another work, which is directed to the development of mobile systems, is presented in [10]. The Distributed Join Calculus is a process calculus that introduces the notions of locations and agents, where locations may be nested. The calculus provides the notion of crash of a location, whereby all locations and agents in the crashed location are halted. A crash of a location changes the behavior of basic characteristics hindering any kind of reaction, like for communication or process migration. Systems that are represented with this calculus can be verified using theorem proving and thus requiring better skilled developers.

6 Final Remarks

In this paper we have presented an approach for the specification and analysis of distributed systems considering the presence of faults. We have presented the definitions of crash and omission fault models, the last one being further specialized in send, receive and general omission. Moreover, we have showed how to combine the behavior of a given system model M_1 and one of the defined fault models, achieving a model M_2. Then, model M_2 can be used to reason about the behavior of the original model M_1 in the presence of the considered fault.

The formal specification language used to represent models of distributed systems, as well as to describe fault models, is a restricted form of graph grammars,

called Object Based Graph Grammars (OBGG). The fact that this formalism is declarative and supports the notions of implicit parallelism and non-determinism resulted that the transformations from M_1 to M_2 were rather simple to define (see Section 3).

Since models described in OBGG can be simulated, we have a method and a supporting tool to help reasoning about distributed systems in the presence of faults. Simulation can be used as means for testing the behavior of models in the presence of faults, as well as for performance analysis. Because we aim to prove properties over defined model, we are working on an approach for model checking OBGG specifications [6]. This approach is based on the mapping of OBGG specifications to the input language of a model checker. As a further step we intend to analyze the suitability of verifying models in the presence of selected fault behaviors.

References

1. B. Copstein, M. C. Móra, and L. Ribeiro. An environment for formal modeling and simulation of control systems. In *33rd Annual Simulation Symposium*, pages 74–82, USA, 2000. IEEE Computer Society Press.
2. F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, 11(1):23–31, 1985.
3. D. Dèharbe, A. M. Moreira, L. Ribeiro, and V. M. Rodrigues. Introduction to formal methods: specification, semantics and verification of concurrent systems (in portuguese). *Revista de Informática Teórica e Aplicada – UFRGS*, 7(1):7–48, 2000.
4. F. L. Dotti, L. M. Duarte, B. Copstein, and L. Ribeiro. Simulation of mobile applications. In *Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 261–267, USA, 2002. The Society for Modeling and Simulation International.
5. F. L. Dotti, L. M. Duarte, F. A. Silva, and A. S. Andrade. A framework for supporting the development of correct mobile applications based on graph grammars. In *6th World Conference on Integrated Design & Process Technology*, pages 1–9, USA, 2002. Society for Design and Process Science.
6. F. L. Dotti, L. Foss, L. Ribeiro, and O. M. Santos. Formal specification and verification of distributed systems (in portuguese). In *17th Brazilian Symposium on Software Engineering (accepted for publication)*, 2003.
7. F. L. Dotti and L. Ribeiro. Specification of mobile code systems using graph grammars. In *4th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 177 of *IFIP Conference Proceedings*, pages 45–63, USA, 2000. Kluwer.
8. L. M. Duarte. Development of distributed systems with mobile code using formal specifications (in portuguese). Master's thesis, PUCRS – Faculdade de Informática – PPGCC, Brazil, 2001.
9. H. Ehrig. Introduction to the algebraic theory of graph grammars. In *1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69, Germany, 1979. Springer.
10. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory*, volume 1119 of *LNCS*, pages 406–421, Italy, 1996. Springer.

