Performance Studies of Fault-Tolerant Middleware

Diana Szentiványi
Abstract

Today’s software engineering and application development trend is to take advantage of reusable software. Much effort is directed towards easing the task of developing complex, distributed, network based applications with reusable components. To ease the task of the distributed systems’ developers, one can use middleware, i.e. a software layer between the operating system and the application, which handles distribution transparently. A crucial feature of distributed server applications is high availability. This implies that they must be able to continue activity even in presence of crashes. Embedding fault tolerance mechanisms in the middleware on top of which the application is running, offers the potential to reduce application code size thereby reducing developer effort. Also, outage times due to server crashes can be reduced, as failover is taken care of automatically by middleware. However, a trade-off is involved: during periods with no failures, as information has to be collected for the automatic failover, client requests are serviced with higher latency. To characterize the real benefits of middleware, this trade-off needs to be studied. Unfortunately, to this date, few trade-off studies involving middleware that supports fault tolerance with application to realistic cases have been conducted. The contributions of the thesis are twofold: (1) insights based on empirical studies and (2) a theoretical analysis of components in a middleware equipped with fault tolerance mechanisms.

In connection with part (1) the thesis describes detailed implementation of two platforms based on CORBA (Common Object Request Broker Architecture) with fault tolerance capabilities: one built by following the FT-CORBA standard, where only application failures are taken care of, and a second obtained by implementing an algorithm that ensures uniform treatment of infrastructure and application failures. Based on empirical studies of the availability/performance trade-off, several insights were gained, including the benefits and drawbacks of the two infrastructures. The studies were performed using a realistic (telecommunication) application
set up to run on top of both extended middleware platforms. Further, the thesis proposes a technique to improve performance in the FT-CORBA based middleware by exploiting application knowledge; to enrich application code with fault tolerance mechanisms we use aspect-oriented programming. In connection with part (2) the thesis models elements of an FT-CORBA like architecture mathematically, in particular by using queuing theory. The model is then used to study the relation between different parameters. This provides the means to configure one middleware parameter, namely the checkpointing interval, leading to maximal availability or minimal response time.

This work has been supported by the European Project TRANSORG in the IST initiative EUTIST-AMI, the FP6 IST project DeDiSys on Dependable Distributed Systems, and by CENIIT (Center for Industrial Information Technology) at Linköping University.
Acknowledgments

First and foremost I would like to thank my advisor Simin Nadjm-Tehrani for always challenging me to find new ideas. Without her this work would not have been accomplished. By letting me on my own feet in the last two years she contributed to my growing and my self-knowledge. I also want to thank my co-advisor Petru Eles for reading this thesis and giving valuable feedback. I thank John M. Noble, our collaborator from the Mathematics Department, for being a real help in the work with the checkpointing interval optimization. Without him, that work could not have been done. It was really rewarding to work with mathematics in a real-life context.

I want to thank Nick Szirbik, over and over again, for proposing me the idea of doing PhD studies abroad. Since the summer of 1998 my life changed completely.

My colleagues in RTSLAB offered a nice working environment with nice discussions. I especially thank Anne Moe for always sorting out difficult administrative questions in an efficient way. It will be difficult to leave them behind.

My thanks go to Johan Moe for priceless discussions around the platform implementations and evaluations. I thank Isabelle Ravot for implementing the FACORBA platform during her visit in RTSLAB in 2002.

I thank Torbjörn Örtengren from Ericsson Radio AB for providing us the telecom application. My colleague Călin Curescu was the one to prepare the application for usage in our experiments. Thank you!

I also want to thank Lillemor Wallgren and Britt-Inger Karlsson for helping me with the administration around the thesis defense.

This work has been supported by the European Commission IST initiative,
Project TRANSORG that is included in the cluster of projects EUTIST-AMI on Agents and Middleware Technologies applied in real industrial environments, by the FP6 IST project DeDiSys on Dependable Distributed Systems, and by CENIIT (Center for Industrial Information Technology) at Linköping University.

I am endlessly grateful to my wonderful friends who, nearby or far away, always think about me and await me to come home. I would not be complete without them.

I dedicate this work to my mother and late father, and to my one and only Saşa.

Diana Szentiványi
# Contents

1 Introduction ........................................ 1
   1.1 Motivation and overview ......................... 3
   1.2 Problem description ............................... 5
   1.3 Contributions .................................. 6
   1.4 Publications .................................... 7
   1.5 Thesis outline .................................. 8

2 Terminology ......................................... 11
   2.1 Faults and failure models ....................... 11
      2.1.1 The notions of fault, error, and failure ...... 12
      2.1.2 Failure models ................................ 13
      2.1.3 Timing models ................................ 14
      2.1.4 Unreliable failure detectors ................. 16
   2.2 Achieving fault tolerance ....................... 17
      2.2.1 Software fault tolerance .................... 17
      2.2.2 Replication strategies ...................... 18
      2.2.3 State and checkpointing .................... 19
   2.3 Communication in fault-tolerant distributed systems .... 20
      2.3.1 Broadcasts .................................. 20
      2.3.2 Message ordering ............................ 21
   2.4 Consensus as a basic primitive ................. 21
   2.5 Queuing theory ................................ 22
      2.5.1 Random variables ........................... 23
      2.5.2 Queues and related notions ................. 24
## Fault tolerance and middleware

### 3.1 Basic algorithms for consensus
- **3.1.1 Consensus in synchronous systems**
- **3.1.2 Consensus in asynchronous systems**

### 3.2 Variants of unreliable failure detectors

### 3.3 Perfect failure detectors

### 3.4 The process group abstraction
- **3.4.1 Group services**
- **3.4.2 Specification and implementation of group services**
- **3.4.3 Fault-tolerant group based platforms**

### 3.5 Fault tolerance and commercial middleware

### 3.6 Assessing performance

### 3.7 Assessing availability

## Two fault-tolerant CORBA implementations

### 4.1 The FT-CORBA infrastructure
- **4.1.1 The standard**
- **4.1.2 Failure model and assumptions**
- **4.1.3 Infrastructure building blocks**
- **4.1.4 Logging and failover mechanism**

### 4.2 The FA-CORBA infrastructure
- **4.2.1 Architecture units**
- **4.2.2 Infrastructure interactions**
- **4.2.3 FA-CORBA implementation**

### 4.3 Related work

## Empirical studies with a telecom application

### 5.1 Experiments with the telecom application
- **5.1.1 The telecom application**
- **5.1.2 Experiment setup**
- **5.1.3 Measuring roundtrip time overheads**

### 5.2 Results
- **5.2.1 Overheads**
- **5.2.2 Failover times**

### 5.3 Lessons learnt

### 5.4 Reflections on the FT-CORBA infrastructure
6 Improving performance of fault-tolerant software 85
   6.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
   6.2 Basic AOP concepts . . . . . . . . . . . . . . . . . . . . . . . . 87
   6.3 Aspects for fault tolerance mechanisms . . . . . . . . . . . . .89
   6.4 Implementation issues . . . . . . . . . . . . . . . . . . . . . . . 93
      6.4.1 Aspects defined perflow . . . . . . . . . . . . . . . . . . 95
      6.4.2 Method execution related advices . . . . . . . . . . . . . 95
      6.4.3 Field level synchronization advices . . . . . . . . . . . . 97
   6.5 Evaluation of the approach . . . . . . . . . . . . . . . . . . . . 98
   6.6 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
   6.7 AOP and middleware . . . . . . . . . . . . . . . . . . . . . . . . 103
   6.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

7 Computing the optimal checkpointing interval 107
   7.1 Related work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
   7.2 The checkpointing procedure . . . . . . . . . . . . . . . . . . . . 112
   7.3 Basic model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
   7.4 Modelling assumptions . . . . . . . . . . . . . . . . . . . . . . . 116
   7.5 Queuing analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
   7.6 Optimal checkpointing for maximum availability . . . . . . . . 118
   7.7 Optimal checkpointing for minimum response time . . . . . . . . 124
      7.7.1 New assumptions . . . . . . . . . . . . . . . . . . . . . . . 125
      7.7.2 Minimizing average response time . . . . . . . . . . . . . 126
   7.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136

8 Analysis of the optimization models 139
   8.1 Numerical studies . . . . . . . . . . . . . . . . . . . . . . . . . . . 140
      8.1.1 Relating response time to request arrival rate . . . . . . . 143
      8.1.2 Relating availability to request arrival rate . . . . . . . . 146
      8.1.3 Relating checkpoint arrival rate to request arrival rate . . 147
      8.1.4 Relating checkpointing interval to request arrival rate . . 149
      8.1.5 Availability maximization and failure rate . . . . . . . . . 152
      8.1.6 Response time minimization and failure rate . . . . . . . . 155
      8.1.7 Average availability-average response time trade-offs . . . 155
   8.2 Simulations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
      8.2.1 Validation approach . . . . . . . . . . . . . . . . . . . . . 159
      8.2.2 Validation results . . . . . . . . . . . . . . . . . . . . . . 162
9 Conclusions and future work

9.1 Concluding remarks .............................................. 169
9.2 Future work ......................................................... 172
List of Figures

3.1 State transition diagram example ................................... 39
4.1 Deployment of the FT-CORBA infrastructure ...................... 46
4.2 Illustration of the double role of the active replication gateway . 48
4.3 Checkpointing and logging mechanism ............................... 50
4.4 Object state portions .................................................. 55
4.5 FA-CORBA Server and Client infrastructure units ................. 58
4.6 Client-Server interactions in the FA-CORBA infrastructure ....... 59
4.7 Client-Server message sequence chart when no failures occur ... 61
4.8 Client-Server message sequence chart under failure suspicions . 64
5.1 Roundtrip time overheads illustration ............................... 75
5.2 Bird’s eye view of minimum and maximum overhead percentages 78
5.3 Time spent in consensus for group sizes of 2, 3, 4, 8 replicas .... 79
5.4 Average method replay and average update application times (in ms) 81
5.5 Variation with group size of the average time to apply an update . 81
6.1 Two method calls that access non-independent parts of the state .. 87
6.2 Aspect weaving ......................................................... 88
6.3 Advices, join points, pointcuts ....................................... 89
6.4 “Introduction” in AspectJ ............................................. 90
6.5 Client-Server interactions in a FT-CORBA primary-backup setting 91
6.6 Client-Server interactions in an improved primary-backup setting 92
6.7 Gray-box description of the server application object ............ 94
6.8 Example of an aspect defined percflow ............................ 96
6.9 An “around” advice .................................................... 97
6.10 Performance improvements in a cold-passive setting ............... 100
6.11 Performance improvements in a warm-passive setting ........... 101
### 6.12 Changed method_a

- Page 101

### 6.13 Field modification impossible to capture by pointcuts

- Page 102

### 7.1 The logging and application servers and their relation

- Page 113

### 7.2 The checkpointing procedure times slices

- Page 114

### 7.3 Servers and queues

- Page 115

### 7.4 Relation between different random variables

- Page 119

### 7.5 Equilibrium, failover and backlog processing

- Page 125

### 7.6 Failures and checkpointing

- Page 126

### 8.1 Minimal average response time vs. load

- Page 144

### 8.2 Load sensitivity of the average response time when system configured for minimal response time

- Page 145

### 8.3 Minimal average response time and with no checkpointing vs. load

- Page 146

### 8.4 Maximal average availability vs. load

- Page 147

### 8.5 Optimizing average checkpoint rate vs. load

- Page 148

### 8.6 Average checkpoint rate minimizing response time and backlog vs. load

- Page 150

### 8.7 Maximizing checkpointing interval and duration vs. load

- Page 151

### 8.8 Maximizing checkpointing interval and duration vs. load

- Page 152

### 8.9 Maximizing average checkpoint rate vs. failure rate

- Page 153

### 8.10 Average number of calls to replay and maximum average availability vs. failure rate

- Page 154

### 8.11 Minimizing average checkpoint rate vs. failure rate

- Page 156

### 8.12 Minimal average response time vs. failure rate

- Page 156

### 8.13 Minimizing and maximizing average checkpoint rate vs. load

- Page 157

### 8.14 Average availabilities vs. load for high failure rates

- Page 158

### 8.15 Average availabilities vs. load for low failure rates

- Page 159

### 8.16 Average response times vs. load

- Page 160

### 8.17 Average availabilities from mathematical and simulation model vs. load

- Page 162

### 8.18 Average availability for different average checkpoint rates vs. load

- Page 162

### 8.19 Average availability from simulation model for different checkpointing intervals vs. load

- Page 163

### 8.20 Average availability for different checkpointing intervals vs. load

- Page 164
Chapter 1

Introduction

The trend in today’s software engineering and application development is to reuse pieces of software [138]. A lot of effort is made to devise easy ways for developing complex, distributed, network based applications with reusable components. Since the late 80s, when the term *middleware* was first used in the context of database systems, the idea of interposing a software layer between hardware and operating system and application, has evolved so that in the future programs will not be needed to build applications, but to integrate and configure reusable components. Thus, the same middleware can be used to develop several different applications. The *maintainability* of the system becomes easier this way. Here maintainability is used with its meaning from software engineering: the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment. Bakken [11] in his article with the same title defines middleware as “a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems”. Middleware is thus a piece of complex software that can encapsulate interoperability features, as well as means to ease development of distributed applications. Several categories of middleware exist: middleware that hides communication details (“distributed object middleware” as called by Bakken), or that allows applications to access a database system, or that provides messaging and other services, e.g. agent platforms. Middlewares in the area of context-aware computing, wireless networks, ad-hoc networks [7, 120] are becoming more and more popular, witnessed by the first workshop on Middleware for Pervasive and Ad-Hoc computing organized in 2003. Another trend in the usage of middlewares is to make them adaptive, thus be able to change their own configuration during runtime. One way of doing
this is by using reflection[84, 32]. Reflection is achieved by making the middleware itself component-based, and thus built of reusable software. The keynote speech at the 2003 Middleware conference [54] praised Aspect-Oriented Programming [80] as a means to reconfigure middlewares themselves, adapting them to build different adaptable applications [53, 163].

Built on top of a middleware or not, an important aspect of an application is its capacity to provide service to users even in presence of a failure. Thus, what application developers should strive for, besides respecting functional requirements in their system, as well as time-to-market requirements, is to provide a highly dependable system. Dependability is a concept with a large number of attributes; these themselves are titles of different research fields. A short definition of this notion is as follows: dependability is the property of a computer system such that reliance can justifiably be placed on the service it delivers [46]. An alternate complementary definition for dependability was introduced in 2001 by Avizienis et al. [5]. It involves the levels of non-trusted service acceptable to a user: dependability is the ability of a system to avoid failures that are more severe or more frequent, and to avoid outage durations longer than is acceptable to a user. Attributes of dependability are: reliability, availability, safety, integrity, confidentiality, and maintainability.

Reliability describes the property of a system to continuously provide correct service ([5]). Safety is the property of a system that assures the absence of catastrophic consequences on the environment ([5]). Maintainability is the property of a system expressing the ability to undergo repairs and modifications ([46, 5]). According to Verissimo [158] maintainability is in fact measured by the time needed by a system to recover from a failure.

Availability is the attribute of dependability that describes the readiness for correct service of the system [5]. This thesis will mainly focus on this aspect of dependability. The means for achieving dependability are related to dealing with the faults in the system: fault prevention, fault tolerance, fault removal, fault forecasting. In particular, fault tolerance, increases the capacity of a system to continue providing its services at the same quality, or at some reduced, acceptable quality, even in presence of faults. According to Fischer [51] a fault-tolerant system is one that operates in presence of all anticipated faults, i.e. those grouped in the fault model of the system. To implement fault tolerance besides error detection mechanisms, some redundancy of resources is needed. A frequently used mechanism is to replicate processing resources.

The importance of having standardized platforms to build available applications is emphasized by the existence of The Service Availability™ Forum. The Forum
is a coalition of the world’s premier communications and computing companies working together to create and promote open, standard interface specifications[55]. Since 2002 this organization has published among others the “The Service Availability Forum Platform Interface” and “The Service Availability Forum Application Interface Specification”.

The cost of fault tolerance in terms of extra resources like (processing) time, processing power and hardware is an interesting issue that needs attention, and thus is worth studying. Also, it is appealing to strive for offering application writers possibilities to develop fault-tolerant distributed applications with low effort. This gives a promising direction for using middleware for purposes other than for interoperability. Middleware could encapsulate support for different dependability attributes such as security aspects or availability and fault tolerance aspects.

The work in this thesis answers questions that the application developer might ask when being provided with a middleware that hides partially or totally the fault tolerance handling details from the application. When we say totally hides, we mean that all mechanisms for handling logging, failure detection and failover are encapsulated in the middleware code. Partially hides, on the other hand, means that, in order to use application knowledge in a better way, some mechanisms are moved to the application level. However, this still happens with no or very little application developer intervention. The context in which the results of our studies are provided consists of soft real-time applications. This means that it is desirable that the roundtrip time for servicing a client request is within given limits. However, no catastrophe will happen in case the client occasionally does not receive the answer within the given interval. The request is simply resent in case of delays. The studies convey information about average response times, and if those are not exceeding a certain limit in a replicated scenario then we deem the studied configuration as a possible choice for deploying one’s application. The work in this thesis is therefore different, for example, from the one presented by Pinho et al. [119] that studies a framework for building reliable (hard) real-time distributed systems. In such a framework there is a separation between the soft and the hard real-time parts of an application, and higher reliability is ensured for the hard real-time parts.

1.1 Motivation and overview

Some researchers see the requirement for fault tolerance as a part of the specification of the system, i.e. derived from availability, safety, integrity, etc. [97, 9]. Thus, there is need for a system to be fault-tolerant because it must be available,
safe and able to defend against altering of data by intruders. Others see the notion of fault tolerance related to distribution. On one hand distribution itself is seen as a motivation for fault tolerance: some server (on some machine in the network) providing services to clients has to be available whenever the clients (possibly on a different machine) ask for its service. On the other hand, distribution is a way of achieving fault tolerance: running services on several machines is better than running all services on one single machine that is failure prone [125]. However, by and large, one can say that early works on fault tolerance either did not have distribution as a basic premise or had strong synchrony requirements on a system’s components. The distinguishing factor between various works is often the emphasis on hardware or software fault tolerance [130, 100, 143, 35, 12], and whether the fault tolerance technique is employed in a particular setting, e.g. databases [14] or disk storage [67].

There is a need for standardizing the way of building systems that combine fault tolerance features with other, e.g. performance properties [161]. By using a middleware, distributed applications can be created such that different computers run code perhaps written in different programming languages, on possibly different operating system platforms. If assuring fault tolerance is not among the roles of the “gluing” instrument, then application writers have to incorporate it in each application that requires it. Thus, from an application development point of view, this is a waste of resources. Even if the fault tolerance mechanisms in different applications are not exactly the same they contain several common components that bring in the potential for reuse and automatic code generation. Furthermore, with support in middleware, the size of application code can be reduced, thereby offering better maintainability of the system. To make informed decisions on how much support for fault tolerance to include in an existing middleware, or what mechanisms this support should employ we need to thoroughly analyze these questions, both in an experimental and a theoretical setting. This thesis has major contributions towards these goals.

There are some drawbacks in incorporating mechanisms dealing with logging, failure detection and failover totally in the middleware. This way, application knowledge is not exploited at all or is very difficult to exploit. Motivated by this background, in our work we study ways of lifting mechanisms from middleware to application level, but with very little application developer intervention. An elegant way of including non-functional portions in an application with little application developer contribution, is aspect-oriented programming (AOP). By using application knowledge in still a restricted way, performance might be improved. A similar
case for improving maintainability by using software-implemented fault tolerance is presented by Deconinck et al. [40]. The framework presented includes a library with fault tolerance functions, middleware to coordinate their usage as well as a language to express non-functional services.

Our industrial partner Ericsson Radio Systems AB develop distributed telecom applications that have to be, among others, fault-tolerant and easy to develop. Therefore, a platform to encapsulate fault tolerance aspects is appealing. Besides, there is little performance measurement data in the literature, involving realistic applications. These aspects motivated us to build experimental platforms and evaluate them by empirical approaches using a telecom application provided to by Ericsson. Such measurements give valuable data on merits of such platforms. However, empirical studies do not offer a very clear picture about how different parameters in the system influence each other. This motivated us to employ mathematical analysis to better understand such complex relations.

Returning to the specific platform implementations, a point also taken into account when choosing the type of middleware, is to ensure interoperability features for applications built for different operating systems and language platforms. The CORBA (Common Object Request Broker Architecture) standard devised by the OMG (Object Management Group) is widely accepted and successfully implementable, therefore its extensions such as FT-CORBA (Fault-Tolerant CORBA) or RT-CORBA (Real-Time CORBA) can be seen as starting points for creating a unified framework for future multi-feature applications. Therefore, we were strongly motivated to consider CORBA as the middleware to work with in our empirical studies.

1.2 Problem description

By hiding different aspects of an application, specifically handling of fault tolerance, in the middleware, a developer could be reluctant to use such a platform. And this is the case especially when one became familiar with a middleware that suddenly is augmented with a totally new and perhaps complex feature. Questions that could arise in one’s mind are performance, as well as security related, to mention just a few. In this thesis we will mainly focus on performance questions:

- How much longer must the client wait for an answer from a server running on top of a middleware with fault tolerance features incorporated, compared to talking to a non-fault-tolerant server? In other words, what are the overheads
while the server is up and running?