11. F. C. Gärtner. Specifications for fault tolerance: a comedy of failures. Technical Report TUD-BS-1998-03, Darmstadt University of Technology, Department of Computer Science, Germany, 1998.
12. F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, 1999.
13. V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, USA, 1994.
14. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
15. P. Jalote. *Fault tolerance in distributed systems*, pages 51–53. Prentice Hall, USA, 1994.
16. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
17. J.-C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *15th International Symposium on Fault-Tolerant Computing*, pages 2–11, USA, 1985. IEEE Computer Society Press.
18. K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, 1986.
19. A. P. Mathur R. A. DeMillo, T. Li. Architecture of tamer: a tool for dependability analysis of distributed fault-tolerant systems. Technical Report SERC-TR-158-P, Department of Computer Sciences – Purdue University, 1994.
20. E. T. Rödel. Formal modeling of faults in distributed systems with mobile code (in portuguese). Master's thesis, PUCRS – Faculdade de Informática – PPGCC, Brazil, 2003.
21. F. Jahanian S. Dawson. Probing and fault injection of protocol implementations. In *15th International Conference on Distributed Computing Systems*, pages 351–359, Canada, 1995. IEEE Computer Society Press.
22. F. A. Silva. *A transaction model based on mobile agents.* PhD thesis, Technical University Berlin – FB Informatik, Germany, 1999.

Panel "Dependability Benchmarks: Can We Rely on Them?"

João Gabriel Silva

DEI/CISUC
University of Coimbra
3030-290 Coimbra - Portugal
jgabriel@dei.uc.pt

If users had benchmarks to easily measure dependability of computer based systems, they would use them when buying. The industry would certainly respond by producing better systems, similarly to what we have witnessed before in the performance and database areas, and everyone would benefit.

Unfortunately dependability benchmarks are quite complex. Among the challenges that must be addressed are incorporating the effects of software bugs and system administrators errors, characterizing COTS hardware and software components, and taking into account that different application areas have different dependability requirements.

This panel aims at answering a very simple question: is the state of the art in the very active research area of dependability benchmarking sufficiently advanced, for the end user to be able to rely on the dependability benchmarks that are available now or expected in the near future?

The issues to address include the following:

- Is the representativity of the benchmarks, particularly of the injected faults, well established?
- What is the error to expect in the results of the benchmarks?
- Can the results be easily generalizable to other configurations that do not exactly correspond to those used in the benchmarks themselves?
- Are the benchmarks results repeatable, or can minor changes e.g. in the hardware or software of the tested system, or the fault profile injected, lead to wide variations in the results?
- What is the level of technical sophistication that a user of dependability benchmarks must have?
- Can the benefit perceived by the end-users be sufficiently high for the benchmarks to get wide adoption?
- Is it realistic to expect that the required industry endorsement of the benchmarks will happen?
- Are all application areas equally amenable to dependability benchmarking, from OLTP to embedded safety critical systems?

It seems clear that much progress has been done in recent years in many of these issues, but also that a lot of open questions still remain. It is not yet clear when the first successful dependability benchmark will emerge.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, p. 361, 2003.

Workshop on Safety: Computer Systems in Critical Applications

João Batista Camargo Júnior[1] and Rogério de Lemos[2]

[1] Computer and Digital Systems Engineering Department
Escola Politécnica - Universidade de São Paulo
Av. Prof. Luciano Gualberto, Trav. 3 – N. 158
05508-900 – São Paulo – SP, Brazil
{joao.camargo@poli.usp.br}
[2]Computing Laboratory
University of Kent
Canterbury, Kent CT2 7HH, UK
{r.delemos@ukc.ac.uk}

Commercial pressures for producing high-quality computer systems in a variety of applications and covering a wide range of complexities, hazards and potential risks are always increasing. The most challenging aspect in the design of computer systems for such applications is to ensure system safety in the presence of component failures.

Based on these considerations, the objective of this Workshop is to discuss the specification, verification, validation and certification of computer systems in several critical applications, including: aviation, nuclear, mass transportation, medical and petrochemical. For encouraging debates and exchange of experiences on the development and usage of computer systems in many areas where human lives might be at risk, each critical application will be presented in the context of a practical case study. Industries and research institutions that have been involved in the development, certification and deployment of safety-critical systems will be presenting these case studies. In the aviation sector, three organizations will be discussing the application of computers in different areas. *Atech Tecnologias Críticas* will discuss the application of computers in air traffic control systems. *Embraer* will present the certification of critical software to be used in commercial aircraft. Finally, *IFI/CTA* will discuss the process of software certification from the point of view of the certification authority. The use of computer systems in nuclear applications will be presented by *CTMSP*, and their presentation will be focusing on the requirements specification process. *Alstom* will present the case study on mass transportation, and their focus will be on the development process of computer systems when applied to signaling systems. The problems related to the certification of software in medical equipment will be presented by *INCOR*. Finally, *Petrobrás* will discuss the use of computer systems in the context of petrochemical plants, by emphasizing the special case of leakage detection and operation control.

At the end of the Workshop, an open discussion will be organized for synthesizing the wide range of experiences, promote the cooperation between industry and academia, and establish a future agenda for the area. We believe that the mixed audience of practitioners and researchers will provide an ideal forum for discussing the current state of the area of safety-critical systems in Brazil, and provide the basis for promoting its future development.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, p. 362, 2003.

Second Workshop on Theses and Dissertations in Dependable Computing

Avelino Zorzo[1], Francisco Brasileiro[2], and Ingrid Jansch-Pôrto[3]

[1] Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
zorzo@inf.pucrs.br

[2] Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
fubica@dsc.ufcg.edu.br

[3] Universidade Federal do Rio Grande do Sul
Instituto de Informática
ingrid@inf.ufrgs.br

The Workshop on Theses and Dissertations in Dependable Computing is a student forum that aims to gather graduate students developing research on topics related to dependable computing.

The first edition of the workshop was held in 2001, at Florianópolis, Brazil, in conjunction with the Brazilian Symposium on Fault Tolerance. Since this symposium is the originator of the Latin American Symposium on Dependable Computing (LADC), it seemed natural to hold the second edition of the workshop in conjunction with the LADC 2003.

At the forum, students have the opportunity to present and discuss the objectives of their research, preliminary results, and future directions of their work. The forum is an excellent opportunity for them to get acquaintance with other research groups, to learn more about projects being developed, and to receive precious feedback on their own research.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, p. 363, 2003.

Development of Safety-Critical Systems and Model-Based Risk Analysis with UML

Jan Jürjens[1⋆] and Siv Hilde Houmb[2⋆⋆]

[1] Software & Systems Engineering, Computer Science, TU Munich, Germany
[2] Norwegian University of Science and Technology

The high quality development of safety-critical systems is difficult. Many safety-critical systems are developed, deployed, and used that do not satisfy their criticality requirements, sometimes with spectacular failures. Part of the difficulty of safety-critical systems development is that correctness is often in conflict with cost. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided.

UML offers an unprecedented opportunity for high-quality safety-critical systems development that is feasible in an industrial context.

- As the de-facto standard in industrial modeling, a large number of developers is trained in UML.
- Compared to previous notations with a user community of comparable size, UML is relatively precisely defined.
- A number of tools are developed to assist working with UML.

However, there are some challenges one has to overcome to exploit this opportunity, which include the following:

- Adaptating UML to critical system application domains.
- Advancing the correct use of UML in the application domains.
- Dealing with conflicts between flexibility and unambiguity in UML.
- Improving tool-support for critical systems development with UML.

The tutorial aims to give background knowledge on using UML for critical systems development and to contribute to overcoming these challenges.

In particular, we consider model-based risk analysis, where we use an integrate system development and risk analysis process, in order to address critical issues at the right time for the right price. The process integrates the Australian standard for risk management AS/NZS 4360:1999, result from the IST-project CORAS, RUP, and RM-ODP, combined with a UML extension for secure systems development, UMLsec. We address risks both at an enterprise and technical level, and which can be analyzed mechanically by referring to a precise semantics of the used fragment of UML. For critical systems, identifying possible failures or vulnerabilities is particularly difficult, as they usually involve subtle and complex dependencies (and sometimes the consideration of domain-specific concepts such as cryptography and random numbers).