- How large is the outage time of a server after it fails, when this situation is solved automatically by middleware mechanisms?

- What parameters of the middleware and the application (e.g. replication style, number of replicas) are to influence overheads and outage times? How should the application developer configure parameters of the middleware?

- How can performance penalties be minimized by better using application knowledge?

- At what extent mathematical modelling can help in devising parameter configurations to offer maximum availability or minimum request response time to a service user? How restrictive a model has to be in order to allow reasonable computations leading to detailed enough results?

To sum up, this thesis discusses trade-offs between sacrificing performance (in terms of timing) and high availability. It will reveal details about how restrictive must the failure model be in order to allow a reasonably sized fault-tolerant infrastructure to be built. To what extent the implementation of a middleware extended with fault tolerance features (FT infrastructure) can influence the resulting trade-offs, will be illustrated on two implementations.

### 1.3 Contributions

Real-life industrial applications such as telecommunication systems are naturally distributed. Due to the complexity of design and implementation tasks, the object paradigm is a good choice. CORBA is a very popular middleware that allows objects to be distributed on different platforms and to be written in different programming languages. In this context, the contributions of this thesis work are as follows:

- We describe a detailed implementation of two platforms based on CORBA with fault tolerance capabilities: one built by following the FT-CORBA standard, and a second obtained by implementing an algorithm that devises full availability; both platforms are available as open source [128].

- We compare, by empirical studies, the two implementations of the CORBA based middleware that incorporates fault tolerance mechanisms;
setting up a realistic (telecommunication) application on top of both extended middleware infrastructures; evaluating and comparing the performance of the application on top of both middlewares;

– trying to find out the influence of different parameters on the timing values;

– comparing and contrasting the benefits and drawbacks of each infrastructure.

- We propose a technique to improve performance in the FT-CORBA based middleware by exploiting application knowledge; to enrich application code with fault tolerance mechanisms we use Aspect-oriented Programming.

- We study by mathematical modelling using queuing theory, the relation between different parameters of a middleware that resembles the FT-CORBA implementation. We focus on determining the checkpointing interval that maximizes availability respectively minimizes response time. To our knowledge, this is the first time a detailed mathematical model is used for such optimization purposes.

For the last two items in the above list, we immediately claim that the results are generalizable to other middlewares than CORBA.

In what the first two items are concerned we can make the following remark. The main reason for building the FT-CORBA platform was to provide feasibility analysis of implementing the FT-CORBA standard and performance analysis. At the time of performing the work, no FT standard ORB was readily available for further performance study. However, even in this case we claim that with low effort the methodology used and the obtained results can be generalized as well.

1.4 Publications

This thesis is an extended version of a book chapter and research papers published in international conferences. These are:

Chapter 1. Introduction


Diana Szentiványi, Simin Nadjm-Tehrani and John M. Noble: Minimizing Response Time by Efficient Checkpointing. Submitted for publication.

1.5 Thesis outline

The thesis is divided in nine chapters, as follows:

- **Chapter 2** presents basic terminology. It introduces basic notions such as asynchrony, failure detectors, consensus, reliable broadcast, queuing theory. These notions form the theoretical basis for the later sections of the document.

- **Chapter 3** presents some research results in the areas of fault tolerance, availability, and reliability. These are aimed to give an overview of the research problems that arise beyond the core notions that are presented in Chapter 2.

- **Chapter 4** describes implementations of two middleware platforms incorporating fault tolerance handling features: one that closely follows the FT-CORBA specification, and one that departs from that standard, but offers more robustness by incorporating unreliable failure detectors.

- **Chapter 5** describes experimental results reflecting availability-performance trade-offs, obtained when evaluating the above infrastructures with a realistic application running on top of them. Pros and cons of both infrastructures are described here.
• **Chapter 6** presents our approach to lift some mechanisms from the middleware level to be incorporated at application level, by using the aspect-oriented programming technique.

• **Chapter 7** presents the mathematical models used to obtain the optimal checkpointing interval when considering two optimization criteria: the average availability of the system and the average response time.

• **Chapter 8** Interesting studies of parameter relations are presented in the above optimization context. These include availability-performance (response time) trade-offs. Simulation results are also presented to validate the mathematical model built to obtain maximal availability.

• **Chapter 9** concludes the thesis and indicates possible future directions.

**Reading guidelines**

To help the reader to easily find the interesting parts to read we provide a brief guide here.

If you are not familiar with basic fault tolerance terminology such as fault, replication, consensus, broadcast, needed for later understanding of the work in the thesis, read Chapter 2. If you are interested in an overview of research results in mainly failure detectors, consensus, group services, performance assessment and would like to see the merits of contributions in the larger research context proceed to read Chapter 3. However, you can skip it, if you are eager to find out about our contributions’ details.

If you are interested in technical details of implementations of CORBA middleware incorporating fault tolerance mechanisms, read the whole Chapter 4. If you are only interested in details of an implementation closely following the FT-CORBA standard, read Section 4.1, and proceed to following chapters. To understand the reasons behind the improvement presented in Chapter 6 you must read Section 4.1.4, but not more.

If you are only interested in mathematical modelling for obtaining the optimal checkpointing interval, skip chapters 4 and 6 and read Chapter 7. If you are not interested in the mathematical computations that lead to the formulas of the quantities to optimize, you need from Chapter 7 only the tables (7.1, 7.2 and 7.3) and the formulas of average availability and average response time, i.e. equations (5) and (11), and then you can proceed and read Chapter 8.
Chapter 2

Terminology

This chapter presents basic notions necessary to make the thesis self-contained. The notions that will be introduced range from synchrony, asynchrony, replication, to consensus, group communication and notions of queuing theory.

Basic notions related to systems that are assumed to fail and still proceed in their actions, were introduced in the early 1970’s. In the early years there was little unifying work to formalize these notions. This unfortunately resulted in authors not systematically building on earlier agreed common grounds for a while. We see, for example, that basic notions such as crash and failstop are used in one way by some authors [144, 143], and the opposite way by others [51, 140]. An early attempt at defining commonly agreed upon terms appeared in the work by Laprie [89], that was later on adopted and developed as the working document for the IFIP WG 10.4\(^1\) on dependable computing. In 2001 a new document about dependability concepts [5] appeared, bringing in few changes or additions to the basic notions.

One of the early attempts at modelling systems for achieving fault tolerance appeared in the works of Cristian [35]. However, it was not until late 90’s that a general framework for formal treatment of fault tolerance appeared [62].

2.1 Faults and failure models

In order for a system to be able to provide service in presence of something “going wrong”, it must be possible to make the system “go right”. For this, there is a need

\(^1\)International Federation for Information Processing Working Group 10.4
for understanding what failures can the system be exposed to, to detect them, as well as continuing to provide services despite the possibility of failures.

2.1.1 The notions of fault, error, and failure

Faults, errors and failures are the threats to a system providing its expected service to its users. Before a failure occurs, there is a chain of events. First, a fault (i.e. a defect at a low level) has to exist or appear in one part of the system. This fault can lead to an error, if the part of the system is activated under certain conditions. An error is, thus, the manifestation of a fault. We can say that if an error happens the system is in an erroneous state. A fault that does not cause an error, i.e. it is not active, is called dormant [5]. If the error is not “treated”, it can lead to a failure. The treatment can be repairing a broken part, switching to a redundant unit, or jumping to an exception handler in a piece of software. According to Avizienis et al. “a failure is an event that occurs when the delivered service deviates from correct service. ... A failure is a transition from correct service to incorrect service.”[5].

The chain fault - error - failure - ... can be quite long. A fault in a higher level of a system can, in its turn, be caused by a failure in a lower level.

Classifications of faults and failures are given in the literature [46, 5]. Two examples are given below.

Considering their occurrence in time, faults can be:

- transient - occurs at a certain moment, lasts for a time and then “goes away”, e.g. a radiation beam hitting a circuit but not affecting memory
- intermittent - occurs in the system from time to time and repeatedly “goes away”, e.g. a system overload fault
- permanent - fault that is always there in the system; the question is when it will cause an error. For example, a design fault is a permanent fault.

Considering the intent of their occurrence faults can be classified as:

- accidental - a user presses a wrong key when using a system, as he/she does not know how to use the system, and the system is not robust enough
- deliberate, non-malicious - usually related to intrusion in a system for reading data; a bank employee can penetrate the security system of a bank’s computer system and read account data (although not authorized to do so); however, this does not affect the well-functioning of the bank’s computer system
• deliberate, malicious - if, for example, a person, besides reading the data from the bank’s databases also manages to tamper the code of the software handling the accounts in a malicious way.

The next section will describe how a system is modelled by the failures it can be exposed to.

2.1.2 Failure models

A failure model shows the way a system can fail as perceived from other systems using its services. A similar term to express the same meaning is, failure semantics [36]. The failure model of a system gives information about which parts of the system can fail in certain ways and, thus, indicates which failures can be recovered from in the system.

The class of failures covered by a particular algorithm is called the algorithm’s failure model. This means that the algorithm works correctly as long as failures indicated in the model occur. For example, a distributed algorithm is designed such that even if one (or maybe several) participant nodes fail in a certain way (given in the model), the algorithm will perform correctly delivering the correct result.

Node failure models

The main constituents of a distributed system that might fail are the processes (nodes) and the network connections (links). In the following, when we talk about node or process failure we mean the unit on which computations are performed, differentiated from the communication media that is the network.

Defined by Versissimo and Rodrigues [158], the arbitrary failure model is the most generic, involving nodes. Thus, when choosing such a model, one is on the safe side, but at a high price. More restrictive failure models are used to obtain reasonable designs. In the arbitrary failure model category we have also the Byzantine failure model (Byzantine and arbitrary are in some cases interchangeable). Here the unintended behaviour of a unit can potentially be caused by malicious attacks. Versissimo and Rodrigues [158] describe Byzantine failures as occurring when a node commits a semantic failure, i.e. it diffuses a semantically incorrect result to other nodes in an inconsistent way. In other words, the node sends different (incorrect) messages to other nodes.

At one extreme, the common failure model involving nodes is the crash failure. According to Fischer[51] and Schneider [140] to fail by crashing means that the
unit stops executing its computation and communication, and does nothing further. This type of failure is not detectable by other nodes. They cannot decide whether the node that stopped communicating with them is failed or only too slow.

Other failure models delimited by Schneider [140] are the failstop failure model (similar to crash, but the failure can be detected), receive and send omission failure model (a processor fails to receive, respectively send some of its messages).

Network failure models

When considering the communication media, failures can be due to a broken communication wire, or to congestion of a communication link. In both cases, the two units connected by the failed link cannot communicate.

Two types of failure models can be considered. One is the partition failure model. In this case, we say that the two nodes are in different network partitions, unless there is some other indirect way for them to communicate. A partition also means that it is possible for the system to split in two (or more) disjoint sets of processes working separately, and each set believing that there are no other working processes in the system.

On the opposite side we have the no-partition assumption, as defined by Ricciardi et al., which means the reverse: there is no such situation when there are at least two disjoint sets of processes in the system, such that members of these sets believe that they are the only alive nodes [136]. As a consequence, if there always exists at least one correct process that is not suspected to have failed by other correct processes, then the no-partition assumption holds. A correct process is one that is not failed, i.e. it is up and running. Note that it is possible that a correct process is (wrongly) suspected to have failed.

Similar to the no-partition assumption is the primary-partition notion. This means that there will always be a majority of correct processes, and progress will be allowed only if this majority can participate in decision making.

The next section will present an important aspect that relates to the possibility of detecting faults: the way the distributed nodes perceive the passing of time.

2.1.3 Timing models

If we think about how the passing of time is perceived by the nodes in a distributed system we distinguish two basic models: synchronous system model, and asynchronous model.
2.1 Faults and failure models

In a synchronous system model one makes assumptions about processing speeds, i.e. for each system component $i$ it is known how many computation steps it performs while another component $j$ performs one step. Also, communication delays are bounded and known. Therefore, in such a model, it is easy to detect when a node (process) is failed, by noting that, e.g. its heartbeat message did not arrive within the known time interval (communication delay).

An asynchronous system model, classified by Schneider [140] as the “no-assumption model”, is a generic one since all systems can be seen as asynchronous. In such a model assumptions are made neither about relative processing speeds of nodes, nor about communication delays. Also, as presented by Verissimo and Rodrigues [158], such systems also lack knowledge about drift of local clocks and relation to other clocks’ indications. There has been a large body of work on clock synchronization algorithms over the years (for historical snapshots see [144, 157]). However, in an asynchronous system even having clocks perfectly synchronized cannot help, as there is no knowledge about how long one has to wait before labeling a (maybe too slow) node as failed. The asynchronous model is most appealing to be used when aiming for building realistic fault-tolerant systems.

Milder variants of asynchronous system models (the above mentioned one is time-free [158]) are the timed asynchronous [37] model, or quasi-synchronous [4] model, or simply partially synchronous models. A partially synchronous system model is derived in fact from real-life observations: there are intervals during the system’s life-time when there exist bounds on message delays and on process execution times, i.e. the system is not permanently asynchronous or synchronous. A notion introduced by Dwork et al. is partial stability, in that the bounds are not known [45]. Actually, Dwork et al. delimit very precisely two partial synchrony models: when the bound on message delays and/or process synchrony hold only after a global stabilization time, and when the same bounds exist and hold all the time, but are not known.

Now, considering that the (time-free) asynchronous model is most appropriate for building realistic fault-tolerant systems, the “FLP impossibility result” of Fischer, Lynch, and Paterson [52] is indeed revolutionary. They prove that in a distributed system, where the time-free asynchronous timing model is used, even with only a single node failing, it is impossible for all nodes to agree on a common outcome, in bounded time. And this, intuitively, only happens because there is no way to decide whether the non-responsive node is only too slow, or it is really failed.

To circumvent the impossibility result, in 1996 Chandra and Toueg [25] formalized the notion of unreliable failure detector. This notion involves further as-
sumptions, but not directly related to the timing model (see next section).

### 2.1.4 Unreliable failure detectors

Any system that treats faults in a foreseeable manner has to be able to recognize the symptoms of faults leading to failures at subsystem level. Therefore, fault detection (or recognition) is an important contributing factor in achieving fault tolerance.

If the time-free asynchronous system model is used, in presence of failures, we saw that there is no possibility of guaranteeing that agreement is reached in bounded time. However, it is possible to augment the distributed system with special modules. Thus, one does not depart from a real setting of a distributed system, i.e. the time-free asynchronous timing model. Instead, the decision about a node’s failure (based on delayed message arrival) is taken, possibly erroneously, and further used as such. However, how harmful can this be as long as one is conscious that this decision can be wrong, and acts to reduce the consequences of a mistake?

This aspect is explored by Chandra and Toueg in their formal characterization of failure detectors [25]. They actually play around with “mistake patterns” that enable them to specify the so-called **unreliable failure detectors**.

For example, when executing some distributed algorithm, processes do not explicitly reason about time – time is simply abstracted away from the algorithm execution level. Instead, they query their failure detectors about the availability of the other nodes.

The failure detectors can make mistakes, by suspecting processes that are correct, to be faulty – the so-called lack of **accuracy**, or by not detecting an actual failure – the so-called lack of **completeness**.

Even though **strong accuracy**, i.e. the property that a correct process is never suspected by any correct process, is difficult to achieve, some weak accuracy or eventual accuracy properties can be satisfied. The same is true about completeness (eventually all process failures are detected). **Strong completeness** (eventually all correct processes will permanently suspect all failed processes) is difficult to achieve in an asynchronous setting. However, Chandra and Toueg prove that for correctness of some distributed algorithms it is enough that the failure detectors have some weak properties. The impossibility result by Fischer et al. can be circumvented if the failure detector properties hold for sufficiently long time for performing the algorithm. This revolutionary outcome of Chandra and Toueg’s work actually made possible to solve the agreement problem in distributed time-
free asynchronous systems.

In particular, the weakest failure detector (called “eventually weak” \( \Diamond W \)) which makes it possible to solve consensus in time-free asynchronous systems, when a majority of processes are always up, has the properties eventual weak accuracy\(^2\) and weak completeness\(^3\) [24].

As intuitively one can guess, it is not enough to detect failures in a distributed system to make it fault-tolerant. The next section will present the other ingredient that leads to a fault-tolerant system.

### 2.2 Achieving fault tolerance

As formally proved by Gärtner [62], there is “no fault tolerance without redundancy”. Although the common usage of redundancy deviates from what Gärtner used, his result is still convincing as a theoretical support to our intuitions. There are two types of redundancy: in space and in time.

Redundancy in space means that multiple copies of a process are added to achieve fault tolerance. The same is true for data; multiple copies of a data item are used to provide masking in case of failures. Another possibility is to augment data items with redundant information (for example parity bits and CRC code in data transmission). When all copies of redundant information are enforced to be consistent, i.e. have the same content, we talk about replication. In such a setting, majority vote can be used to enforce changing a data item consistently, or creating a unique reply to a query.

In Gärtner’s view, redundancy in time means that some action in the code of the process is executed only in case of failures. For example, a process can be divided in mandatory and optional parts where, upon failure in the mandatory part this is reexecuted instead of optional parts [8].

#### 2.2.1 Software fault tolerance

A well known approach for writing replicated fault tolerant software is design diversity. This is part of the so-called software fault tolerance. There are two techniques employing design diversity: \textit{N-version programming} [6] and \textit{recovery blocks} [130].

\(^2\)there is a time after which some correct process is never suspected by any correct process

\(^3\)eventually, every crashed process is permanently suspected by some correct process
With N-version programming, several versions of the same program run in parallel and the result of the computation is chosen by voting on the results of each version.

When employing recovery blocks, the outcome of a specific algorithm to be executed is checked against an application related acceptance test. A new recovery block is chosen and a new version of the algorithm reexecuted, until such a test is passed.

In the first item, implicitly it is mentioned how the redundant items are deployed: “... versions of the same program run in parallel ...”. In the next section the possible ways of deploying and using redundant units in a fault-tolerant server setting will be explained.

2.2.2 Replication strategies

If the fault-tolerant system is a sort of server that receives queries from clients, it is important to distinguish different replication styles. These tell which replica(s) resolve the client queries, as well as how the replicated server is maintained in order to offer the client the image of a single unit.

Four replication styles are mentioned in the literature.

- Primary-backup: in this approach, there is only one replica that processes requests from clients and answers to them (the primary). The backups in this case, are just receiving state update information (in some forms of primary-backup) from the primary. The reason for this is that in case of a primary crash, one backup to be able to take over the job of the failed replica in a predetermined manner [69]. See Figure 2.1(a) for a graphical illustration of the concept.

- Active replication: in this approach, all replicas process requests from the client in an active way. The restriction here is that all replicas perform deterministic computations in a state machine fashion [139]. If the crash failure model is considered it is enough to send one of the answers, from the fastest replica. If the Byzantine failure model is considered, failures of members are masked by using a voting mechanism on the results. In the former case it suffices with one replica more than the number of failures tolerated [139]. In the latter case when a synchronous system is considered, the number of replicas has to be $3f + 1$ if the failure of $f$ replicas has to be masked [86].
2.2 Achieving fault tolerance

- Semi-active strategy: in this approach, there is a central replica that does all the computation steps and that leads all the other replicas. In case of a non-deterministic choice, the replicas wait for the leader’s decision, and they follow it [41].

- Semi-passive strategy: in this (semi-primary-backup) replication style, the primary is elected by a consensus algorithm [125].

2.2.3 State and checkpointing

The state (as used in “state update” or “state machine”, see Section 2.2.2) refers to the collection of data on which the processes act. The state might also include actions performed up to a certain point during a computation.

Ideally, in all types of replication strategies, failures have to be handled such that the states of the replicas (of the group) remain consistent. In a group using primary-backup replication, if the primary fails, this has to be noted by the backups that have to choose a new primary, and continue providing services from where the primary left them. This can be achieved by remembering the operations performed by the primary, while up, and as soon as it fails replay those operations on one of the backups. To avoid situations in which a large number of operations are reexecuted on the backup when this takes over the role of the primary, from time to time the state of the primary is read and recorded in preferably stable storage. The operation of reading and storing the state is called checkpointing. By employing checkpointing, operations that have to be stored for replay are only those executed by the primary since the last state recording. This improves storage requirements as well as backup take-over times. Here we introduce also the term checkpointing.
that we define as the time interval from the end of one checkpointing, i.e. the moment of state storage, until the beginning of the next checkpointing. The procedure that follows the occurrence of a failure when a backup takes over the role of the primary, is called failover. It consists of transferring the latest checkpointed state of the primary in the backup, followed by replaying of the update requests that arrived since the last checkpoint was read on the backup.

The next section will describe primitives for communication between replicas.

2.3 Communication in fault-tolerant distributed systems

Because in a replicated fault-tolerant service framework, all processes have to be involved in the communication, the sending of messages is done via broadcasts [70]. When the message is sent to only a subset of a global destination set, then we talk about multicast.

2.3.1 Broadcasts

In a failure prone setting, even a broadcast activity can be affected by problems caused by failures. We have to introduce the notion of reliable broadcasts. Before that we need to define the notion of message delivery and distinguish it from message reception:

- **Receiving a message** means that the message arrived at the destination process. This is similar to a letter arriving in one’s post box.