⋆ http://www4.in.tum.de/~juerjens
⋆⋆ siv.hilde.houmb@idi.ntnu.no

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 364–365, 2003.

Outline. The tutorial presents the current academic research and industrial best practice by addressing the following main subtopics:

- UML basics, including extension mechanisms.
- Model-based risk analysis.
- Developing safety- and security-critical systems with UML.
- Extensions of UML (UML-RT, UMLsec, ...)
- Using UML as a formal design technique for critical systems.
- Case studies.
- Tool-support for developing safety-critical systems using UML, in particular: Using the standard model interchange formats (XMI) for tool integration and to connect to validation engines.

The tutorial includes a demo of a prototypical tool for the analysis of UML models for critical requirements and a tool to support the process of model-based risk analysis, which is based on XMI.

History. The proposed tutorial is a sequel to a highly successful series of about twenty tutorials presented at international conferences (see http://www4.in.tum.de/~juerjens/csdumltut). It also builds on the material in [Jür02,Jür03a,Jür03b,HJ03,Jür03c].

References

[HJ03] S. Houmb and J. Jürjens. Developing secure networked web-based systems using model-based risk assessment and UMLsec, 2003. Submitted.

[Jür02] J. Jürjens. Critical Systems Development with UML. In *Forum on Design Languages*, Marseille, Sept. 24–27 2002. European Electronic Chips & Systems design Initiative (ECSI). Invited talk.

[Jür03a] J. Jürjens. Developing safety- and security-critical systems with UML. In *DARP workshop*, Loughborough, May 7–8 2003. Invited talk.

[Jür03b] J. Jürjens. Developing safety-critical systems with UML. In P. Stevens, editor, *UML 2003 – The Unified Modeling Language*, Lecture Notes in Computer Science, San Francisco, Oct. 20–24 2003. Springer-Verlag, Berlin. 6th International Conference.

[Jür03c] J. Jürjens. *Secure Systems Development with UML.* Springer-Verlag, Berlin, 2003. In preparation.

On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial

Idit Keidar[1] and Sergio Rajsbaum[2]

[1] Dept. of Electrical Engineering, Technion, Haifa, 32000, Israel.
[2] Instituto de Matemáticas, UNAM; Ciudad Universitaria; 04510 México D.F.

Abstract. We consider the consensus problem in realistic partial synchrony and timed asynchronous models where processes can crash. We describe algorithms and lower bounds that show that two communication steps are necessary and sufficient for solving consensus in these models in failure-free executions.

1 Motivation

Consensus is a fundamental problem in distributed computing theory and practice alike. A consensus service allows a collection of processes to agree upon a common value. More specifically, each process has an input, and each correct process must decide on an output, such that all correct processes decide on the same output, and furthermore, this output is the input of one of the processes. Consensus is an important building block for fault-tolerant distributed systems [1]; to achieve fault-tolerance, data is replicated, and consensus can be used to guarantee replica consistency using the *state machine* approach [2,3].

Consensus is not solvable in pure asynchronous models where even one process can crash [4][1]. However, real systems are not completely asynchronous. The timed asynchronous model [5] and some partial synchrony models [6] better model real systems. We consider a particularly realistic kind of model [6] that allows the system to be asynchronous for an unbounded but finite period of time, as long as it eventually becomes synchronous, and less than a majority of the processes can crash. Consensus is solvable in this model [6]. Similarly, consensus is solvable in an asynchronous model enriched with certain oracle failure detectors [7], including unreliable failure detectors that can provide arbitrary output for an arbitrary period of time, but eventually provide some useful semantics. In this tutorial, we discuss consensus algorithms in such *non-synchronous* models. Algorithms that solve consensus in such models are sometimes called *indulgent* [8].

What can we say about the running time of consensus algorithms for non-synchronous models? Unfortunately, even in the absence of failures, consensus

[1] Consensus can be solved in the asynchronous model by randomized algorithms, and in some shared memory models. In this tutorial, we consider only message-passing, deterministic algorithms.

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 366–368, 2003.

algorithms in these models are bound to have unbounded running times. This is because, by [4], in an asynchronous system, a consensus algorithm can have an infinite run in which no failures occur, and non-synchronous systems can be asynchronous for an unbounded period of time. In practice however, there are often extensive periods during which communication is timely. That is, many executions are actually synchronous. In such executions, failure detectors based on time-outs can be accurate. In this tutorial, we study the running time of consensus algorithms for non-synchronous models under such benign circumstances. We focus on executions in which, from the very beginning, the network is synchronous or the failure detector is accurate. We will call such executions *well-behaved* if they are also failure-free[2]. Since well-behaved executions are common in practice, algorithm performance under such circumstances is significant. Note, however, that an algorithm cannot know a priori that an execution is going to be well-behaved, and thus cannot rely upon it.

There are several algorithms for non-synchronous models that decide in two communication steps in well-behaved executions (e.g., [9,10,11,12]). Distributed systems folklore suggests that every fault tolerant algorithm in non-synchronous models must take at least two communication steps before all the processes decide, even in well-behaved executions. We formalize this folklore theorem in this tutorial. Specifically, we show that any consensus algorithm for non-synchronous models where at least two processes can crash will have at least one well-behaved execution in which it takes two communication steps for all the processes to decide. We also present algorithms that achieve this bound.

2 Outline

The tutorial is structured as follows:

Part I: Preliminary introduction – consensus and models of computation and complexity;
Part II: Realistic timing model and metric – asynchronous, synchronous, partial synchrony, and timed asynchronous models;
Part III: Upper bounds – consensus algorithms and the failure detector abstraction;
Part IV: Lower bounds – the lower bound: a simple proof of the uniform consensus synchronous lower bound and its implications to non-synchronous models;
Part V: New directions and extensions.

We have presented a tutorial covering similar material at the 21st ACM Symposium on Principles of Distributed Computing (PODC), July 2002. A detailed early version of this tutorial appears in [13]; the lower bound of Part IV appears in [14]; and [15] discusses some related open problems.

[2] Our notion of well-behaved behaved executions is similar to the notion of *stable* periods, e.g., in the timed-asynchronous model [5] and in self-stabilizing systems.

3 Assumed Background of Audience

Basic knowledge of the field of distributed computing, at the level of an introductory text book.

References

1. B. Lampson. How to build a highly available system using consensus. In Babaoğlu and Marzullo, editors, *Distributed Algorithms*, LNCS 1151. Springer-Verlag, 1996.
2. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.
3. F.B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
4. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.
5. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.
6. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
7. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
8. R. Guerraoui. Indulgent algorithms. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 289–298, July 2000.
9. A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.
10. M. Hufrin and M. Raynal. Asimple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4), 1999.
11. A. Mostefaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: a general approach. In *13th International Symposium on DIStributed Computing (DISC)*, Bratislava, Slovak Republic, 1999.
12. I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.
13. I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults – a tutorial. Technical Report MIT-LCS-TR-821, MIT Laboratory for Computer Science, May 2001. Apreliminary version in SIGACT News 32(2), Distributed Computing column, pages 45–63, June 2001 (published May 15th, 2001).
14. I. Keidar and S. Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 85(1):47–52, January 2003.
15. I. Keidar and S. Rajsbaum. Open questions on consensus performance in wellbehaved runs. In *Future Directions in Distributed Computing*, volume 2584 of LNCS, pages 35–39, Bertinoro, Italy, June 2003. Springer Verlag.