- **Delivering a message** – usually “to an application” or “by a process” – means that the message is ready to be used by the application. Thus, message delivery in this context can be seen as analogous to taking out a letter from one’s mailbox and reading it.

A reliable broadcast is a broadcast with the following three properties:

- **agreement**: all correct processes deliver the same set of messages

- **integrity**: a message is not delivered unless it is sent, and it is delivered only once

- **validity**: a message sent by a correct process is eventually delivered by all correct processes
2.4 Consensus as a basic primitive

From the agreement property it follows that every message is either delivered by all correct processes or by none of them. The other two properties imply that the underlying message delivery system neither creates, nor looses messages.

2.3.2 Message ordering

Besides being reliable, sometimes a broadcast primitive has to guarantee different types of message delivery order. Depending on the delivery order restrictions, we can distinguish different types of reliable broadcast primitives.

One possibility is to ask for a FIFO delivery order (thus, related to a sender) - messages from sender A are delivered in first-in-first-out order. There is no restriction about messages from sender A as compared to messages from sender B [70, 158].

A second possibility is to ask for causal delivery order. In this case it is required that before delivering a message, all messages causally preceding it should be delivered first. For node B delivering m sent by node A, one way would be to first deliver all messages that were delivered by A before sending m, and then deliver m [15].

A third possibility is to request the same order for delivery of messages. This means that messages should be delivered at all the receiving nodes in a total order, also called atomic order. Of course, this atomic order property refers to ordering of messages that otherwise are not restricted by other criteria (like causal or FIFO delivery ordering) [70, 158].

Communication between replicas is important especially when they need to perform distributed algorithms for agreement. Agreement algorithms are put under one name: consensus. This is the topic of the next section.

2.4 Consensus as a basic primitive

A consensus algorithm is usually defined in terms of the two notions propose and decide.

Replicas typically need to agree on a joint value within a reasonable time. Besides termination of the consensus operation, the value decided upon has to be a valid one, in the sense that it has been proposed by one of the replicas.

The consensus operation is invoked by the replicas at a certain moment when there is a need to get a unique result. Each process proposes its value. After a set of steps, the processes arrive at a point when they decide on the final value. The
algorithm has to be constructed such that it guarantees that processes decide on the same value.

Summing up the above, an algorithm that solves consensus has to satisfy the following three properties:

- Agreement - no two (correct) processes decide differently
- Validity - each (correct) process decides at most once and on a value that was previously proposed by one of the processes, or is related to those proposed values
- Termination - every (correct) process eventually decides

The reason why consensus is considered as a basic primitive is that a number of problems in distributed systems are reducible to consensus, or solvable by using consensus. Examples of such problems are the atomic broadcast problem (ABP) [25], and the atomic commit problem [68].

Take, for example, the atomic broadcast problem: processes have to agree on the order they deliver a set of messages. When all processes have the set of messages in possibly different orders, they will start a consensus algorithm to choose the unique order of delivery. This is an intuitive way of showing that the ABP is reducible to consensus - if there is an algorithm that solves consensus then the ABP can also be solved. On the other hand, if there is a solution for the ABP, meaning that all processes in a message delivery step will deliver the same message, then we can say that processes agree on the value of that message and thus reach consensus.

The next section will present some terminology that is valuable for quantified measures of availability and the relationship between this system level property and other system parameters.

### 2.5 Queuing theory

In a well written book [3] about queuing theory you will find at least the notions of Markov property, exponential distribution, server, customer, customer interarrival time distribution, $M/M/1$ queue, service time distribution and wait time in the queue. We will explain these in turn later.

First, let us introduce the notions of random variable, probability distribution, conditional probability and expectation (average value). Then, we will explain the very notions of queuing theory, in the last part of this section.
2.5 Queuing theory

2.5.1 Random variables

A simple definition of a random variable [65] is: a random variable is a function that associates a real number to the outcome of an experiment. This outcome can be a continuous number (like e.g. the outside temperature, in case the experiment is the reading of a thermometer), or a discrete number (like e.g. a number associated with the face of a coin). In the first case we talk about continuous random variables, and in the second case we talk about discrete random variables.

A random variable receives a certain value with a certain probability. For example, for the thermometer reading, the probability of the temperature to be less than 20 degrees is 0.58. Now, each number \( x \) has associated a probability of the random variable being less than \( x \) (\( P[V \leq x] \)), at any point in time. The function \( f(x) = P[V \leq x] \) is the probability distribution of variable \( V \). Note that for \( \forall x \in \mathbb{R}, 0 \leq P[V \leq x] \leq 1 \). Let \( v \) be the probability density function (abbreviated pdf), with the following properties:

\[
\begin{align*}
& P[V \leq x] = \int_{0}^{x} v(t) \, dt \\
& P[x < V \leq y] = \int_{x}^{y} v(t) \, dt \\
& \int_{-\infty}^{\infty} v(t) \, dt = 1
\end{align*}
\]

The probability density function for the exponential probability distribution with parameter \( \lambda \) is

\[
\begin{cases}
0 & \text{if } t < 0, \\
\lambda e^{-\lambda t} & \text{if } t > 0.
\end{cases}
\]

For example, if we have a server at which client requests arrive with an exponential interarrival time distribution, then \( \lambda \) is the average rate of arrival of those client requests.

For a discrete random variable that can take discrete values, say in the set \( \{v_1, v_2, \ldots, v_n\} \), the probability distribution associates a probability with the event that \( V = v_i \) (\( P[V = v_i] \)). Here, we have the property \( \sum_{i=0}^{n} P[V = v_i] = 1 \), with \( 0 \leq P[V = v_i] \leq 1 \), for all \( i \).

Conditional probability is defined in the context of several events occurring dependent on each other. If we have two events \( e_1 \) and \( e_2 \), if they are independent then the probability of both events occurring is \( P[e_1 \land e_2] = P[e_1]P[e_2] \). In general, whether they are dependent or not:
In case $e_1$ and $e_2$ are independent, $P[e_2|e_1] = P[e_2]$ and $P[e_1|e_2] = P[e_1]$.

The expectation (or average) of a random variable is, intuitively, the value of that variable that gives the highest value of the probability density function of that variable. The average of a continuous random variable is computed as follows:

$$E[V] = \int_{-\infty}^{\infty} v f(v) \, dv$$

For example, the average value of a random variable with exponential distribution with parameter $\lambda$ is

$$\int_{-\infty}^{\infty} 0 \, dv + \int_{0}^{\infty} \lambda r e^{-\lambda r} \, dr = \int_{0}^{\infty} \lambda (-e^{-\lambda v})' \, dv = \int_{0}^{\infty} \lambda e^{-\lambda v} \, dv = \frac{1}{\lambda} \int_{0}^{\infty} (-e^{-\lambda v})' \, dv = \frac{1}{\lambda} \lambda (1 - 0) = \frac{1}{\lambda}$$

The average value of a discrete random variable is computed as follows:

$$E[V] = \sum_{i=1}^{\infty} v_i P[V = v_i]$$

In Chapter 7 we will often use the following property of average values: the average of a sum of random variables (not necessarily independent) is equal to the sum of the averages (e.g. $E[A + B] = E[A] + E[B]$) [3].

Conditional expectation can be defined in several flavours (conditional expectation given an event, conditional expectation given another random variable, etc.). In this work we are interested in conditional expectation given an event. Given a random variable $V$ and an event $e$ we have that:

$$E[V|e] = \frac{E[V1_e]}{E[1_e]}$$

where

$$1_e = \begin{cases} 
1 & \text{if the event } e \text{ happened}, \\
0 & \text{if the event } e \text{ did not happen.}
\end{cases}$$

Therefore, $E[1_e] = P[e]$.

More concretely, if $e$ is the event $A < B$, where $A$ and $B$ are random variables, then

$$1_{\{A < B\}} = \begin{cases} 
1 & \text{if } A < B, \\
0 & \text{if } A > B.
\end{cases}$$

Therefore $E[1_{\{A < B\}}] = P[A < B]$.

### 2.5.2 Queues and related notions

Here we present a very brief overview of basics of queuing theory.
Customers are arriving at a server to receive service. If the server is busy servicing other customers when a new one arrives, then this one has to wait in a queue that builds up in front of the server. The time between two consecutive customer arrivals (customer interarrival time) is a random variable with a certain probability distribution. The most popular distribution used is the exponential distribution. Also, most of the queuing models use the assumption that these interarrival times are independent of each other.

The service time of the server is also a random variable, and the most popular distribution used is the exponential distribution. The nice about the exponential distribution is that it has the Markov property. This is the reason why it is used in queuing theory.

Let $X$ be a random variable. We give the definition of the Markov property by citing from the book by Allen [3]: "... Markov property, sometimes called the “memoryless” property, given by $P[X > t + h] = P[X > h]$, $t > 0$, $h > 0$. ... if $X$ is the waiting time until a particular event occurs and $t$ units of time produced no event, then the distribution of further waiting time is the same as if it would be if no waiting time had passes – that is, the system does not “remember” that $t$ time units have produced no “arrival”.”

The average time a customer waits in the queue of a server with exponential service time distribution, when customer interarrival times are independent and exponentially distributed, is given by a classical formula involving only the average rate of customer arrivals and average service rate.

A queue of a server with exponential service time, and exponential customer interarrival time is called $M/M/1$ (Markov/Markov/1 server).

According to Burke’s theorem [21], the interdeparture times of customers from a server in a $M/M/1$ queuing system, are also independent and exponentially distributed random variable, and have the same average as customer interarrival times.
Chapter 3

Fault tolerance and middleware

This chapter presents an overview of theoretical and practical research findings in the area of availability, fault tolerance and middleware. The first four sections of the chapter cover more details of the notions introduced in Chapter 2. The reason for providing a deeper coverage of research results in this chapter is to give the reader a larger perspective within which our work should be understood. The last three sections briefly survey fault tolerance support in commercial middleware, and assessment of performance and availability. These three subjects also set the broad scene in which our contribution takes place.

The first section deepens the knowledge about consensus. Chapter 2 defined the consensus problem. Here, we will review algorithms for solving it. The following two sections are dedicated to broadening the view on failure detectors, including perfect detectors. The fourth section provides a perspective on how transparency of replication is achieved via groups and sets the stage, in the fifth section, for a review of commercial middlewares and their fault tolerance features.

3.1 Basic algorithms for consensus

Studying consensus in asynchronous systems is interesting because of the early “FLP impossibility result” [52]. However, in that context, the failure model was the simplest one: crash. A survey of consensus algorithms in synchronous systems by Raynal [132] shows interesting problems arising in these more restrictive timing models. Here, failure models can be crash, as well as omission and Byzantine. Also, the notion of uniformness is possible to exploit here. Uniform agreement, for
Chapter 3. Fault tolerance and middleware

example, means that no two (correct or not) processes decide different values.

### 3.1.1 Consensus in synchronous systems

Interesting findings for synchronous systems were [132]:

- When processes fail by crashing, i.e. after failing, do nothing, or by omitting to receive or send messages, a relatively simple algorithm helps processes to reach agreement after \( f + 1 \) (\( f \) is the number of failing processes) rounds of message exchanges.

- When processes fail by omission failures, since omissions allow the “failed” processes to eventually reach a decision step, the agreement property of consensus might be violated. This is due to the fact that if, e.g. the processes that constantly omitted receipt of information from the rest of deciding parties, will finally decide on their own proposed value. In other words, uniform consensus cannot be solved by the simple \( f + 1 \) round algorithm, in presence of omission failures. A different algorithm in \( f + 1 \) rounds can solve consensus for this failure pattern, but only if a majority of processes stay correct.

- When processes are exposed to Byzantine failure patterns, an algorithm with \( 2f + 2 \) message exchange rounds solves consensus, but only if at most one fourth of the processes fail (\( f < \frac{n}{4} \)).

Note the hierarchy of failure models: crash is less severe than omission that is less severe than Byzantine; the algorithm that helps processes reach agreement in presence of Byzantine failures necessitates more messages exchanged by processes, and more correct processes present in the agreement procedure.

### 3.1.2 Consensus in asynchronous systems

In asynchronous systems, combined with the simple crash failure model, from the perspective of node behaviour after a crash, different assumptions are made: no-recovery and recovery assumption.

Algorithms for solving consensus in the crash-no-recovery model were presented by Chandra and Toueg [25] in their seminal work about unreliable failure detectors. No-recovery means that a process, after it (really) crashes, it is not able ever to return to participate in the distributed algorithm. Thus, as soon as the failure
detectors suspect a crashed process, it will stay in their suspect list forever. On the other hand, if a process is only too slow and suspected to have crashed, the failure detector might “change its mind”. Thus the crash-recovery model is also present, in a way.

One of the algorithms presented solves consensus by using a strong (strongly complete and weakly accurate) failure detector, even if all but one process fail. Since the weak accuracy property is satisfied, there is a correct process that is never suspected to have failed. The main idea is to execute the algorithm in such a way that the decision of this one process eventually reaches all the other correct processes. The consensus algorithm is distributed in the sense that each process executes the same steps. The steps contain also synchronization points when in a certain iteration – also called a round – a process waits for messages sent in the same round by the other non-suspected processes.

When using eventually strong (strongly complete and eventually weakly accurate) failure detectors, consensus can be solved only in the presence of at least a majority of correct processes. The algorithm is based on a rotating coordinator approach (this coordinator is the decider). By eventual weak accuracy it follows that eventually there is at least one correct process that is not suspected by any correct process. Thus, eventually, the current coordinator will manage to decide the final value. When suspecting the coordinator to have crashed, processes move to a new round with a new coordinator. However, this situation will not last forever.

When solving consensus in the crash-recovery model, a new type of failure detector is needed. In this model, the process loses all data from its memory when it crashes, but it is repaired and it can return and participate in the algorithm. Aguilera et al. [2] define the new failure detector to have somewhat different completeness and accuracy properties than the Chandra-Toueg initial detectors. The new type of failure detector that is needed is justified as follows. There can be processes that crash and recover infinitely often. Thus, it would be wrong to allow a failure detector to permanently suspect them, as allowed by the strong completeness property in the sense of Chandra and Toueg [25].

This section described an important primitive used in fault-tolerant distributed systems. It is typically used in the context of presenting a unique result of an operation that was performed by a set of participating entities. The next section will present a follow up of Section 2.1.4 strongly connected with consensus, i.e. research results concerning different types of unreliable failure detectors.
Since the formal introduction of unreliable failure detectors by Chandra and Toueg [25] in the context of circumventing the “FLP impossibility result” in asynchronous distributed systems, a lot of research has taken place in this area. A good description of failure detectors is given in the book by Tel [149].

Most research results are relating different classes of unreliable failure detectors to the solvability of consensus. The main class of failure detectors studied, stamped as the weakest failure detector to solve consensus, is the “eventually strong” $\Box S$. This failure detector is stronger than $\Diamond W^1$ in that it has the property of strong instead of weak completeness. Ensuring the strong completeness property demands little effort.

Considering the $\Box S$ failure detector, the way implementations of the failure detector influence the termination time of consensus, is studied by Sergent et al. [142]. The findings are related to trade-offs between how quickly a failure is detected, implying frequent checking the availability of a processing unit, and the number of messages sent on the network. Also, the way one checks whether a processing unit is up, makes a difference in when the consensus algorithm terminates in case of no-failures, as well as when failures occur. The main conclusion of the work is that the design of a consensus algorithm is orthogonal to implementing the failure detector.

Larrea et al. [91] study failure detector implementations. One important result is an optimal implementation of the eventually strong failure detector. The approach is to reduce the number of messages exchanged when detecting failures. Another work of Larrea et al. ([90]) shows that it is impossible to implement failure detectors with perpetual (as opposed to eventual) accuracy in partially synchronous systems. Their result shows also that since consensus can be solved with “eventual” failure detectors (even with the weakest one) perpetual failure detectors are not necessary for solving it.

In theory, the fact that consensus in an asynchronous system, where process failures are detected possibly erroneously, eventually terminates, is a good enough quality. However, in practice, as stated by Chen et al. [26] some applications have certain timing requirements and thus eventual termination guarantees are not sufficient. The notion of quality of service of failure detectors is introduced as a measure of how fast a failure detector can detect actual failures and how much it can avoid false suspicions. Depending on the quality of service required by a certain applica-

---

1 Defined in Section 2.1.4
tion the failure detector can be adapted to the new situation by changing different timeout parameters. Further extensions of the adaptability of failure detectors are provided in the work of Sampaio et al. [137]. Analysis of the timing behaviour of the rotating coordinator based consensus algorithm is done when processing as well as bandwidth resources available for a process are not known a priori. The authors employ Petri nets to model the consensus protocol with different resource availability scenarios and different failure detection scenarios.

The next section will explain how perfection can be achieved in failure detectors.

### 3.3 Perfect failure detectors

Sometimes, when solving consensus problems, besides the minimal safety requirements like agreement and validity (i.e. decision on a unique valid value), an important demand is for decision making to be based on using proposed values from all correct processes. For example, this is needed to solve the strict version of the atomic commit problem, when it is not permitted to ignore any answer (YES or NO) from a correct process. This requirement can not be satisfied if the failure detector makes “wrong suspicion” mistakes, i.e. it suspects a correct process to have crashed, thus ignoring its value in the decision phase.

A strict(er) form of consensus is the problem of decision based on a global set of values in presence of perfect failure detectors. Helary et al. [78] call this notion global function computation. The questionable part of this argument is the method for implementing perfect failure detectors. This relies on a somewhat “perfect real-world” assumption: the possibility of having privileged channels – channels that are never congested and never go down, and therefore have predictable timing behaviour. Thus, processes could use these channels for sending “I am alive” messages. The failure model includes crash failures and the assumptions that a process does not recover from a real crash, and there is no such thing as revoking ones suspicion about a process crash. The reason for the latter is that failure detectors used are perfect and they do not suspect a crash if this is not real. Also, as there is no mistaken crash suspicion, all correct processes will see the same processes as being up or down. As a consequence, there is no possibility for disjoint sets of separately working processes to exist.

Another view on perfect failure detectors is given by Delporte et al. [42]. Their thesis is that the weakest failure detector to solve consensus when the number of failed processes is not restricted is no longer $\Diamond \mathcal{W}$ (or $\Diamond \mathcal{S}$), but $\mathcal{P}$ (perfect)!
this, by only considering failure detectors that are realistic, i.e. cannot guess the future. In this context, any realistic failure detector used when any number of processes can crash, can be transformed to a perfect failure detector. From here they concluded that the perfect failure detector is the weakest to solve the problems. Thus, the failure detector hierarchy of Chandra and Toueg tends towards one single failure detector to solve consensus, when the number of failures is not restricted.

The Timely Computing Base (TCB) model introduced by Casimiro and Verssimo ([156, 23]) employs a perfect timing failure detector. A timing failure occurs if a timed action terminates some time after its specified termination time. The TCB is used by applications built over an asynchronous network, that need control of the timely execution of processes. When the TCB is involved, it offers the possibility of measuring time durations, detecting timing failures and being able to execute in a timely manner functions of the application. The “perfection” is achieved by making the TCB module synchronous. This way, the properties of timed strong completeness and timed strong accuracy can be enforced. Timed strong completeness means that any timing failure is detected in a bounded time interval after the specified termination moment of the respective action. Timed strong accuracy means that a timing failure is not deemed to have occurred if the timed action finishes in a bounded time interval before the specified termination moment of the respective action.

The next section will present the notion “process group” related to how a set of replicas act towards an external client or viewer, as one unique entity.

### 3.4 The process group abstraction

After introducing the notions of “group” and “group services” this section will describe some platforms incorporating fault tolerance handling mechanisms, that were the precursors of today’s commercial middleware platforms.

Intuitively, a process group is a set of replicas being able to do the same processing task, or contribute towards fulfilling a common goal. The meaning of process group in the fault-tolerant computing community is the following: a group of processes that cooperate with each other, is a mechanism for building fault-tolerant service providers. The members of the group are not fixed a priori, and the group mechanisms take care of dynamic changes to the group. A service provider, in this context, is a collection of replicated servers, service group, that process requests from clients. The group is transparent from the client’s side. Thus, the client, when sending a request to the replicated server (or now, a process group), perceives it
3.4 The process group abstraction

as one addressable unit. A group might be used to manage complexity, to balance load, or to provide fault-tolerant services [126]. The group has to manage itself to assure the one-unit image.

An important notion is the group member. A process may or may not be a member of a group at different times. If it is not, it might have been excluded from the group due to a crash (real or suspected), or due to isolation from the group because of communication failures. In the latter case, it is possible that the process is still working, but its view of the group membership differs from that of other processes’ views. For example, in the AQuA framework [133], when passive replication is used, two types of members can exist in a group: persistent and transient. Persistent members do not leave the group once they joined it. Transient members only join a group when they need to send a message to its members.

3.4.1 Group services

An indispensable service in a group setting is the one that informs about the group membership. The group members themselves use this to be able to assure transparency even in presence of failures that involve masking. A membership service has to ensure that group members have a consistent view of the group.

Another specific group service is the state merging (or transfer) service. This is used when the initial group is split in several, or one majority and some minority, partitions. When, e.g. links are repaired and partitions disappear then all new group members have to know the state of the rest of members. The state transfer service is also used when a new node joins the group. The operation of state transfer has to occur as an atomic action.

3.4.2 Specification and implementation of group services

Formal description of membership service properties both for synchronous systems and asynchronous systems are found in the works of Cristian ([35, 38]). For explaining problems in presence of the asynchrony assumption, Cristian introduces the notions of stability and instability of a system. When a system is stable, then communication between processes either works all the time with constant (bounded) delays, or it does not work. In contrast, during instability periods there are no guarantees of message delay bounds; the communication can be working, but very slow, or not at all. The important thing is that the membership service has to ensure the agreement of group members on the group view and to correctly include and exclude members from the view.
Following the line of Cristian’s work, is the formal specification and implementation of a membership service for partitionable systems, by Babaoglu et al. [10]. A formal description of a communication infrastructure for processes in a partitionable distributed system is given. Thus, here, including link failures gives rise to a failure model that allows partition failures to occur.

Work by Birman et al. on the ISIS system, has minted the notion of virtual synchrony and view synchronous multicast. Virtual synchrony is related to the provision of well-chosen “synchronization” points for processes in an asynchronous system. Thus, a process will stop and synchronize with other processes, when it has to wait for messages causally preceding other messages ready to be delivered. View synchronous multicast is tightly coupled with virtual synchrony, and directly relates to groups. A view synchronous multicast works as follows: a message sent by a process while the process membership view is $V$, has to be delivered before any other new view $W$ is registered, by all surviving members of $V$. In this context, one says that a group uses a view synchronous multicast service. Although one of the leading groups in the area of group communication, much of the work done on ISIS in the early years can be characterized by a “system building” approach. A lot of the detailed algorithms were initially not formally presented [18]. Further, the procedure execution was more transaction based, and the behaviour of the system was analyzed pragmatically. Later, however, with the introduction of the notions of virtual synchrony and view synchronous multicast, algorithms and communication primitives are described more formally [15, 17].

### 3.4.3 Fault-tolerant group based platforms

Precursors of today’s fault tolerance handling middleware platforms were platforms providing underlying group communication primitives.

One of the successful experimental platforms around the ideas of software replication and group services were implemented in the Delta-4 project [125]. This work leaves the decision concerning the particular way of replication as an open question for the application and provides several modes to choose from in the open architecture. The project was discontinued in 1992, but several mechanisms were studied during the project.

Totem, developed by Moser et al. [107] is a platform providing group communication primitives. It is used in the message delivery layer in the CORBA based Eternal platform. Totem is structured in several layers, deals with sending messages

\[\text{"registered" is referred to as } \text{installed} \text{ by Birman et al. ([15])}\]
among different process groups, enforces total order on delivery of messages, and also deals with merging of groups. The notion of \textit{extended virtual synchrony} is defined with Totem. In this model of virtual synchrony, care is taken even of the processes not in the current group of a certain process. Messages that are received and delivered by certain processes are delivered in the same order by all of these, regardless of whether they (currently) have the same view on their group or not. The application writer can request a certain type of message delivery order. The platform is suitable for use in soft real-time systems, since high throughput can be achieved, as well as predictable latency.

The Horus toolkit was developed by Van Renesse et al. \cite{134,155}. It allows flexible composition of protocols to support group-based services. The application developer can use protocol components with well-defined interfaces to build the needed protocol stacks, while excluding unnecessary overheads. With Horus, the difficulty lied in the transformations in the protocol stacks written in C.

The next generation of Horus was the Ensemble toolkit \cite{16,154} which is in essence similar to Horus, but written in a higher level programming language allowing more flexibility. Ensemble is one of the few group communication toolkits that was benchmarked \cite{13}. The behaviour of different primitives – membership service, FIFO-ordered reliable multicast, atomic reliable broadcast – was investigated when injecting real errors.

Challenges when integrating group communication with transaction based processing are presented by Little and Shrivastava \cite{96}. Implementation of the approach is considered on top of CORBA standard since it readily includes a transaction service specification. It was much later that fault tolerance by process replication started to become standard. There is a trade-off between the use of systems that only support transactions, and the use of systems with only process replicas (function replication). Systems with transactional fault tolerance recover better from total crashes, because the transaction concept includes undo capabilities. On the other hand, partial crashes that can happen more often than total ones are handled much faster (and more transparently) in the case of replicated groups.

The following section will present a short overview of existing commercial middlewares and how they handle fault tolerance.

### 3.5 Fault tolerance and commercial middleware

When an application developer uses a middleware to run an application on top of it, he/she expects that many common non-application related features are incorpo-
rated in the middleware. Features related to fault tolerance handling are: client
transparency to failures, and automatic replica management with state saving and
restoring features. Today’s extensively used middlewares exist in several flavors:
those incorporating component/object as well as communication management (e.g.,
CORBA, COM/DCOM), those dealing with communication management (e.g.,
Java RMI), those that support resource allocation in a dynamic way (e.g., Jini), and
those that offer containers for objects that can communicate using several offered
technologies (e.g., EJB). In the following we will briefly survey these platforms and
how they support application writers in easily writing and deploying fault-tolerant
applications.

Microsoft’s COM/DCOM allows customizing different elements of the remot-
ing architecture (infrastructure connecting clients to server objects) in order to ac-
commodate communications with multiple server copies and transparent failover.
The COMERA extension by Wang and Lee [159] was needed in order to provide ef-
ficient ways especially for replica migration where connections have to be restored
in a transparent way and logged messages to be replayed.

Java RMI, as a basic technology, does not include fault-tolerance features. On
the other hand, built on top of this communication infrastructure, middleware such
as the Jgroup toolkit [106] or the AROMA [115] system provide transparent fault
tolerance to applications. In Jgroup, clients transparently access replica groups as
one entity. Replica group members, when processing a query, exchange messages
ensuring proper state logging. The group transparently handles failure of a replica.
AROMA intercepts RMI messages, and sends them in the same order to all server
replicas, thereby assuring consistent states at failover. Replication style can be cho-
sen by the application writer on the basis of the fault tolerance needs and failover
time allowed.

Jini addresses a different approach to fault tolerance. As a technology inher-
ently used for distributed, potentially mobile, networked applications, it does not
advocate explicit replication for fault tolerance. Therefore, no automatic appli-
cation server or client replication is supported; neither is application state saving
and restoring. On the other hand, by offering the Lookup Service (possibly repli-
cated) and the lease-based resource allocation approach, failures of different service
providers can be made transparent to the client [92, 114].

EJB technology uses clustering and transactions to provide transparent fault
tolerance to client applications [76]. Still, this is not sufficient for the servers to
be available in case of failures. Enhanced with group communication [118], the
architecture can provide transparent failover to clients as well as proper state log-
Assessing performance

3.6 Assessing performance

Algorithms for solving problems in distributed systems are expected to satisfy some specifications. These contain some properties – usually considering eventual termination as opposed to efficiency – that have to be fulfilled each time the algorithm is executed. Examples are agreement and validity in connection with consensus. Using the specification, correctness of the algorithm can be proved. Thus, an interesting question is: what can one look at when reasoning about the performance of a distributed algorithm?

Some criteria could be related to time complexity – meaning the possibility of bounding the execution time. Further, two cases can be considered: the best case (when no failures occur), or the worst case when failures occur. For these criteria, analysis can include, e.g. the consideration of round numbers for an algorithm execution. This is interesting to be done when the round number is not fixed (like in the work of Chandra and Toueg [25]). Other criteria could be related to resource utilization (like memory or stable storage). The communication media can also be regarded as a resource, thus the number of messages exchanged during communications is also a measure of efficiency.

Another way to look at measuring efficiency is to see how an algorithm behaves if the worst-case assumption under which it was built does not hold (e.g. the assumption is that n failures can occur, but, in fact, only m<n failures occur in 90% of the cases).

Few works discuss real-time aspects (i.e. time bounds) in the context of distributed algorithms. Synchronous systems (i.e. those in which a global clock exists to which nodes have access), together with associated protocols and fault tolerance...
mechanisms have been extensively studied by Kopetz et al. and described under the general name of time-triggered systems [85]. The time-triggered protocol (TTP) is used in hard real-time distributed systems where timing requirements are very strict, and strong guarantees have to be provided about satisfying deadlines. Verissimo et al. [156, 23] employs a partly synchronous approach by using the timely computing base (introduced in Section 3.3). This offers also the possibility to reason about execution timeliness. Cristian and Schmuck discuss real-time aspects in the context of distributed algorithms, in a client-server configuration [35, 38]. In a synchronous setting it is not so difficult to reason about bounds on, say, agreement time, and thus on response time to the client request. In an asynchronous system, the problem is somewhat more challenging. As already mentioned before, the system model is not totally time-free. There can be some bounds on message delays, but these hold only during stability periods of the system. The interesting part is that the analysis is done by taking in consideration the instability periods of the system, as well. The algorithms described for asynchronous settings do have quite long worst-case stabilization times. More recently Lima and Burns [94] have introduced consensus mechanisms on top of a CAN bus model which can rely on the earlier real-time guarantees provided by that model.

Helary et al. give upper (worst case) bounds for the number of rounds executed in algorithms for global function computation [78]. Aguilera et al. give time estimations and message number complexity for best case executions and compare them with the same condition performances for other algorithms [2].

Other works analyze different aspects of performance of software systems in presence of failures. The goals of these works range from prediction of reliability [66], and response-time analysis in presence of failures [152, 169], to fault-detection time analysis[131], and models for checkpointing systems [22, 47, 82, 108, 164, 88, 30, 29, 121, 95, 63, 148, 146].

### 3.7 Assessing availability

A failure mode analysis on a CORBA service implementation is conducted by Marsden et al. [101, 102]. Several implementations of the CORBA Naming Service are chosen for tests. The results of the analysis are supposed to be used mainly to provide information about what CORBA implementation to choose for a certain application. This is similar to some extent with doing analysis on usage of different replication styles to provide fault tolerance, and providing information for application writers about which is the best choice for a certain application. The technique
used is fault injection of faults like bit-flips and double-zeros in the IIOP (Internet Inter-ORB Protocol) messages sent to the server. Depending on the implementation of the service, the exceptions generated give satisfactory information or not (for example, if the exception is of type UNKNOWN).

Measurements on a system under failure conditions are done by Iyer et al. [77]. More precisely, data are collected about failure causes, and downtime. The analysis by Iyer et al. concerns local area network computers, i.e. running in a more or less controllable environment, and Internet computers, i.e. computers that are connected to Internet and thus exposing unpredictable behaviour. Downtimes are measured by looking at the timestamp of the last event in the log before the reboot, and the timestamp of the event immediately after the reboot. Iyer et al. show concrete results concerning the behaviour of the system, by giving a model of the system in the form of a state machine. The arc between say, states A and B is weighted with the fraction of the total number of transactions from A into B. For example: state A is Reboot state, state B is state Functional, and state C is state Connectivity Problem [77]. In Figure 3.1 the value 0.41 means that only in 41% of the cases after a reboot, the system works properly. In 7.2% of the cases after a reboot a connectivity problem appears and has to be solved in order for the system to work properly (see Figure 3.1).

Safety-critical systems constitute a category for which dependability is very important. However, for these systems it is difficult to make safety assessments credible. A way of obtaining incontestable proofs is to perform static analysis on source code. Abstract interpretation and Hoare logic methods are investigated by Nguyen and Ourghanlian [113] in the context of a realistic neutron flux measurement system. The techniques studied were further used by the French Electricity
Department for safety-critical hardware dependent system’s (operating system) dependability assessment.

A way of improving availability by measurements is also to understand a running distributed system. Moe and Carr present a solution to the problem of understanding and modifying a distributed system [105]. The approach chosen is to trace the execution of an application built over a CORBA platform. The tracing is done at the remote procedure call level, by using CORBA portable interceptors to get hold of the relevant information. Next steps after tracing are parsing (of the traces) and visualization. By using these techniques, several flaws in the system design can be identified. Experiments were conducted on a large Ericsson Operation and Management application, consisting of around 600,000 lines of code. When employing tracing, no extra effort from designers and programmers is required. Also, care is taken that system performance is not materially affected.

The results presented in this section are useful when devising failure models, for example, of infrastructures on which applications are built.
Chapter 4

Two fault-tolerant CORBA implementations

We mentioned earlier that fault tolerance mechanisms can be implemented either in every application separately, or in a middleware, thereby providing reduction of application code size. Besides, the middleware can be used as underlying platform for running several types of applications that need fault tolerance.

In this chapter we advocate the encapsulation of fault tolerance mechanisms in middleware. More precisely, we describe extensions of an existing CORBA implementation (OpenORB [39]) with fault tolerance features. The chapter has two main parts:

- Part one presents the extension of OpenORB with fault tolerance which closely follows the FT-CORBA standard. Here, extra infrastructure building blocks are created and assumed as non-failure-prone, as opposed to application units that may fail, and whose fault tolerance capabilities are enforced by this infrastructure. We will call this platform the FT-CORBA infrastructure.

- Part two deals with another extension of OpenORB, towards a fully available platform (FA-CORBA infrastructure). Here, infrastructure units may also fail together with application units. The fault tolerance capabilities of the application are enforced by the infrastructure that executes a distributed algorithm helping in reaching consensus.

Both parts will present challenges and lessons learnt when extending an existing ORB with fault tolerance. The implementation of the platforms was a prerequisite
for benchmarking experiments that study the performance-availability trade-offs. These experiments will be presented in the next chapter.

4.1 The FT-CORBA infrastructure

This section will present the architecture of the infrastructure obtained following closely the FT-CORBA standard. First, a short overview of the FT-CORBA standard will be presented.

4.1.1 The standard

OMG’s generic CORBA Specification was extended to provide application developers with support for fault tolerance. The Fault-Tolerant CORBA Specification V1.0 was adopted in April 2000 [116]. In December 2001, the FT-CORBA specification became part of the CORBA standard. In this section we give a short overview of the standard.

To obtain fault-tolerant applications, the underlying architecture is based on replicated objects (replication in space). Temporal replication is supported by request retry, or transparent redirection of a request to another server. Replicated application objects are monitored in order to detect failures. The failover procedure is dependent on the replication strategy used.

Support is provided for use of active replication, primary/backup and stateless replication. The primary/backup replication styles provided are warm passive and cold passive. In case of cold passive replication, backups are not active and are not updated with primary state and information about processed method calls. The state of the primary is checkpointed periodically, and information - such as arguments, unique identifier, etc. - about update (i.e. non-read) method calls are stored in a log. At failover, all this information is transferred to the backup that has to take over the role of the primary. For warm passive replication, on the other hand, state checkpointing at the primary coincides with transfer of that information to all the backups. Also, information about method calls incoming to the primary are broadcast to the backups and stored there in a log, without being executed. At failover, all necessary information is present at the backup that will be promoted to the primary position. In case of active replication, the standard strongly recommends the use of

\footnote{This type of replication is used when the server object data is accessed only via read method calls.}
4.1 The FT-CORBA infrastructure

A gateway for accessing the active replicas. This gateway plays the role of a relay for method calls - it broadcasts the method calls to all replicas that have to execute them in the same total order.

Some necessary building blocks are specified as interfaces. The Replication-Manager interface incorporates methods from three interfaces:

- The PropertyManager interface has methods used to establish the replication style, the number of replicas, the consistency style and group membership style.

- The ObjectGroupManager interface has methods that can be invoked to support the application-controlled membership style, at the price of losing transparency.

- The GenericFactory interface has the method `create_object()` that in case of replicated objects is called transparently by the replication manager in response to a call to its own `create_object` method by the application. Each object in a group has its own reference, but the published one is the Interoperable Object Group Reference (IOGR).

The specified FaultNotifier interface contains methods for creating event filters, for registering fault consumers, as well as for announcing faults. The fault monitor is not specified as an interface, but its functionality is quite well described.

To be able to manage large applications, the notion of fault tolerance domains (FT domains) is introduced. Each fault tolerance domain contains several hosts and object groups and a separate replication manager, and fault notifier is associated with it. On every host there must be one object factory and one fault monitor. The object factory creates group replicas on demand. The fault monitor has to detect failures of group members running on its host.

Every application object has to implement at least these two interfaces: Pull-Monitorable, containing the method `is_alive` and Checkpointable, containing the methods `get_state` and `set_state`. They are used for the purpose of fault detection and checkpointing. It is also possible that the object implements the Updateable interface, and thus the methods `get_update` and `set_update`, to deal with state updates. This interface is appropriate to use when the state is quite large, while state changes are of reasonable sizes. The saved state (or state changes) information is later used in case the object fails and a new replica has to take over its role (and state). Logging and failover mechanisms are automatically used in case
of infrastructure controlled consistency and membership. They are needed only for passive stateful replication. In case of active and stateless replication styles, request replies can be logged in order to avoid repeated execution of a method call, and directly send the logged reply.

There are some limitations pointed out in the FT-CORBA specification. For example, the ORBs on hosts in the same fault tolerance domain have to be from the same vendor. If this is not the case, then full interoperability is not always possible. Another limitation is related to the types of faults detected and treated. Namely, no correlated (design) faults are treated. Also, no mechanism is provided to treat partition failures. There is a quite simplistic way to protect against Byzantine failures: the Active-With-Voting replication style can be used.

4.1.2 Failure model and assumptions

In the FT-CORBA standard the emphasis is on nodes and not on links. The considered failure model does not include link failures. Links are assumed to be reliable. This means that no message is corrupted by the link. Also, no duplication of messages happens. Therefore, if a message (request) is sent from a correct client process, then the destination server process should deliver the message eventually, as long as it stays correct. The same holds for sending the reply back to the client. If the server process is correct all the time while the request is processed, and the answer is sent to the client process waiting for it, then this message will be received (delivered) by the client, eventually - as long as the client stays correct the whole roundtrip time of the request, so that the connection stays up.

The application on top of the infrastructure is an asynchronous distributed system. There is no bound on message delays and thus no way to differentiate between a slow server and a crashed one. Therefore, all failures reported by failure detectors are just suspicions.

The node/process failure model considered in an application built on top of the infrastructure is the crash model. To be more precise, it can happen that an application server object is no longer able to process requests from clients, because it crashed. Infrastructure server object crashes are not included, the infrastructure being considered robust. Therefore, the failure of an entire host (e.g., because of a power failure) running infrastructure objects as well as application servers is not part of the failure model. Failover information such as state and information about requests serviced since the last checkpoint, is collected in an infrastructure unit outside the application server object. This unit is supposed to continue being up
after the crash of the application server. As a result, a new member of the replica group (residing on a different host) can easily be invested with the role of the failed member.

As specified in the FT-CORBA standard, there is a property called “minimum number of replicas” associated with a replica group. The value of this number is supposed to be at least two. As soon as the number of previously created correct replicas drops under this value, one (or more) new replica has to be created. As already mentioned, with the FT-CORBA standard, more complicated failures such as Byzantine type are not dealt with. Also, the failure of the client object sending requests to the server is not dealt with explicitly.

By state of an object (application) we mean internal (and possibly external) application object data - fields of a class - that can be modified by so-called update method calls. Further, update method calls are assumed to be made only via CORBA calls, i.e. those which can be “seen” by hooks installed outside the application, at the ORB level. If, for example, the application is such that the state can be changed by internal calls (for example made from different threads running in the object) not following a CORBA call path, no guarantee can be made by the FT-CORBA infrastructure about the state consistency of the replicas. And this is the case even in primary/backup replication.

The type (update or read-only) of a server object method is exposed to the infrastructure by the application writer. The Object Factory, written also by the application writer, creates objects that are instances of different classes. When doing this, it has knowledge about the types of the methods for that class. When receiving a client request, while running, the hook underlying the application looks up and uses this information to decide whether to store the call information in the log, in order to be used for replay at failover, if necessary. Read-only method calls are not stored in the log.

### 4.1.3 Infrastructure building blocks

Figure 4.1 summarizes the deployment of the built FT-CORBA infrastructure and application objects on hosts in a fault tolerance domain. The different boxes in the picture will be explained below.

Our implementation of the FT-CORBA standard combines the following three ingredients:

- a collection of service (CORBA) objects
Chapter 4. Two fault-tolerant CORBA implementations

![Diagram of FT-CORBA infrastructure]

Figure 4.1: Deployment of the FT-CORBA infrastructure

- CORBA portable interceptors at the client and server side
- ORB source code extensions

We describe each building block in the three following sections.

Service objects

The service objects in our FT-CORBA infrastructure have their roles clearly specified in the FT-CORBA standard: the replication manager, the object factories, the fault notifier, the fault monitors and the logging and failover mechanism. The standard, however, leaves open some implementation choices. Thus:
• three of the service objects are specified via interface descriptions in the standard: the replication manager, the object factory and the fault notifier. Therefore, we implemented them in a straightforward way by CORBA Objects.

• The fault monitors were implemented as CORBA Objects.

• The implementation of the Logging and Failover Mechanism required some consideration.

• For the active replication style, the standard leaves open the choice of a mediator gateway object. In our implementation client requests are sent to the active group via a gateway CORBA Object.

Standard service objects

As specified in the standard, in a fault tolerance domain there is one replication manager, one fault notifier, and several object factories, fault monitors, and logging and failover controllers (one on every host on which group members are to be created and run), as depicted in Figure 4.1.

The object factory running on a certain host creates CORBA Object replicas, on demand, and starts them in different processes on that host. Every replica of an object group is on a different machine, and on that machine in a different process. Thus, it is possible that on the same host there are several processes running different replicas from different groups.