A Practical Approach to Quality Assurance in Critical Systems

Carlos A.T. Moura[1], Carlos H.N. Lahoz[2], and Martha A.D. Abdala[2]

[1] Centro de Lançamento de Alcântara, Rod. MA-106, km. 7
65.250-000 - Alcântara, MA, Brasil
catmoura@uol.com.br
[2] Instituto de Aeronáutica e Espaço, AEL-C, CTA
12.228-904 - S. José dos Campos, SP, Brasil
{lahoz, martha}@iae.cta.br

Software is safe when it works in the context of a greater system without contributing to hazard conditions. Several factors should be considered to achieve that condition, specially the use of software engineering good practices for development and maintenance. Among those we can emphasize the adoption of standards and processes of software quality assurance and configuration management.

This tutorial discusses the use of some of those practices, as the project structuring based on standards, the verification and validation tests using a simulation environment and the change management, showing the case of the On-Board Software (SOAB) development for the Brazilian Satellite Launch Vehicle (VLS).

The main purpose is to stimulate the discussion of suitable approaches for the development of software for critical systems, among professionals and others interested in this area, by means of practical experiences exchanging. Those information and know-how are believed to be applicable also in other Software Engineering domains and should be useful for software developers' education.

Although quality has been emphasized in software development, trying to correct errors in the final product is not sufficient to assure quality. The adherence to the Quality Assurance (QA) management practices that should be enforced in the organization requires the definition and the improvement of development processes, which is essential to achieve dependable products.

Several accidents are reportedly related to software changes, originated by modifications in external systems, internal subsystems, or due to unintentional ones. So, change control is of paramount importance for critical systems. Knowing that changes are unavoidable, a software configuration management (SCM) process is required to: establish a baseline for the software items in a system; control changes and items versions; register and inform items status and change demands; and control items storage, handling, and delivering.

Our case of study, SOAB, is considered safety critical because all the vehicle control system depends on it, and a malfunction may cause mission failure or severe acci

R. de Lemos et al. (Eds.): LADC 2003, LNCS 2847, pp. 369–370, 2003.

dents. As the VLS project required four prototypes to qualify the vehicle, system evolutions and changes in components and subsystems are expected, requiring review in many outputs of the SOAB development cycle. Recently, e.g., the development team has implemented the changes for the third flight.

The mini-tutorial presents the main characteristics of the project organization and the QA and SCM approaches, considered essential to attain the system requirements.

The DoD-STD-2167A standard was chosen as a basis to the software development. It works as an instrument to delineate and register the project activities, establishing requirements for the development, acquisition, and support of the software. Thus many standardized documents were selected, tailored, and described to support the project development and control.

Safety Engineering uses several techniques to assess a risk as acceptable or not. In this mini-tutorial, we emphasize the validation of general and safety requirements using tests. This approach is based on the concept of software safety as a quality factor. To conduct SOAB formal qualification tests, it was used an environmental simulator originally constructed to test VLS aerodynamic performance models. It could be verified that the implementation of a system's environmental simulator carries benefits to the assessment of a critical system, because it is impracticable to execute all tests in the real conditions of the operational environment.

SCM adopted the four classic activities: Identification, Control, Audits, and Accounting. Nevertheless, special attention is paid to the following instruments: Software Development Library and Tools; and Configuration Control Board.

Besides the presentation of QA and SCM practical results, some discussions are developed on the applicability of those approaches in safety critical projects. Specially on the trade-offs between the overhead of documents and rigorous processes established by conservative standards, and the necessary tailoring required to make them executable processes, mainly when considering small developing teams.

International standards like ISO/IEC 12207 describe software development and maintenance processes. Although the life cycle is not specified, it presents a basic set of activities to obtain the software product. Some others present models as reference for the organizational maturity levels, as ISO/IEC 15504 and CMMI, which assists in the verification of the maturity level of development processes and practices. Those standards and models are presented in order to identify the best quality practices to define, improve, and evaluate the software development processes.

Also, it is presented a brief overview of environments that support project development activities, providing a variety of services, such as automation of routine tasks, invocation and control of development tools, and enforcement of mandatory rules and practices.

Author Index