Failures of group members are detected by fault monitors running on the same hosts as the replicas. The fault monitor is implemented as a separate CORBA Object and it consists of a collection of monitoring threads. Whether one thread is created or not, depends on the fault monitoring granularity, for every object to be monitored on that host. A pull based monitoring style is used. By implementing the PullMonitorable interface, application objects provide the method “is_alive” that is periodically called by the fault monitor. As soon as this call cannot be completed (there is no answer from the server), or the call returns a false value, the object implementing the interface is considered as failed, and the failover mechanism is started. In order to assure that the failure is real, the object is “killed”. This means that the CORBA server replica is deactivated, and the process running it is destroyed. Of course, if the object really failed, because e.g. the process running it is no longer in the system, or because the Java thread of the object failed, then the “killing” is executed in the same way. However, this might not have any effect because, e.g. if the process is no longer in the system, there is nothing left to destroy.
Or, if the thread failed, the replica’s object is no longer accessible and there is no need to deactivate it. The fault monitor reports the failure to the fault notifier. The latter sends fault notifications to its consumers, one of them being the replication manager. The replication manager is in charge of coordinating the failover process that includes also the promotion of a backup to the role of primary, in case of non-active replication.

**Gateway for active replication**

Instead of letting the client forward requests to all replicas of an actively replicated group, our platform uses a gateway CORBA Object. The gateway object is created at the same time as the replicas of the group, and started on a machine from the fault tolerance domain. It is located on one of the hosts where the actively replicated group members reside. As shown in Figure 4.2, the gateway plays a double role, as the mediator between the client and the active server group. Firstly, the gateway broadcasts incoming client requests as calls to all the group members. Secondly, a duplicate suppression mechanism is needed when the server group, acting as a replicated client, has to execute calls to a third tier server. The gateway is involved in this mechanism, as well. Outgoing calls from the members of the active server group, are routed through the gateway, and the same happens to the replies to these calls. When broadcasting client requests to the actively replicated group members, the gateway assigns unique identifiers to the calls and the replicas must execute requests in the order given by these identifiers. If a replica receives request with e.g. identifier \( j+1 \) before receiving the request with identifier \( j \), it must wait for this call, before executing \( j+1 \). As no message is lost on the links, eventually request \( j \) will arrive. The gateway object participates in the reconfiguration of the

![Figure 4.2: Illustration of the double role of the active replication gateway](image-url)
4.1 The FT-CORBA infrastructure

server group in case an active replica crashes. At creation, it registers itself as a fault consumer, to receive notifications about failures of members of the respective active object group. When announced about a failure in the group, the gateway finds out the new membership of the group, so that it can direct future requests to the right replicas. When a new object joins the active group, the gateway handles the state transfer, as well as stopping incoming calls until the state is consistent in the group. Note that by using the gateway, view-synchronous multicast is implemented. Thus, a message (client request) is delivered by all correct members of a group while in the same view. All membership changes are announced at the gateway and request broadcasting happens synchronously with the membership update messages.

Logging and failover service object

The logging and failover mechanism is used only in cold and warm passive replication (see Figure 4.3). The mechanism was implemented on each host as a separate CORBA object (called Logging and Failover Controller). This object’s interface offers methods for logging method call and reply information. Other methods offered by the interface are used to retrieve this information later, at failover. This object also performs checkpointing of the primary server object. At failover, the calls for which information is in the log, are those that arrived since the last checkpoint.

We will come back to the logging and checkpointing operations in Section 4.1.4.

Portable interceptors for FT-CORBA

In our implementation of the FT-CORBA standard, some operations have to be performed around a CORBA request at the client side, as well as before the start or after the finish of the processing performed at the server application object. CORBA portable interceptors offer an elegant way for encapsulating and executing the needed operations at the right places. They were used to perform the following operations on a request:

- adding extra information to the request: the CORBA standard recommends “carrying” this information in service contexts. One such piece of information is the group reference version known to the client. Further, one service context is dedicated to carrying unique request identification information: a client identifier, a request identifier that does not change even if the request is resent, as well as an expiration time. The group reference version is used
as follows: as soon as a change happens in a group (e.g., one member fails) a new version number is assigned to the group reference. The group reference obtained by the client contains such a version number that is or not the same as the one at the server. All these service contexts information are added, at the client side, in a client portable interceptor.

- recording information about a call in a log, as well as the reply for that call. These operations take place in a portable request interceptor at the server side.

- tracing of timing information when performance measurements are needed. This operation is done both in client and in server interceptors.

For different replication styles, application server objects are equipped with different portable server interceptors. For example, in case of stateless as well as active replication, no logging of call information is needed. Only replies are logged in order to detect requests that are resent. In case of cold passive replication, call information is logged only at the primary (see Figure 4.3).

Beside the uses mentioned above, in the present infrastructure, operations performed in portable interceptors are needed when messages (not necessarily in the
form of CORBA calls) are sent to a CORBA object. The interceptor can be used to stop the request before entering the called object that does not implement an interface containing the method that is called.

**Extensions of ORB source code**

When building our infrastructure, portable interceptors were plugged in elegantly using the existing hooks from the ORB code. This way the basic ORB implementation could be left unchanged. However, this augmentation was not enough for the infrastructure to handle all new elements introduced with the fault tolerance standard. This was probably also due to the fact that one certain CORBA implementation (OpenORB) was used. Some classical ORB code (that of e.g. the classes called ORB, POA, Delegate) had to be modified.

In our infrastructure, when primary-backup replication is used, the multiple profile CORBA object reference (IOGR) contains an indication about the primary member of a group. Therefore, to handle this case, the part of the ORB where the request target address is extracted from the object reference, had to be modified in order to be able to choose the address of the primary.

As already mentioned, the unique request identifier consisting of the client identifier, the special request identifier and the expiration is added to the request information as a service context. However, in the client interceptor where this is done, it is not possible to sense when a request is a resent one. Therefore, the unique identifier information has to exist in the request already on arrival at the interceptor level, and every time the request is resent this information should be the same. For this purpose, the ORB code where the request object is created had to be modified. Also, some modifications had to be made in the client stub that is automatically generated by the IDL compiler. Therefore, the code of the IDL compiler was also extended.

Besides class extensions, some new classes had to be defined. When, for different reasons, a request must be stopped in the server or client interceptor and not sent further, there must exist special exceptions that are thrown from the level of those interceptors. An extension is also needed for handling these types of exceptions.

**4.1.4 Logging and failover mechanism**

In this section we highlight some design decisions in the logging and failover mechanism, in order to set the stage for analysis in chapters 5, 6 and 7.
Checkpointing decisions

For cold and warm passive replication, from time to time the logging and failover mechanism reads the state of the primary replica and stores it in a so-called state log (see Figure 4.3). Reading of the state is done by calling the method `get_state` on the primary replica, when no update method is performed on the object. This means, on the one hand, that as soon as the `get_state` call is made on the server object, it has to wait until all previously arrived update calls finish their execution on the object. Analogously, any update method call that is currently incoming to the server object has to delay its execution until a currently executing `get_state` method call finishes. In this context, some further problems arise: must the call to `get_state` be made periodically? If this is the case, then how often should the call be made? This is interesting because the average roundtrip time for requests is influenced by the waiting time for `get_state`. Also, if the state does not change so often, it might be better to save the state only after it changes. This would lead to “event triggered” checkpointing.

In our experiments we used periodic checkpointing in the following way: a separate thread in the logging and failover object starts “from time to time” the state reading procedure by registering the call to `get_state` in the logging and failover object, then putting it to wait in the queue of the application object and finally read and record the state; “from time to time” means that a fixed time (period) after the state recording, the checkpointing thread starts the procedure again. If we take for example, the value 1 second for the checkpointing period, it means that each time after the state recording happens at the end of a checkpointing procedure, 1 second later a new instance of the procedure is started. In our studies we chose the values for the checkpointing interval on a purely empirical basis, with no underlying mathematical support. Ideally, we would like to provide guidelines for choosing the checkpointing period, to optimize, for example a certain attribute of the system: availability for processing client requests, or response time of client requests. We will come back to a more informed choice of the interval by mathematical modelling in Chapter 7.

As soon as the state is read from the primary and is ready to be stored in the log, the current record residing there is removed and replaced with the new one. In the call information log, records of update method calls arrived before the state read request arrived at the server are deleted, as their changes are reflected in the state of the object obtained by `get_state`. If the state read from the primary is very large it can be problematic to transfer it to the log, as well as keeping it there, even
4.1 The FT-CORBA infrastructure

if only one state record is present in the log at a time. In such situations it could be helpful to read and store only state changes instead of whole state records. Here, we assume that the sum of state changes that are stored is smaller than a single large state record. To be able to obtain state changes (or updates) from the object, it must implement the Updateable interface, offering the two methods `get_update`, and `set_update` for reading and writing (or setting) a state change. There are two problems with this approach:

- first, all state change records have to be kept in the log, in order to be able to reconstruct correctly the state at failover. So, if many state changes are recorded until the failure occurs, it is probable that the storage space required by them becomes larger than that required by one single, even large, state record.

- second, if the fields of the application object are rather complex, as well as the changes to them, then it is not straightforward to express these changes as a simple array of bytes, to be used when writing this state in the object. Since the code for the `get_state/set_state` or `get_update/set_update` methods is written by the application developer, it is important to impose feasible demands.

It is possible to use a combination of the two approaches: at larger intervals (say every 2 hours) to perform a state reading, thereby removing all update or call information records reflected in the new state, and at smaller intervals (say every 3 minutes), between the state readings, to perform state change readings. In our implementation we used only the periodic state reading approach. Experimental results are presented only for this variant of checkpointing.

Logging of method call and reply information

In primary-backup replication style, besides state information, update call information has to be logged at the application server, for later replay purposes at failover. There is no need to log read-only method calls, because they will not be replayed, since no changes on the server state are induced by these operations. It can happen, however, that after a failure, a new primary is set up and a request involving a read-only method call is resent by the client. Thus, the read-only method will be executed again and it is possible that it returns a different answer than the first time. However, this is not a problem, because the client is interested anyway in the latest reading result.
To explain the execution model of requests on the server, we will first formulate the following statements: (1) we know that, at failover, the only information available for replay are the call information records from the log. (2) the order of replay will coincide with the order of logging. (3) we consider the state of the application object to be composed of several independent portions (see, for example, the fields of the class ExampleClass in Figure 4.4). By independent, we mean that changes performed on, for example, \( v_1 \) and \( v_2 \) during the execution of different methods, are allowed to happen in any order. (4) we do not want to make any assumptions about commutativity of changes performed on the state, e.g. that changing \( v_1 \) by two methods in different orders should be the same. (5) at middleware level, where logging of call information happens, there is no knowledge about when/where, during their execution, methods access independent portions of the state of the object.

From statements (1) and (2) we can conclude that the state changes on non-independent state portions, at replay, happen in the same order as the call records are logged. From statement (4) we conclude that, in order to obtain in the backup a state consistent with the one of the primary before this failed, the state changes on non-independent state portions at replay have to be the same as those changes were executed on the primary while being alive. These imply that the order of state changes executed on non-independent state portions, in the primary, must coincide with the order of logging call information records. Since, according to statement (5) we cannot differentiate, at middleware level, between changes performed on non-independent or independent portions of the state, within methods executing on the server, we conclude that we must order all state changes (on non-independent as well as independent state portions), according to the order of call information records in the log. Now, logging is done separately from call execution on the application, and besides, we do not make assumptions about the threading policy of the ORB. Therefore, the best way to enforce the above mentioned property is to impose, from the middleware level, sequential (serialized) execution of the calls on the application. This is obtained simply by queuing requests at the server interceptor level, and letting the next request to leave the queue and proceed to the application object, as soon as the previous request was executed on the application. In warm passive replication, where call information is broadcast to the backups for logging, the order of logging is enforced by assigning consecutive identifiers to broadcasted information (as in active replication). The backups record call information in the log in the increasing order of the identifiers. Of course, the order of assigning identifiers coincides with the order in which the calls are executed on the
4.1 The FT-CORBA infrastructure

```java
class ExampleClass{
    int v1,v2,v7;
    boolean v3,v4;
    StructureA v5;
    StructureB v6;

    void method_a(...){
        ...
    }
}
```

Figure 4.4: Object state portions

primary object.

Logging of call replies, or at least of a note that the request was already executed once, is done at the server side interceptor level, with the purpose of avoiding reexecution of a method call. There are two cases:

- for update method calls - the reason here is to avoid changing the state of the server twice, by reexecuting the method that was called. Thus, even in case when the call does not return a value, or even more, it is a one-way method call\(^2\), there has to be a note that the method was executed and changed the state accordingly.

- for read-only method calls - the reason here is to gain time. On the other hand, as already mentioned, it is reasonable to assume that the client always wants the latest result of a reading, regardless of when the request was sent. Thus, if the reply to a request does not return on time, or for some other reason the request is resent, obtaining the latest result is just as good as to get the answer back with less extra delay. Independent of whether read-only call replies are logged or not while a replica is correct, there is definitely no need to log these replies for use by the new recovering primary.

Table 4.1 summarizes the call and reply logging points at the server side (the only place where these operations are done).

To our knowledge, this is the first fault-tolerant CORBA infrastructure closely following the CORBA standard. The next section will present a CORBA middleware supporting application and infrastructure fault tolerance that does not respect the CORBA standard, but is more robust than the FT-CORBA infrastructure.

\(^2\)The sender does not wait for an answer for the call
4.2 The FA-CORBA infrastructure

As explained in Section 4.1.3, the CORBA specification provides no elegant way to handle failure suspicion notifications. The straightforward solution is to turn the crash suspicions into real crashes, by “killing” the suspected server object. Besides, the CORBA fault tolerance standard defines infrastructure components that, unless replicated in a possibly ad-hoc manner, constitute single points of failure.

Here we present an approach that deals with these problems. We employ an improved version of an algorithm originally developed by Dutta et al. [43]. Our improvement to the algorithm does not affect its availability aspects, but the performance when executing failover operations (defined in Section 2.2.3). In this approach, infrastructure units handling fault detection and failover are distributed and fail together with application units. Therefore the algorithm executed by the distributed infrastructure handles failures of the whole system. Thereby it assures full availability. The working of the algorithm does not depend on any particular middleware, but it is interesting to implement it in a real middleware to test its characteristics. This section presents a CORBA based implementation of the algorithm. The infrastructure obtained in this way will be called the fully available (FA) CORBA infrastructure. The notion of transparency is kept. Also, the application developer still concentrates on application functionality, and extends the code with very few elements demanded by the infrastructure developers. In this section we focus on the embedding of the algorithm in a CORBA infrastructure. Architectural elements together with implementation issues will be presented. Also, the improvement to the algorithm will be explained.

<table>
<thead>
<tr>
<th>before executing request on application object</th>
<th>after executing request on application object</th>
</tr>
</thead>
<tbody>
<tr>
<td>(at receive request in server interceptor)</td>
<td>(at send reply in server interceptor)</td>
</tr>
<tr>
<td>try to retrieve call reply</td>
<td>notify queued update methods</td>
</tr>
<tr>
<td>wait for update and get_state methods to finish</td>
<td>log call reply information</td>
</tr>
<tr>
<td>log method call information</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Logging related operations in the server interceptor
4.2.1 Architecture units

The system model used by Dutta et al. [43] incorporates a set of processes, that can suffer crash failures, without recovery. Communication links do not duplicate messages, do not create messages, and any message sent on a link is eventually received by its destination, if this one is not failed. For the algorithm to terminate, at least a majority of processes must be correct, thus being able to participate in decision making. Even though similar, the algorithm is different from active replication in two ways: servers can have non-deterministic behaviours and there is no parallel independent processing of requests after agreeing on their total order. For the latter, an agreement protocol is run only when the result of a request is sent back to the client. This allows the possibility of having two or more server replicas attempt to execute the same request and perhaps temporarily obtain two different results. Also, the order of request processing is not agreed upon. The outcome is made unique by enforcing in every replica the updates and results obtained by the current leader, which is the node sending the result back to the client.

To achieve this, two conceptual building blocks are involved in the fully available infrastructure: (1) a consensus service used for making sure that replicas agree on the result to be sent back to the client, as well as on the state modification enforced on all replicas, and (2) a leader election service that is used to determine the replica that is supposed to process requests and reply to the client.

The application replicas are the participants in the algorithm. They are equipped themselves with unreliable failure detectors (see Section 2.1.4) incorporated in a leader election unit, as well as with consensus units (see Figure 4.5(a)). The type of failure detector used is the weakest one to solve consensus (see $\diamond W$ in Section 2.1.4). The failure detector is queried from time to time via the leader election unit, about the identity of the leader. As the failure detections are imperfect, there may be two or more servers that consider themselves to be leaders at a point in time. In such a situation, the consensus helps resolving the resulting conflict. A replica that is not leader is called a witness in the sense that it helps in the election of the unique leader, and it stores the outcome of the real leader’s actions in an total order to be agreed upon.

Figure 4.5(a) shows the deployment of infrastructure units present in each server replica. Besides the leader election unit (LEU) and consensus object (CO), the actual unit performing the computations employed by the request processing is the application server object (ASO) local copy in the server interceptor (more details will follow shortly). The failure detector is set up as a separate server thread, at-
tached to the application CORBA object. Being part of the leader election unit, the failure detector is supposed to receive *I-am-alive* messages from other LEUs at other application replicas. Note that we assume the leader election unit (and the failure detection unit) to be sending correct *I-am-alive* messages on behalf of the application replica. In other words, the fault tolerance approach properly deals with crashes of the whole node. The absence of an *I-am-alive* message leads to suspecting the application replica to have failed, although the message should have come from a separate unit. The leader, determined by the LEU has the lowest index among all replicas that are considered not failed.

Figure 4.5(b) shows the deployment of architecture units at the client side. The consensus object is missing here, but the leader election unit is present and has the same role as in server replicas. When the client wants to send a request to the server group, it transparently queries the LEU to find out the identity of the leader from the client’s point of view.

### 4.2.2 Infrastructure interactions

This section will describe major interactions inside the fully available infrastructure. The picture in Figure 4.6 visualizes the flow of information between client and server group, and between server replicas when processing a request either when no conflict and failures arise, or when disagreement or failures intervene.

The dashed line boxes containing ASO copy suggest that the server interceptors make method calls on the application object from the interceptor level (see
The FA-CORBA infrastructure

Figure 4.6: Client-Server interactions in the FA-CORBA infrastructure

arrow with execute() in Figure 4.6). The thick dashed arrows that connect two consensus object boxes or two leader election unit boxes, represent message flows by using simple socket connections, not involving CORBA calls. The reason for not using CORBA messages where possible is to improve performance as well as bandwidth usage. The thin dashed arrows inside the client solid host box represent the portion of the call from client to server that spans from the ORB to the client interceptor (CI). The thick solid arrows represent flow of CORBA calls. Finally, the thin solid arrows denote flow of method calls that are done via socket connections (see e.g. leader(), propose() or execute()).

I-am-alive messages that flow among LEUs at different replicas are used to determine who the leader is. The client, in order to enable its leader election unit to receive I-am-alive messages from LEUs at server replicas, has to register with the server group. Registration is done via sending I-am-in messages to all members of the replica group as soon as the application is started. This message carries with it
the address in the client where the future I-am-alive messages have to be sent, i.e. the address (in terms of host and port) of the client’s leader election unit.

By concentrating on the solid arrows (thick and thin) in Figure 4.6 we will describe the working of the algorithm in a non-failure scenario.

1. arrow from ACO to FA-ORB: client initiates the call to method \texttt{m1};

2. arrow from FA-ORB to LEU on \texttt{Host}_2: the client’s LEU is queried about the identity of the leader;

3. arrow from client CI to server interceptor SI on \texttt{Host}_1: client sends request corresponding to call to method \texttt{m1};

4. arrow from SI (ASO copy) to ASO: call to \texttt{execute()} meaning tentative execution of the method by the application object. Tentative means that no durable change of the object state is made so far;

5. arrow from SI to LEU with \texttt{leader()}: the LEU is queried about the identity of the current leader;

6. arrow from SI to CO: if the LEU on \texttt{Host}_1 indicates leadership, then \texttt{propose()} is called with the outcome (meaning result and state change) of \texttt{execute()}

While performing step 6 above, the leader tries to write the outcome of its action in a unique position of the stores maintained at all witnesses plus itself (see the sequence diagram in Figure 4.7 for better understanding the flow of actions). It achieves its goal only if a majority of the replicas acknowledge its leadership. The dashed arrow between CO on \texttt{Host}_1 and CO on \texttt{Host}_2 implements this by a series of READ, WRITE and Ack/Nack (see also Figure 4.7).

Note that the eventual leadership property of the leader election algorithm used, is translated in the usage of a timer at the client side that establishes a time after which the request is resent if no reply is received. Thus, each time before executing steps 1, 2, 3, the client sets a timeout in its timer. If the reply did not arrive when the timeout expired, the request is resent.

4.2.3 FA-CORBA implementation

This section provides some details about the algorithm’s implementation on top of CORBA. Note that the ORB that we extended was OpenORB like for the FT-
4.2 The FA-CORBA infrastructure

CORBA infrastructure. We used portable interceptors for executing the algorithm and extended the ORB source code.

**Portable interceptors**

The main CORBA specific elements used to implement the algorithm and the FA-CORBA infrastructure were portable interceptors. Client side portable interceptors are employed to augment the client request with a service context containing the unique request identifying information. Server side portable interceptors are mainly used to intercept I-am-in registration messages from clients, as well as normal client requests. Once a client request is received, all operations devised in the algorithm are performed at the level of the interceptor. Thus:

- When an I-am-in message from the client (sent via dynamic CORBA invocation) arrives in the interceptor, the call is stopped from reaching the application object. For this, an exception is generated and further handled in a well-defined code of the ORB. Before the exception is generated, however, the leader election unit of the intercepting replica is informed about the address where I-am-alive messages have to be sent later.
• As soon as a client request for an application method call arrives in the interceptor, it has to wait until the previous request finishes execution on the application object (note that waiting is an inherent feature of the algorithm, and is not implementation related; it does not make sense to agree on the output of one operation on the server, when the server may execute several operations in a multithreaded manner). After the waiting interval, the algorithm steps 4, 5, and 6 mentioned in Section 4.2.2 are executed. The tentative execution of the requested method call (step 4 of the algorithm) on the application is simulated in three steps: (a) reading of the replica state by using the method `get_state`; (b) executing the needed computations on the application replica, so that the result, if any, is returned and the state of the replica is changed; and (c) reading the state change from the replica by using `get_update`. After this execution, if the replica in question is not a leader, or if it is locally seen as leader, but its proposed outcome is not the decided one, the state read before the execution is re-instated on the replica.

Note that if a request is made for a read-only method call, the server replica does not execute the consensus algorithm. After executing the reading operation the replica sends the reply to the client and stops returning to the point where it waits for a new request to arrive. Here, no leader election unit querying takes place.

It is also important to note that the only published references (addresses) are those of the application server replicas. Therefore, the I-am-in messages are sent via CORBA dynamic request invocations to these addresses. I-am-alive and other infrastructure related messages are sent using simple reliable socket connections, not involving CORBA specific overhead.

**Modifications of ORB source code**

Similar to the FT-CORBA infrastructure some ORB source code modifications were made. The good part was that some extensions required by the FT-CORBA implementation could be reused. However, some new modifications had to be devised. For example, the code of the client ORB that parses the object reference and decodes the address where the request has to be sent (target address) had to be extended. In a normal CORBA setting, the client has a fixed address of the server in the form of an IOR (Interoperable Object Reference) and the target address is chosen after extracting port and host information from the IOR. To cope with the
need to query the LEU at the client, the modification consisted of introducing the leader election algorithm execution as the part returning the target server address.

Furthermore, the client ORB code had to be modified to handle timeout in a transparent way. In a classical ORB, there is a code portion containing a routine that handles the case when a reply does not return before a certain timeout. However, this only generates an exception supposed to be handled at the client application level. To avoid the generation of the exception, we decided to modify the code so that the request is resent in a transparent way, as envisioned by the description of the client side behaviour of the algorithm [43].

**State transfer for better performance**

To clarify our improvement of the original algorithm of Dutta et al. [43], let us detail the activities performed by a witness including the moment when it becomes a leader, in the original algorithm (see also the sequence diagram in Figure 4.8 for a clearer picture). The interested reader may also refer to a more detailed description of the algorithm elsewhere [147].

During normal processing periods, that is, when no replica fails or is wrongly suspected to have failed, the leader transfers the update method execution result and the state change to the witnesses as soon as a request is processed. All state changes are stored at witnesses, possibly in a structure residing in memory. With no further intervention (as in the published algorithm [43]), the storage can be filled up with state change structures. Therefore, one limitation of the algorithm was the possible infinite growth of memory needs.

The second limitation is related to the moment when a witness becomes a leader. This can happen either because the previous leader really failed or because the leader suddenly became very slow and the client as well as the other replicas do not recognize its leadership anymore. After this, when the “former-witness-now-leader” replica receives the first request from a client, it has to bring its state to the value found in the previous leader at the time of its abdication. Before the improvement we made, this state was built in the new leader by incrementing the initial replica state, using the series of state changes stored at the other replicas. Obviously, the time for this procedure grows in an unbounded manner the further we get from the start-up time.

The original paper already proposed a way to solve the problem in the future. When a witness receives a state change record from the leader replica, it will immediately apply it on its own local copy of the application object, instead of only
storing it. Thus, storage needs are reduced almost to zero. Also, the new leader does not need to reconstruct its state by using information from witnesses anymore, since it has the current state itself.

Our platform implements a different improvement. In this setting, from time to time, the leader sends entire state information to witnesses. When receiving it, witnesses prune their storages, that is, replace the set of existing state change records with only one state record. Thus, the storage is dimensioned so that it accommodates a limited (quite small) number of records. Consequently, the failover process in the new leader will take a limited and relatively short time to execute. It is possible for the leader to trigger the pruning periodically (say every 5 or 10 s). Another alternative is to send state after executing a certain number of requests: say after every 20 requests executed, the leader sends its current state to the witnesses.

The second alternative offers the possibility to dimension the storages to accommodate a known number of records. In the present implementation, the periodic alternative was used, since it offered a direct way to compare with the passive replication case in the FT-CORBA implementation, where checkpointing of state
was done periodically.

4.3 Related work

Prior to the specification of the FT-CORBA extension, few works had studied alternative augmentations of CORBA with a process (object) group module. These works, however, were ones to contribute to devising the FT-CORBA standard. Performance assessments of these platforms were done in the context of artificial applications, and little trade-off analysis was conducted.

Felber et al. show some possible approaches for introducing object groups in CORBA [49]. Three different approaches are presented depending on the position of the group communication module relative to the ORB\(^3\): integration approach, interception approach and the service approach. If the integration approach is used, then the group toolkit is integrated in the ORB. Thus, each request to the ORB directed towards a process group is sent as a multicast message by the group toolkit. If the interception approach is used, the ORB is totally detached from the group toolkit. There is an extra layer between the group toolkit and the ORB - the interception layer. The “object service” approach is the most CORBA-oriented. The Object Group Service (OGS) is specified as a set of IDL\(^4\) interfaces, like any other CORBA service. So, it does not depend on implementation language and it can be used with any compliant CORBA implementation. The service is composed of several distributed objects, avoiding single points of failure. Failure detection of remote components is provided, as well as distributed agreement protocols.

With the OGS, clients are not aware of the fact that they are invoking methods on a non-singleton object. Thus, transparency of invocations is ensured. The OGS is constituted from a set of CORBA services specified separately and interacting through the ORB, defined as components. These are:

- a Messaging Service that provides non-blocking reliable point-to-point and multicast communication
- a Monitoring Service, that provides application object failure detection mechanisms
- a Consensus Service, that allows group members to solve the distributed consensus problem

\(^3\)Object Request Broker
\(^4\)Interface Description Language
• a Group Service that solves group multicast and group membership problems, using the Consensus Service

The OGS can be used in two configurations: as a linkable library model or as daemon model. The first model brings application objects and service objects in the same address space. In the second model, service objects are located in a different process on the same host or on a remote host. In the first case, both application objects and service objects have to be written in the same programming language.

Measurements of the OGS performance were done with an artificial application where the client sends 100 synchronous invocations in one round. The results were averaged in every round and the best result was kept. Several experiments were done using the linked model or the daemon model, with group sizes varying from one to ten. Also, total order multicast was used, as well as reliable and unreliable multicasts. The observed results showed that the latency of total order and reliable multicast grows fast with the size of a group.

Narasimhan et al. [110] propose an interceptor approach to enhance CORBA with fault tolerance. The result of the research work in this direction is the Eternal System. Interceptors can be used for several other purposes as well: real-time scheduling, profiling and monitoring, as protocol adaptors for protocols other than IIOP and for enhancement of security. By using Eternal, fault tolerance can be added to an application built over a CORBA implementation, without modifying the ORB or the application. This fault-tolerant CORBA infrastructure is not entirely conforming to the CORBA standard. There are two interception mechanisms used by Narasimhan et al.: system call, respectively library routine interception. In case of the first mechanism interceptors are placed at the operating system level, to modify actions of certain system calls. The UNIX /proc file system is used, where images of processes can be found. The files can be accessed via specific operations and the information in them modified, thereby indicating what system calls to intercept and how to act for them. In case of library routine, interception is based on the possibility to add shared objects to a process when it initializes.

When a system has to be enhanced to provide fault tolerance mechanisms the interceptors are used in several directions:

• for reliable multicast - as protocol adapter - an IIOP message is parsed at the operating system level, and then it is passed to a reliable multicast protocol like Totem to be further sent to the group replicas;

• for consistent multi-threading - as scheduler - in case of multi-threaded objects or processes with multiple single-threaded objects sharing data, the
thread dispatching is done so that mutual exclusion is achieved when needed;

- for replication management - duplicate message suppression, logging of messages, state transfer to new recovering replicas.

Replica state consistency related problems are pointed out in the context of Eternal [111]. Several components of the replica state are delimited: application level state (read from time to time by using method calls specified in the FT-CORBA standard), ORB/POA\(^5\) level state and infrastructure level state, specific to Eternal. The ORB/POA level state has two parts: GIOP\(^6\) request identifiers (for outgoing server messages) and client-server handshake messages. These data have to be transferred to new or recovering replicas, in order for them to be able to resend a message (if necessary) with the same identifier like before. This cannot happen if request identifiers in the recovered server start from a default initial value. The third component of the replica state is stored by the Recovery Mechanism of Eternal and is made up of information regarding requests processed by the server, invocations sent out by the server for which it awaits replies, replication style of the replica group, etc. In the works with the Eternal system, overheads given by the interception, multicast and replica consistency mechanisms were measured.

As a summary of experiences with FT-CORBA, Felber and Narasimhan, two of the main contributors to the standard, published a journal article in 2004 [50]. Chung et al. present a fault-tolerant infrastructure (DOORS) built using the service approach, on top of CORBA [28]. In this setup, application objects register to DOORS in order to be made fault-tolerant. Fault tolerance services are realized with two components: ReplicaManager and WatchDog.

Using DOORS as an FT-CORBA implementation, Natarajan et al. [112] present ways of improving the performance of an FT-CORBA infrastructure. The focus of the authors is on the Replication Manager and Fault Detectors. Different design, architectural, and optimization patterns are applied in order to reduce time to detect faults, and further, failover times. Experiments were performed using warm passive replication style, with always creating a new backup in order to maintain the size of the replica group. The timing values obtained after the measurements show the need to do something in order to optimize the system.

Mishra et al. [104] describe a CORBA group communication service based on a service approach. The design and two implementations are presented. A UDP\(^7\)

\(^5\)Portable Object Adapter
\(^6\)General Inter-ORB Protocol
\(^7\)User Datagram Protocol
socket based implementation is chosen for comparison purposes. The CORBA version is preferred, because of the possibility to have heterogeneous distributed systems in terms of application parts written in different programming languages, or machines with different operating systems. The evaluation of the two implementations is directed towards measuring throughput of updates broadcast by group members, delivery time, stability time, and number of messages per broadcast. Results clearly show the superiority in performance of the UDP based implementation. Timing figures differ with a ratio of approximately ten, i.e. stability and delivery times are around ten times longer in the CORBA implementation, while throughput is ten times smaller. Number of messages per update is around two times more in the CORBA implementation. The conclusion of the work is that CORBA is suitable for building the group communication protocol in systems that do not require high performance.

A framework for fault-tolerant CORBA services with the use of aspect oriented programming is developed by Polze et al. [122]. The goal is to provide the application writer with the possibility to build a fault-tolerant application by choosing the types of faults to be tolerated. The toolkit then chooses the appropriate fault-tolerance strategy. Communication between a client and a replicated service is done via an interface object. This object, a sort of gateway, may be equipped with evaluators to be able to detect possible computation faults in the replicas. Due to it being a single point of failure the interface object poses interesting problems that are however left to the ORB vendors. Non-functional aspects of a system are described and used as starting points when building the framework for creating fault-tolerant services. Some of these aspects are: the fault tolerance aspect that describes fault tolerance techniques used by a component, the timing aspect describing timing behaviour of a component (average and worst-case execution time), or the consensus aspect implemented by application voter objects.

Killijian and Fabre [81] describe a framework for building fault-tolerant CORBA applications by using reflection. A meta-object protocol is defined, and an open compiler is used, to be able to extend CORBA objects with wrapper methods. These wrappers help the state capturing mechanism, by always saving the state when the wrapped modifier method is called. By this, of course, the application writer is relieved of the effort of writing get_state or get_update methods.

Zhang and Zhang [167] study trade-offs related to replica consistency for performance and availability. The approach is not to fix the level of consistency stati-
4.3 Related work

...cally in one of the extremes, strong consistency or eventual consistency, in a static way. The aim is to adapt the level of consistency depending on the client tolerating a certain obsoleteness of data. Thus, consistency can be tuned to provide higher availability and better performance, if applications are satisfied with having up-to-date data later than it was modified. The analysis and adaptation algorithm is based in enlarging or squeezing the so-called update window. This is a measure of how many updates a replica can buffer before it sends them to all replicas. If the window is enlarged, then the other replicas might have outdated results for a longer time than when the window is small (e.g. having value 1).

After the FT-CORBA specification was made part of the CORBA standard, still not so many works evaluated trade-offs, and feasibility of implementing the standard.

Friedman and Hadad [57] present a fault tolerance supporting CORBA infrastructure built using a combination of portable interceptors and the service approach. The infrastructure does not totally conform to the FT-CORBA standard, as other design goals were more important than conformance (e.g. one goal was to provide partition failure handling, which is not provided in the FT-CORBA standard). A special portable adapter, the Group Object Adaptor is used. The work only provides support for active replication of servers. Communication and node crash failures are handled. An underlying group communication subsystem is employed to provide consistency. No single points of failures are present. Transparency to clients is provided only partly, since clients obtain a handle to one of the replicas and communicate only with it. This one replica is responsible to diffuse the result of its operations to the other replicas. The infrastructure is portable and interoperable. No performance measurements are made on applications running on top of the platform, and no trade-off analysis is provided. Only some remarks are made about how good performance is achieved by using active replication and clients connecting to only one server.

Yue et al. [165] use CORBA portable interceptors to obtain a fault tolerance supporting infrastructure by linking an ORB platform (ORBUS) with a non-CORBA fault-tolerant framework, EDEN. Only active replication is used, where it is assumed that applications are deterministic. EDEN has the role to forward CORBA messages from a client to all replicas of a server, in the same order. It uses a sort of agreement algorithm for deciding the identifier of the request to executed. Portable interceptors are used to add extra information to a request such as group identifier and request identifier, or to extract this information to be able to execute requests in the right order. The work does not provide performance or trade-off
Narasimhan [109] surveys trade-offs between real-time and fault tolerance in middleware applications. The starting point is that CORBA standardized both types of support (RT-CORBA and FT-CORBA). However, it is not straightforward to implement and use a CORBA implementation with both features, because their presence sometimes is conflicting. For example, it is hard to predict occurrence of a fault and further the time needed for a server to execute a request and reply to a client when the system recovers from failures. It cannot be predicted how soon the failure will be detected, and the situation of the server’s state, for example, at the time of the failure. These properties are not feasible to appear in a real-time system were predictability and boundedness of the time spent to execute an operation are crucial. Therefore, using FT-CORBA in a real-time system with strong predictability requirements as indicated by the RT-CORBA standard, becomes very difficult.

As a consequence of the above observations, Narasimhan et al. [20, 135] developed a middleware for real-time fault-tolerant applications called MEAD. In this infrastructure, attention is paid to, for example, obtaining predictable failover times, developing a tool to assist the decision of assigning fault tolerance properties, as well as even providing run-time feedback about resource availability and usage. Fault tolerance features such as state consistency can be configured to obtain better and predictable timing values. Based on resource availability (for example, due to a failure, some resources can be temporarily not available) the system can negotiate new quality of service values with clients, if possible. The resource manager is implemented in a decentralized way such that on every node there is one such module, to avoid single points of failures.

**Final remark**

The work presented in this chapter is, to our knowledge, the first to deal with building a fault-tolerant CORBA infrastructure closely following the CORBA standard. Furthermore, it is a work that contrasts the FT-CORBA implementation with an implementation not conforming to the standard. The next chapter will present trade-off analysis in the two infrastructures. A realistic telecom application is run on top of the middleware platforms for performing measurements.
Chapter 5

Empirical studies with a telecom application

This chapter describes our findings when running a realistic telecom application on top of each of the middleware platforms with support for fault tolerance presented in Chapter 4.

The quantitative results presented are in terms of roundtrip time overheads and failover time. Besides experimental results, we will also bring to light some of our reflections about the feasibility of implementing FT-CORBA, or the trade-off for achieving full availability.

The obtained numerical results are significant mainly in a comparative context, and increase the knowledge about trade-offs in both platforms.

We do not claim by any means that experimenting with only one application via a quite limited number of experiments can lead to immediate generalization. However, the study gives a flavour of what problems appear in these contexts.

5.1 Experiments with the telecom application

This section will describe the application we used in our experiments. Also, the setup in our experiment such as computation resources used, replication parameters, or client request patterns will be described. Finally, the performance metrics will be briefly presented.
5.1.1 The telecom application

Earlier work in the area [49, 111], has presented experimental results when using artificial applications and test cases. These are easy to set up, however, not always provide results that can be compared to practical deployment scenarios. The advantage is that infrastructure properties are quite easy to separate from application-induced ones in this case.

To shed a light on more practical settings, a realistic telecom application was used to perform our experiments. This realistic application is a generic service in the operations and management (O&M) part of the radio networks, together with artificial client side test cases, provided by Ericsson Radio AB. The application is a so-called Activity Manager server. The functionality of this server consists of creating jobs consisting of activities that have to be scheduled immediately or at later times. Also, after creating an activity, the server receives activity status and progress reports. The state of this server is made up of internal structures storing the collection of jobs and activities. These are managed according to what the clients (other parts of the O&M network) ask for. The server’s functionality is described by fifteen methods for job and activity creation, as well as activity and job start and stop operations. The size of the server application is around 10,000 lines of Java code. Our industrial partner chose this service as it has some generic characteristics representative for other applications as well.

5.1.2 Experiment setup

We performed our experiments on SUN Ultra SPARC workstations, running SunOS 5.8, with no isolation from other tasks running on the hosts. The machines were connected via an IP network where the links were not used exclusively by the communication in our infrastructure and application. The reason for not performing experiments in a controlled environment was to mimic the realistic setting where the service will eventually run. For both platforms, in all our experiments the clients were unchanged. The comparison baseline when devising our results was a non-replicated server where no FT support in the ORB was included. Note that in this way neutrality to different replication styles is preserved, as opposed to comparing to a setting with application level replication, which could resemble one of the styles.

There were two goals for performing experiments: to measure roundtrip time overheads and to measure failover times. Roundtrip time overhead is the extra time spent by a request on its way client-server-client due to different fault toler-
5.1 Experiments with the telecom application

ance handling related operations; roundtrip time (see figures 5.1(a) and 5.1(b)) is exactly the time spent on the “round trip” client-server-client; the overhead for a certain request is obtained by subtracting the value of the roundtrip time in the non-replicated setting from the value of the roundtrip time when sending the same type of request in the fault tolerance enhanced middleware. Measurement probes were placed in client and server interceptor units to record important time points in the trip of a request from client to server and back. To avoid probe effects, null probes were inserted in the non-replicated case in the same position for each interceptor unit.

For the measurements on failover times, the baseline was chosen as the time taken from the crash of a (non-replicated) server to its restart (possibly manually as it is the case in the real setting). In the different platforms, the failover time is affected by the replication style and is computed as follows, for the FT-CORBA and FA-CORBA platforms respectively:

- In cold passive replication, it is the time taken to set the state on the new primary, plus the time needed to replay the requests arrived since the last checkpoint.
- In warm passive replication, it is the time spent in replaying requests.
- In active replication it is the time spent to reconfigure the server group.
- For the FA-CORBA platform, to measure failover times, the current leader is forced to fail after a certain number of processed requests. Then, a new leader is elected and forced to bring its state up-to-date as soon as the next query arrives.

To obtain relevant results, the number of requests processed before the failure was chosen to be the same for both warm and cold passive replication in the FT-CORBA platform and for the experiments with the FA-CORBA platform. For measuring overhead, the clients used in the tests were simple: they called six of the update methods of the server in a loop. The code of some of the methods contained calls to third tier server methods. This way the duplicate suppression mechanism of the gateway, when in active replication, was tested. The clients called the methods in loops of 100 and 200 iterations. Some of the methods were called once per iteration (Method 1, 2, 3, and 5), others two (Method 6), and four (Method 4) times, respectively. The results are presented as averages of the measurements for every method call computed over runs in both types (100 and 200 iterations) of
loops (hereafter called one experiment). The averaging was done in order to even out the different network and processor loads at the time of the experiments.

In our experiments we varied two parameters: the replica group size and the checkpointing/pruning interval (where applicable). We expected mainly these two attributes to influence the results. The group size was expected to influence the roundtrip time overheads in the following cases:

- in warm passive replication - due to the broadcasting of state to the backups;
- in active replication - due to broadcasting request to all replicas;
- in the FA-CORBA infrastructure - due to the execution of the consensus algorithm which involves a decision from a majority of group members.

The checkpointing/pruning interval was expected to influence both roundtrip time overheads (due to interference of client request processing with checkpointing respectively pruning operations), and failover times (due to number of requests to replay at failover: most probably high if checkpointing/pruning is done seldom).

In both platforms group sizes of 2, 3, 4 and 8 replicas were used. In the FA-CORBA platform, the group size was of special relevance, since it reflects different majority rules. In the FT-CORBA platform three different checkpointing intervals were used: 1, 5 and 10 seconds. In the FA-CORBA platform, the pruning interval was chosen in a similar way to make valid comparisons. Note that the values for the group size was chosen especially guided by its significance in the FA-CORBA platform, where depending on the group size different majority group sizes are obtained.

All in all twelve experiments were performed for each of the following styles: warm passive, cold passive, fully available replicas (corresponding to 4 replica configurations and 3 checkpointing intervals). For the active replication four experiments were performed (corresponding to 4 replica configurations).

5.1.3 Measuring roundtrip time overheads

The slices in the roundtrip time were used to identify dominant parts of the overhead, i.e. where the largest $\delta$ appears. Shown in Figure 5.1(a) the slices that appear in a non-replicated setting as well as in replicated scenarios, where they are larger, can be described as follows: the time spent in the client interceptor (when adding information to the request) $- t_1$, the time spent on traveling from the client side node to the server side node $t_2$ and until the request is taken up for processing, the time
spent in the server side interceptor \((t_3)\), when a request arrives at the server and when it returns to the client as reply\((t_4)\). Figure 5.1(b) shows explicitly the sources of overhead in a replicated setting. For both platforms the server-side time includes time spent waiting for all earlier arrived update methods to finish execution on the server. In the FT-CORBA platform, the time for logging the method call
information is included, as well as the time spent recording reply information. The approximate computation time is counted from the moment of leaving the interceptor at receiving the request, until it is entered again when sending the reply to the client. In the FA-CORBA platform the situation is different: the time spent in the server interceptor includes the method execution time, as well as the consensus execution time and state reading and updating times. The time taken by the reply to travel back from the server to the client node is traced in the same way in both platforms ($t_5$).

5.2 Results

This section presents experimental results for both the FT-CORBA and the FA-CORBA platforms. First, overheads are presented, followed by the time taken for failover. Note that the numerical values provided are not meant as absolute profiling information. They merely provide high-level views on orders of magnitude of timing values, as well as comparative characterizations of different replication styles within the same platform or in different platforms.

5.2.1 Overheads

Table 5.1 summarizes overheads in the FT-CORBA platform (columns three to five) and the FA-CORBA platform (column six). Each row of the table corresponds to one of the six method calls. Column two contains average roundtrip times for the different method calls when the client was calling them on the non-replicated server. The results in the next columns are presented as average percentages. Each average is computed as follows. Consider the highlighted cell in Table 5.1 that contains the value range $62\%-163\%$. Here, the average roundtrip time value for Method 3 on the non-replicated server was $65\text{ms}$. The term $62\%$ for example, indicates that the average roundtrip time in the replicated experiments was $65 + 0.62 \times 65 = 105 \text{(ms)}$. This was the smallest average roundtrip time measured within the twelve experiments. The highlighted cell shows that the average overhead when calling Method 3 on the server, while using cold passive replication style, ranged between $62$ and $163\%$ for group sizes of two, three, four, or eight replicas and checkpointing intervals $1\text{s}$, $5\text{s}$, or $10\text{s}$. In particular, $62\%$ corresponds to a group size of three and checkpointing interval of $10\text{s}$, while $163\%$ corresponds to a group size of eight and checkpointing interval of $1\text{s}$. When we further examined the range of values for each experiment set, we noted that the group size slightly
5.2 Results

Table 5.1: Summary of overhead percentages in the FT-CORBA and FA-CORBA platforms

<table>
<thead>
<tr>
<th>Method</th>
<th>Non - FT</th>
<th>Cold Passive</th>
<th>Warm Passive</th>
<th>Active</th>
<th>FA - CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method1</td>
<td>130ms</td>
<td>55%-117%</td>
<td>52%-134%</td>
<td>174%-500%</td>
<td>141%-472%</td>
</tr>
<tr>
<td>Method2</td>
<td>61ms</td>
<td>77%-221%</td>
<td>110%-285%</td>
<td>770%-3360%</td>
<td>265%-950%</td>
</tr>
<tr>
<td>Method3</td>
<td>65ms</td>
<td>62%-163%</td>
<td>74%-240%</td>
<td>270%-360%</td>
<td>213%-861%</td>
</tr>
<tr>
<td>Method4</td>
<td>80ms</td>
<td>76%-447%</td>
<td>100%-550%</td>
<td>1790%-5100%</td>
<td>308%-1020%</td>
</tr>
<tr>
<td>Method5</td>
<td>133ms</td>
<td>44%-333%</td>
<td>59%-363%</td>
<td>905%-4300%</td>
<td>223%-726%</td>
</tr>
<tr>
<td>Method6</td>
<td>106ms</td>
<td>37%-419%</td>
<td>68%-383%</td>
<td>800%-3100%</td>
<td>280%-844%</td>
</tr>
</tbody>
</table>

influenced the overheads in warm passive replication. In cold passive replication, the group size had almost no effect on the overheads. The variation in overhead for this replication style is mainly given by the different checkpointing intervals. For lower values of the checkpointing interval, the overhead was slightly higher, due to higher degree of interference between method calls and state checkpointing. Another interesting observation is that the overhead percentages are not much different between cold and warm. For active replication, on the other hand, the overheads are large, and significantly affected by group size. The large variations present in Table 5.1 are due to variation over the number of replicas. The least value in the range corresponds to a group size of two replicas, while the largest to a group size of eight replicas. The FA-CORBA overheads were also influenced by the group size, but also by the chosen pruning interval. Both parameters influence the average time taken for execution of the consensus primitive which is part of the overhead, in the following ways:

- The group size gives the number of nodes to which messages have to be sent, and from which answers must be received (their majority), as part of the consensus execution.

- The pruning interval affects the time spent in execution of consensus, due to the pruning action on the registers. Thus, the shorter the pruning interval, the more often the time spent in consensus is larger than in a no pruning situation. Therefore the average roundtrip time grows, and so does the overhead.

Figure 5.2 gives a bird’s eye view of the overhead percentages for the different methods. We have chosen to show the figures for Method 2 that are representative for other methods too. The chart represents the lower end (min %) of the overhead ranges described in Table 5.1 as well as the higher end (max %). It shows that
Figure 5.2: Bird’s eye view of minimum and maximum overhead percentages

- the overheads in the FT-CORBA platform for passive replication are lower than those in the FA platform;
- the overheads for active replication are much higher than the overheads in the FA platform;
- the overheads for cold passive replication style are slightly lower than those for warm passive.

The slices of the roundtrip times for the replicated scenarios can be further investigated to find out where the overhead is coming from. It turns out that

- in the FT-CORBA passive styles, as well as in the FA-CORBA platform, the main part of the overhead comes from the wait time when executing methods in a serialized way. This wait time is the sum of times $t_r$ spent by all requests arrived before the one currently waiting, where $t_r$ denotes the time from the arrival of the request in the server interceptor until the result of its processing is sent back to the client. In the FA-CORBA platform, this wait time, an inherent feature of the algorithm, is even larger because individual $t_r$'s are increased due to the execution of the consensus primitive (see Figure 5.3 for the values of the consensus time);

- for active replication, the average overhead values are generally large. Since the larger overheads appear only for method calls that themselves made outgoing calls, it was deduced that it was because of the gateway’s duplicate
5.2 Results

Figure 5.3: Time spent in consensus for group sizes of 2, 3, 4, 8 replicas

suppression mechanism. Method 3 did not call other methods, and thus experienced a lower overhead.

5.2.2 Failover times

As mentioned earlier, support by middleware requires writing little extra code by the application writer, so in that sense it is directly comparable to a non-replicated scenario. In this section we quantify the benefit to the application writer in terms of shorter failover times (compared to manual restarts). We summarize failover time performance in both platforms and relate to the parameters influencing it.

In the FT-CORBA platform there is a significant difference between failover times using passive or active replication. In active replication, the failover time was extremely short: 70 to 75 ms. In passive replication, when using the warm style, the faster failover was due to the fact that no state transfer was needed at the moment of the reconfiguration, as opposed to the cold style. In both passive styles, the failover time was dependent on the number of method calls logged since the last checkpoint that had to be replayed on the backup. This dependence was stronger in the warm passive case.

In the FA-CORBA platform the failover time was dependent on the number of state changes that happened in the application object since the last witness regis-
Chapter 5. Empirical studies with a telecom application

ter pruning. These state changes (updates) have to be applied at failover on the application object inside the new leader. The operations associated with applying an update on the new leader, consist of reading the update to be applied while performing the consensus primitive (in failure or no failure case), followed by applying the read update on the application object. Let us consider these operations as similar to the operations associated with replaying a method in the FT-CORBA platform. This will give us the possibility to elegantly compare failover times obtained in experiments with the two passive styles in the FT-CORBA platform and failover times obtained in experiments with the FA-CORBA platform. Using absolute failover time values for the comparisons would not yield relevant information, as the number of requests to replay, respectively updates to apply, could be different from one platform to the other.

Figure 5.4 shows how the average time per method replay (FT-CORBA), respectively update application (FA-CORBA) depends on the group size in a scenario where the server failed after approximately 600 requests, and the checkpointing (respectively pruning) interval was 5s. The general tendency is that the average replay time is the lowest in warm passive replication, followed by the cold passive replication. The highest average time per individual replay (update application) was experienced in the FA-CORBA platform. The explanations are as follows:

- the relation between the average replay time per method in cold and warm passive replication is due to the fact that the failover time in cold passive replication includes, besides the time for replaying the methods, the time to set the last checkpointed state on the new replica

- the average time for applying the stored state changes in the recovering leader (in FA-CORBA) is higher because each such operation implies execution of the consensus algorithm which, of course, is not needed in the FT-CORBA platform

The length of the checkpointing/pruning interval makes a difference in the number of requests to replay (updates to apply). In our experiments, the numbers become larger as the intervals grow. Figure 5.5 shows the general tendency of the average time to execute one update application in the FA-CORBA platform. This average time grows with the size of the replica group (due to the consensus execution). Here we show the experiments with pruning interval of 10s.
5.3 Lessons learnt

This section will present lessons learnt after our two years experience with building two platforms and performing empirical studies by running the Activity Manager on top of both of them. The points presented here are not necessarily reflected in the result graphs presented earlier. They contain an overall characterization of the platforms and the types of applications that present advantageous characteristics to be run on top of them.
Average roundtrip time for update method calls is highly influenced by the nature of the client side code. That is, the sequence and placing in time of the method calls affects the wait time caused by serialized execution of methods on the server. The wait time for an update method call is the sum of execution times for update operations that arrived just before that method call and are executing or queuing on the server. In the FA-CORBA infrastructure the wait time also includes the time spent in executing the consensus algorithm after each execution on the application.

Further, in the FT-CORBA infrastructure, by enforcing the serialized execution of (non-independent, as well as independent) state portion changes from the middleware level, a request cannot be executed on the application object until the previous one completely finished execution there. Chapter 6 will present an alternative: imposing the serialized execution from the application level where there is more knowledge about when a portion of the state is no longer accessed in a method, and thus can be changed within a subsequent method execution.

In both infrastructures, there is no support for handling application created threads, these being in the same address space as the object itself. In this context, it is possible to modify the object’s state by executing operations directly on it, and not using CORBA calls that would be possible to intercept and log. Thus, such state modifications are not reflected in the logged method calls/state changes, and thus cannot be replayed once there is a crash. Therefore, using such a design pattern is not advisable in sensitive applications.

In the FT-CORBA infrastructure, because of the separation between the failure detector and the client (both can be seen in some way as failure detectors), it is possible that the client will discover the application server crash before the “official” failure detector finds and reports it: e.g., when the monitoring interval is relatively large (30s), and meanwhile the server thread crashes. This is a problem for cold and warm passive replication. Thus, even though the client cannot reach the server (primary) possibly due to a real crash, no failover action is initiated. What happens is that the client sends the request to an alternative server (a backup). Still, the backups are not supposed to process any requests, but send a forward exception to the client. The latter will receive the still unchanged address of the server group. It will use it over and over again until it realizes that all alternative addresses were tried out. Finally it will give up (throw an exception in the application). An immediate solution to this problem would be to tune the application such that the right fault-monitoring interval is chosen. It would be possible that the client announces the failure to the replication manager, and the latter one modifies the monitoring
5.4 Reflections on the FT-CORBA infrastructure

Building and using the FT-CORBA infrastructure facilitated the evaluation of the concept of support for FT in middleware in itself. On the one hand it is attractive to use a service oriented middleware (CORBA) extension to build fault-tolerant applications. The only requirement from the application writer is to start the service objects, deploy them on a set of hosts, and set the replication policy. Building the infrastructure has shown us the feasibility of extending an existing CORBA implementation to support fault tolerance by closely following the FT-CORBA standard.

On the other hand, building a fault-tolerant infrastructure following the CORBA standard implies using more resources and introducing new single points of failure. For example, the fault monitors, though not fully specified in the standard, are implemented as separate entities (CORBA Objects) as the most straightforward solution. Also, monitoring of an object being done via a CORBA call is somehow waste of resources.

Furthermore, in the current context (following the FT-CORBA standard) there is no straightforward solution to the late failure detection problem mentioned above. The backups could be designed so that they can notify the failure detector of the primary that something seems to be wrong - because the request arrived at the backup. The request has to be blocked until the failure detector can tell whether it was just a false alarm or a possible failure. In the latter case, the failure detector will announce the failure, and failover action will be taken. To implement unreliable failure detectors is not straightforward either. The reason is that the failover is handled by a separate unit (the replication manager) rather than the server group itself, unlike the FA-CORBA infrastructure. As soon as the failure detector senses something looking like a failure, it reports it to the fault notifier and further to the central failover handler. The failover procedure is triggered, and therefore a drastic change in the group (e.g. promotion of a backup to the role of primary) happens, without a straightforward way of turning back. Therefore, the failure suspicion of the failure detector is made definite, implicitly assuming a perfect failure detector. A solution to this problem is to “keep alive” the suspected server (of course, if it really crashed, then the suspicion is correct). Further, the client will get its request processed by the server that it can reach (the old or the new primary). Meanwhile, if the failure detector “changed its mind” about the suspicion, it will report this to the fault notifier in the same way that it reports the failure, and a reverse process
can be started. However, this solution can lead to the existence of transient periods (even if very short) when two primaries coexist.

**Conclusions**

The most critical point is perhaps that all service building blocks of the FT-CORBA infrastructure constitute single points of failure. Thus, we can obtain transparency and modularization, but we have to deal with infrastructure vulnerabilities. In earlier works there are indications about using the infrastructure itself to replicate CORBA objects that are part of the infrastructure [111]. However, it is not clear how the approach deals with the vulnerability. Extra resources are needed but single points of failure are still not removed. In case of infrastructure components it might be easier to use more efficient replication strategies that do not provide transparency, a property that is important for application object servers, but not for an infrastructure.

The FA-CORBA infrastructure sorts out most of the mentioned problems. However, it requires a majority of the replicas of the group to be always up. Thus, if the number of replicas that the application can afford to use is fixed (n) then if $\lceil \frac{n}{2} \rceil$ replicas fail, the algorithm cannot function, and the server cannot answer anymore requests. In the FT-CORBA infrastructure, in a similar setup, even if $n - 1$ replicas fail the server will still function.

The results shown in Figure 5.2 plead in favour of using passive replication in the FT-CORBA infrastructure. The failover time comparison shown in Figure 5.4 reassures that the FT-CORBA passive replication styles are to be chosen for high availability of the application. However, active replication does not recommend itself, due to the extra resource requirements and the high overhead during no-failure periods. If we consider the FA-CORBA infrastructure, the resource demand is also quite high, as a majority of the group members have to be up in order to be able to process client requests at all.
Chapter 6

Improving performance of fault-tolerant software

This chapter presents our approach for using aspect-oriented programming to extend application code in order to accommodate a middleware level mechanism. More precisely, the call logging and serialization enforcement mechanism is moved from middleware level to be executed at application level.

The chapter includes two results:

- the implementation of the approach by extending the FT-CORBA infrastructure that we described in Chapter 4 and by devising appropriate aspects

- the performance results in terms of roundtrip time overhead measured using the same application as in Chapter 5 to run on top of the modified infrastructure. We compare the overheads obtained when using the original FT-CORBA infrastructure and when using the AOP extended infrastructure.

We will begin the chapter by presenting the motivation for our work, followed by explanation of basic terms used in this relatively new area of software engineering. The last section of the chapter will be dedicated to presenting other research on involving aspect-oriented programming as a means to offer flexibility to middleware.
6.1 Motivation

As mentioned in Section 5.3 in the context of the FT-CORBA infrastructure, enforcing sequential execution of methods from middleware level, impairs average roundtrip time of requests. An alternative is to lift middleware level mechanisms, to be performed partly at application level. The advantage is that at application level more information about the semantics of methods is present and can be used. Although the application code has to be changed for implementing the approach, this can be done with very little application writer intervention by using Aspect-Oriented Programming. We provide our implementation and measurements by extending the FT-CORBA infrastructure that we built. However, we claim that the methodology described can be used for other middlewares with fault tolerance handling capabilities such as logging, and for other programming languages than Java.

We use aspects to introduce the call information logging clauses from middleware level in the method code itself (i.e. at application level). As soon as a method is executed on the application, as a result of a client request processing, the logging will be performed as well. Method executions must still inflict their changes on the non-independent state portions in the same order that the corresponding call information records were logged. The difference now is that, if independent portions of state must be changed by two executing methods for which call information was logged in a certain order, those state changes can be performed in any order, after all non-independent state portions were changed in the respective order given by the logging. In a Java language terminology, in this variant, synchronization of thread execution is not done at whole method execution level, but at state variable access level. Let us take an example Java class ExampleClass containing methods method_a, and method_b as in Figure 6.1. Consider the scenario where method_a’s call information record is logged before method_b’s. When logging is performed at a platform (interceptor) level, the start of execution of method method_b on the application object is delayed until execution of method_a is ended. On the other hand, with application level logging and field level synchronization, the execution start of method_b is delayed only by the execution time of the first line in method_a (the assignment \( v1=v2/3 \)). This is due to the fact that \( v1 \) is the only non-independent state portion involved in both method executions. After (but only after!), the ordered changes to \( v1 \), the other (independent) state portions (\( v3, v4, v5, v6 \) and \( v7 \)) can be changed in any order due to execution of method_a and method_b.
6.2 Basic AOP concepts

The term AOP (Aspect-Oriented Programming) was first introduced by Kiczales et al. [80] in 1997. This new programming paradigm was needed to complement the well-known and praised object-oriented programming, due to OOP failing to aid software engineering where non-functional features have to be represented in the design. Besides, non-functional properties of a system crosscut functional blocks and this makes them difficult to represent as separate modules. Crosscutting examples, and how easily they can be implemented in an aspect description languages are given by Lopes et al. [98].

AOP involves defining so-called aspects to be weaved in the objects (components) they cross-cut. In our work aspect is used in the sense of code that is inserted at a well-defined point in the execution of the code for the functional part of a system, and thus changes the code. In order for the code of an object (or component) to incorporate the non-functional features described by aspects, a process called aspect weaving must take place. This involves the code of the component/object, an aspect weaver and the aspect code. The result is a component/object enhanced with non-functional features. Figure 6.2 illustrates the concept of aspect weaving.

In the following we will briefly describe AOP related notions we used in our work. These are defined conforming to their implementation in AspectJ, the aspect language we chose to use for defining our aspects. The concepts used in AspectJ

```java
void method_a(String[] val){
    StructureC loc;
    v1=v2/3;
    v2=17;
    v3=v4 || v3;
    loc=v5.field2;
    loc.set_a(3);
    v5.meth_(v6,4);
    for(int i=0;i<val.length;i++)
        System.out.println(val[i].the_v);
}

void method_b(){
    v1=35-v1;
    if (v7>25)
        v7=v7-6;
}
```

Figure 6.1: Two method calls that access non-independent parts of the state

To further understand why using aspects is a good way for modifying application code with no application developer intervention, we present the related concepts below.
are very similar to those used in the other languages, but they are written according to different syntax.

AspectJ is an aspect description language that is used in relation with the Java language [31]. It provides specific constructs to define pointcuts and advices. An aspect in AspectJ is a first class entity, meaning that at runtime, an instance of an “aspect class” is created whenever that is needed, depending on the description of the aspect class. Advices are the parts of the aspect code that are actually executed when specified join points are reached. A join point is a well-defined point in the execution of a program. For example, join points defined by AspectJ are, among others, start of execution of a certain method, call to a certain method, read/write (get/set) access to a field of a class. In AspectJ, the code of an aspect consists of pointcut and advice definitions.

A pointcut is a program element that picks out join points, as well as data from the execution context of the join points. Pointcuts are used primarily by advices. The advice code can be executed before, after or around (instead) of the application code at the join point. These three keywords are used when defining the respective advices inside the aspect. The example in Figure 6.3 illustrates a “before” advice. Thus, if the aspect code is weaved in the application code, the new application code

---

1 AspectJ is a trademark of Xerox Corporation.
2 When used in connection with objects or classes, method means operation of a class
6.3 Aspects for fault tolerance mechanisms

In our approach, the application writer will be provided with aspects incorporating clauses for call information logging, as well as variable level synchronization clauses. The aspect code will be automatically generated from a simple gray-box description of the server application object, provided by the application writer. By gray-box we mean that no application code has to be provided (white-box description), but it is not enough to only provide a list of methods with their parameters to be called on the object from outside (black-box description). Aspects can easily

object will print out the given text whenever it starts executing method_a. The italic words are keywords in AspectJ. The code is written using the AspectJ syntax for defining aspects, pointcuts and advices.

Besides enriching existing methods of a class (component) by adding pieces of code that execute at join points, it is possible to enrich the component itself by adding new methods and fields. This way of enriching is called *introduction* in AspectJ. For example, our aspect (ExampleAspect) can be extended to introduce a new method method_c in the class ExampleClass (see Figure 6.4). It is possible to define pointcuts (and thus, advices) related to the new method, as well.

Figure 6.3: Advices, join points, pointcuts

```java
public aspect ExampleAspect {
    pointcut execmethod_aPointcut(Str[] val):
        execution(method_a(Str[]));

    before():execmethod_aPointcut(Str[])
        System.out.println("before execution of method_a");
}
```

6.3 Aspects for fault tolerance mechanisms

In our approach, the application writer will be provided with aspects incorporating clauses for call information logging, as well as variable level synchronization clauses. The aspect code will be automatically generated from a simple gray-box description of the server application object, provided by the application writer. By gray-box we mean that no application code has to be provided (white-box description), but it is not enough to only provide a list of methods with their parameters to be called on the object from outside (black-box description). Aspects can easily
be weaved in the application code, to provide better performance of the application on top of e.g. an FT-CORBA middleware that uses a separate call information logging mechanism, and a different synchronization mechanism. To illustrate this approach, we use our implemented FT-CORBA platform (Chapter 4), and performed some modifications to support aspect-oriented application extensions. The modifications to the middleware were mainly within the interceptors that now skip the call information logging execution leaving this to be completed at the application level. The application code is automatically modified using corresponding aspects. The exact changes in the FT-CORBA infrastructure that are concerned with the logging mechanism (in order to perform call information logging at the application level) are as follows:

1. Interceptors used in primary-backup replication, were adapted to throw an exception containing the call information that has to be logged from the application level. Throwing an exception is needed, because the call to a method (in the form it comes from the client) has to be stopped from reaching the application object. After the exception is thrown from the interceptor, other middleware levels are informed to take the necessary steps so that the application receives the call to the right (pseudo) method (the one that
incorporates logging clauses and whose signature has an extended set of parameters. These changes solely affect the server side interceptor in the FT-CORBA infrastructure. Thus, an application may choose to use the standard (FT-CORBA) mechanisms (i.e. those that do not throw the mentioned exception), or the version supporting aspects. This is simply done by using different interceptors when deploying the enhanced infrastructure. Applications with different performance requirements can thus exist side-by-side using different interceptors.

2. The original logging mechanism’s interface was extended with a method for storing call information, without waiting for previous calls to finish execution. Of course, if the application writer wants to use the regular FT-CORBA platform, he/she can choose to use the new logging object, and calling the (old) logging method that includes waiting, or the old implementation of the logging object.

In figures 6.5 and 6.6 we present the FT-CORBA infrastructure providing support for application writers to deploy fault-tolerant applications. Figure 6.5 illustrates the FT-CORBA platform we described in Chapter 4, where all infrastructure operations such as logging are performed at middleware level. Figure 6.6 illustrates the extension of our FT-CORBA infrastructure with the addition of support for aspects, and logging performed from application level.

As described earlier, an application writer that will use the support from a regular FT-CORBA infrastructure has to add a couple of small extensions to the
application: methods for getting and setting the object state \( \text{get}\_\text{state} \) and \( \text{set}\_\text{state} \). Logging of call records in the right order is entirely taken care of by the infrastructure. Similar additions to the application code are also necessary when the application writer chooses to use our FT-CORBA infrastructure with support for aspect orientation. Besides, to aid the automatic generation of the aspect code, the application writer has to provide a gray-box description of the server application object. This description complements the, e.g. CORBA IDL description of the methods offered by the application server object.

Synchronization and call information logging aspects will be weaved only in the code of update methods. Synchronization at variable level involves the fields of the object that are accessed inside an update method. With respect to taking and releasing the lock corresponding to a variable (in the context of variable access level synchronization), the first and last access to that variable inside the method is important. To simplify the implementation, we make no distinction between read/get and write/set accesses to a variable. Also, the accesses are counted in terms of number of appearances of that variable in an instruction (for example: in the assignment \( a=a/2+a \), the number of accesses to \( a \) is three, although to compute the right hand side expression \( a \) has to be read only once). When working with a CORBA based platform, the following elements have to be present in the gray-box description for a server object:

- A table (\text{TypeConv}) of Java type to IDL type mappings; the table is used when call information is logged; thus, the mappings have to be given only
for types of parameters or results of the update methods;

- A list (Interface) with update method descriptions, in terms of signatures, with parameter types and names;

- A list (State) with the types and names of object fields (state portions), that are accessed in update methods, as well as the types for field portions that are accessed via local variables in those methods;

- A table (FieldAccess) of method names and corresponding accessed fields (with names), or types of local variables, followed by number of accesses during the execution of the corresponding method.

Note that all of this information can, in principle, be automatically deduced by static analysis of the program. However, for the purpose of this study, we assume these as given. Further, although we presented the elements of a gray-box description in a CORBA based middleware case, we claim that similar descriptions can be produced easily for other middlewares used to which aspect support is added.

For our class ExampleClass the description is presented in Figure 6.7; the method code in which aspects have to be weaved is that of method a and method b. Method method a, when executing, accesses the fields v1, v2, v3, v4, v5, v6, v7, and a local variable of type StructureC, which when being modified changes the state of the object (via field v5). The description is written according to syntax that we impose: the italic words in Figure 6.7 are keywords; also the use of “:” in the FieldAccess section is our choice.

### 6.4 Implementation issues

The FT-CORBA infrastructure requires application writers to indicate which replication style they wish to use in their application (primary-backup with cold or warm passive replication, or active replication). In this section we explain how the support for the two primary-backup mechanisms is implemented within the aspect-orientation framework. Support for active replication can also be provided with little effort. However, queuing time not being the largest part of the overhead in that case, no big performance improvement is expected. For warm/cold passive replication, call information that is logged consists of the name of the method (e.g. method a), the list of call parameters encapsulated in a list of middleware spe-
specific types\(^3\), and the unique request identification information (retention identifier, client identifier). When logging is done at the infrastructure level, the latter piece of information is easy to obtain, as it is sent in a service context carried by the request. On the other hand, at the application level, this information is not available unless the method is called with these extra parameters as well: at application level, unless something specific is done about it, there is no way to access request related information such as service contexts. Our approach is to generate one separate aspect code file for each method described in the list Interface. Each aspect so generated, combines the introduction of a new pseudo method in the application code, with the definition of advices that contain code for call information logging, and the definition of advices for variable level synchronization. The advices will be executed as the specified join points are reached. The new (pseudo) method is introduced in the class, in order to be able to extend the parameter list of the original method, without the application writer being obliged to write a new method with the extended signature. The body of the pseudo method is simply a call to the original method with the original set of parameters. To use the extra parameters, around advices will be used.

\(^3\)In our example the argument \textit{val} of type \texttt{Str[]} is encapsulated in the single element of an array of elements of the CORBA specific type \texttt{Parameter}.
6.4 Implementation issues

6.4.1 Aspects defined percflow

An explanation of the AspectJ *percflow* notion is needed here. If aspect `Asp METHOD A` is defined with the attribute `percflow(Pointcut p)`, where `Pointcut p` is defined inside the body of the aspect (exec: `method p` in Figure 6.8) then one object of type `Asp METHOD A` is created each time the flow of control reaches a join point from the set defined by the pointcut `exec: method p`. The difference as compared with a non-percflow aspect `Asp METHOD A` is as follows: the variables defined inside the body of the aspect (is_first and counter in Figure 6.8), do not become instance variables of the aspect-woven class (here, `ExampleClass`), initialized at the aspect instance creation, but they are defined as local variables within the scope of `exec: method p`, each time an instance of `Asp METHOD A` is created, i.e. each time `method a` executes; also, the code of advices (before and after in Figure 6.8) described in the body of the aspect, is executed at join points selected by pointcuts defined inside `Asp METHOD A` (get_field in Figure 6.8), but only those reached while in the scope of `exec: method p`.

6.4.2 Method execution related advices

To illustrate how an aspect file looks like, let us take the example of `method a` from our Java class `ExampleClass`. There will be one pseudo method introduced in the class, that is `pseudo method a`. The parameters of `pseudo method a` are the parameters of `method a`, plus the extra parameters needed, as mentioned before in case of primary-backup replication. The join point for which the method related advice is defined is the execution of the pseudo method. The aspect code is defined to activate its advices at the given join points, whenever the method `pseudo method a` is executed, i.e. the control flow reaches any join point defined by pointcut `exec pseudo method a`. To achieve this we need an aspect defined with the attribute `percflow`, which is written as follows:

```java
public aspect Asp METHOD A percflow (exec: method a p(Str[], int, String)) {
    ... 
}
```

The reason for defining all aspects to be of type `percflow` is the need to introduce some extra local counter variables in the methods, used as explained below. The advice code that will be executed at the method execution join point is of type `around`. This means that the code written inside the advice is run instead of the code of the method. The new execution starts with logging the method call information...
public aspect AspectForMethod_A percflow(exec_method_a_p(Str[])){
    int counter;
    boolean is_first;

    public AspectForMethod_A(){
        counter=0;
        is_first=true;
    }

    pointcut exec_method_a_p(Str[] val):execution(method_a(Str[]))
    && args(values);

    pointcut get_field():get(int v1);

    before(): get_field(){
        if(is_first){
            ...
        }
    }

    after(): get_field(){
        counter++;
    }
}

Figure 6.8: Example of an aspect defined percflow

obtained from the parameters of the call. Further, the numbers of threads currently
accessing fields that will be read/written by this method (thread) are assigned to
the extra local counter variables. The global variables containing the numbers are
incremented. These global variables are static fields in an aspect class (not detailed
here) specially defined to contain them. Finally, the (pseudo) method itself is called
(by using the AspectJ keyword proceed). A fragment of the aspect code used for
the extension of method_a is shown in Figure 6.9.

The reason for the somehow strange combination of introducing a new method
in the application class and then replacing it with an around advice is that the local
variables defined inside the percflow aspect cannot be accessed in the pseudo
method itself as method (only introduced) in the application class. Of course, the
6.4 Implementation issues

pointcut associated with execution of pseudo_method_a

\[\text{pointcut exec pseudo_method_a_p(Str[]} val,\text{int} r\text{.id,}\text{String} c\text{l_id}):\]
\[\text{execution(pseudo_method_a(Str[]),int,String))&&args(val,r\text{.id},c\text{l_id});}\]

\[\text{void around(Str[]\text{,int,String): exec pseudo_method_a_p(val,r\text{.id},c\text{l_id})}\{\]
\[\text{...}\]
\[\text{logging_obj.log_method_call(val,r\text{.id},c\text{l_id});}\]
\[\text{...}\]
\[\text{proceed(val,r\text{.id},c\text{l_id});}\]
\[\text{around advice}\]

\[\text{call pseudo_method_a}\]

Figure 6.9: An “around” advice

server side ORB has to know to call the pseudo method instead of the original one, since the client will transparently call the latter. The code of the skeleton generated at the server side is augmented with calls to pseudo methods, according to the component description provided by the application writer. At runtime, the skeleton object is directly provided with the name of the pseudo method to be called: in the point immediately over the server interceptor, the method name is simply changed from the original (as called by the client) to the pseudo method’s name. The server interceptor is responsible for throwing the exception that will inform other ORB levels about the name change, and of course about the extra parameters\(^4\).

Note that e.g. method\(_a\) still exists in the class ExampleClass. However, it will not be called when the aspect-supporting infrastructure is used. Instead, pseudo_method\(_a\) is called with the functionality of method\(_a\) preserved inside it.

6.4.3 Field level synchronization advices

Other join points used to define advices are the get/set accesses to object fields and local variables that can modify the state. To select get/set accesses to e.g. a field called \(v1\) of type \text{int}, the following pointcut is written in AspectJ:

\(^4\)The IDL compiler was changed to cope with the addition of pseudo method calls to the skeleton’s code
pointcut get_set_vlp(): get (int vl) || set (int vl);

The variable level synchronization takes place as follows. Before the first access to a field in a method, the current thread gains the semaphore corresponding to that variable, after all previous accesses by other methods (threads) are finished. It is known when this happens, because the local counter variables are decreased whenever the semaphore corresponding to that field is signalled. The semaphore is signalled in a method, after the last access to the field in that method. The first access to a variable in a method is detected in the advice code that is always executed before the variable access. To detect the final access, counting of all get/sets is done and the number compared with the value specified in the component description file. The semaphore variables are also static fields in the same aspect class that contains the global counter variables.

### 6.5 Evaluation of the approach

We performed experiments in a replicated setting by using the aspect-supporting FT-CORBA platform built as described in the above sections. To obtain relevant results, the same experiments with the O&M service (Activity Manager) as in Chapter 5 were conducted. The update operations performed on the activity manager server object by client requests modify its state by mainly creating jobs and activities, starting and terminating those, as well as reporting status changes in activity executions. The client that calls methods on the server was also the same as in previous experiments. Thus, the client called six methods of the server out of the fifteen available, as part of the experiments. The O&M service was augmented by the aspects generated based on the gray-box description.

Our goal was to compare average roundtrip time overheads obtained in this setting with the average overheads measured using the baseline FT-CORBA platform. Overhead values were computed by subtracting the roundtrip time obtained in a non-replicated scenario from the time taken in the replicated one. Six update methods (called m_1, m_2, etc. in our result tables, and referred also as method 1, 2, etc.) were used in the tests and average overhead percentages were computed for each. We performed our experiments in the same hardware setup as the experiments with the FT-CORBA platform: a set of SUN Ultra SPARC workstations running SunOS 5.8, connected in a LAN. Also, as previously, we did not have control over the link traffic or the load on the machines.

The parameters we varied during the experiments were: the replication style (warm or cold), the number of replicas, the checkpointing interval, and the number
of times the client called the methods on the server, i.e. the number of iterations of the loop which the client used. This was the case for both infrastructures: FT-CORBA (from Chapter 4) and aspect supporting FT-CORBA (this chapter). The measurements obtained after each varied parameter will be referred to as one experiment. Thus, in case of cold passive replication, for each server method called by the client, and for each infrastructure, we obtained 12 numbers 5 designating average roundtrip time overheads, corresponding to 12 combinations of parameters:

- number of replicas: 2 (1 value)
- checkpointing intervals: 1, 5, and 10 seconds respectively (3 values)
- number of iterations: 100, 200, 400 and 800 respectively (4 values)

In case of warm passive replication, for each method called by the client, and for each infrastructure we obtained 36 numbers 6 designating average roundtrip time overheads, corresponding to 36 combinations of parameters:

- number of replicas: 2, 3 , and 8 respectively (3 values)
- checkpointing intervals: 1, 5, and 10 seconds respectively (3 values)
- number of iterations: 100, 200, 400 and 800 respectively (4 values)

Choosing only one value for the number of replicas parameter, in cold passive replication, was due to the fact that the parameter is expected not to influence at all the overheads. For warm passive replication, the overhead does not change much either with the number of replicas, but more than for cold passive. The two charts (Figure 6.10 and Figure 6.11) present the average roundtrip overhead computed over the above averages obtained in each experiment. That is, for each of methods 1 to 6, and each platform, a comparison is presented between:

- cold passive: the average of the 12 numbers obtained, and
- warm passive: the average of the 36 numbers obtained

The charts show a general drop in the presented average values. This is especially true for cold passive replication where the drop is sometimes over 50%. Due to the fact that the call to \( m_1 \) does not interfere with so many other method calls, it should not be a surprise that for method 1 the results do not show an improvement. In fact, the synchronization clauses in the application code can themselves be causing an overhead. In addition, there is a further explanation for the phenomenon:
when a method call arrives in the server’s wait queue, it is possible that a call to get_state is there somewhere in the front. Now, get_state cannot start executing until all previous methods in the wait queue finish their execution; this is valid in both platforms (the FT-CORBA and the aspect-supporting FT-CORBA). Therefore, the method call that arrives after the call to get_state (e.g. m_1), but before get_state finished execution, has to wait until all methods arrived before get_state (those for which get_state is itself waiting) finish execution. In such a situation our variable level synchronization does not improve performance.

6.6 Discussion

In this chapter we presented the results of the successful merging of two areas: middleware supporting fault tolerance with aspect oriented programming and weaving in the application. Using aspects is successful, since with almost no application writer intervention, application code was modified such that performance of server request processing was improved. In some cases, as the result charts show, the overhead percentage dropped with more than 50% as compared with a straightforward FT-CORBA implementation.

As expected, there are also some drawbacks of the approach. By using a smaller granularity on method synchronization, average performance of method executions in a multithreaded environment is improved. However, this is valid when field
6.6 Discussion

Figure 6.11: Average over average overhead percentages for warm passive replication

```java
void method_a(Str[] val){
    StructureC loc;
    v1=v2/3;
    v2=17;
    v3=v4 or v3;
    loc=v5.field2;
    loc.set_a(3);
    v5.meth_(v6,4);
    for(int i=0; i<val.length; i++)
        System.out.println(val[i].the_v);
    v1=v2/3;
}
```

```java
void method_b(){
    v1=35-v1;
    if v7>25
        v7=v7-6;
}
```

Figure 6.12: Changed method_a

accesses inside methods are organized such that the extra synchronization code execution time can be neglected. Also, there are cases where even by synchronizing at variable level a method cannot start executing before a previous one completely finished. Imagine the case where method_a in our example (the class ExampleClass) is changed so that the line $v1=v2/3$ is moved from the beginning to the end of the method body (Figure 6.12). In a scenario in which the call to method_a is logged first, method_b cannot start execution, until method_a releases the lock on $v1$, i.e. finishes its execution.

An important issue is how to find out where the code for releasing the lock corresponding to a state portion has to be inserted, when this state portion can be
void method_a(String[] val){
    StructureC loc;
    ...
    loc=v5.field2;
    loc.set_a(3);
    ...
}

Figure 6.13: Field modification impossible to capture by pointcuts

accessed inside an if statement or a loop. The last access of a variable can easily be detected when the method code does not contain if or loop statements where the variable is possibly accessed an unknown number of times. Thus, the information provided by the application writer in the gray-box description already mentioned can only be used for these simpler cases. Further, even if the application writer could specify that a variable is possibly accessed in an if or loop statement, AspectJ or other aspect languages are not equipped with the ability to detect when an if or loop statement is entered and exited. In our application we do not have variable accesses inside if and loop statements, and therefore we did not face this problem when we built our experiments.

As mentioned earlier, an update method can modify a local variable and this can lead to changing the state of the object (see in Figure 6.13 the fragment of method_a where v5 is modified via loc). However, it is not possible to synchronize the access to the local variable with the access to another local variable in another method, referring to the same portion of the state. Consider the following scenario: method method_a accesses v5.field via loc; we can imagine method_c in the class ExampleClass accessing the same element also for writing, via a local variable loc defined in method_c. When loc in method_a is accessed, there is no way to know that loc in method_c is referring to the same portion of the state, or the other way around. In our approach, the problem is solved by using a semaphore associated to the type StructureC, besides those associated to class fields (this is the reason why mention StructureC in the gray-box description in the State and FieldAccess lists). Thus, a semaphore s_StructureC is associated with type StructureC, and each time when a variable of type StructureC is accessed first time, the lock is taken exactly like for object field accesses. Therefore, sometimes, unnecessary waiting will occur, if the two variables of type StructureC do not access the same memory location.

If we think about code maintainability (in the software engineering sense), as
soon as the gray-box description changes, new aspects have to be generated and weaved in the code. It is possible that the server class was only extended with new methods, and maybe state variables. In this case, the new aspects can be weaved directly in the already “aspected” code. The worst-case scenario is when the non-aspected code has to be modified by weaving in totally new aspects.

Of course, the improvement is most significant when considering the worst-case scenario, i.e. no commutativity assumption can be made about the execution of update methods. This is a reasonable assumption, especially when the middleware builder and the application writer do not have the chance to communicate. Moreover, in general, the middleware has to be built general enough to be able to accommodate many different applications.

To our knowledge this is the first work that uses aspects for improving performance of a fault-tolerant application on top of a standard middleware. To reinforce this, the next section will present a survey of existing works using AOP to provide flexibility in middlewares.

### 6.7 AOP and middleware

Aspect-oriented programming was used from early years to incorporate adaptivity and quality of service features in middleware. Loyall et al. [99] (early years) and Duzan et al. [44] (recently) show how the concept of CORBA IDL was extended with a set of Quality Description Languages (QDL) to represent quality of service requirements from clients. This is done in the QuO (Quality Objects) framework where the QoS (Quality of Service) specification, monitoring and adaptation are described as aspects. The client can use CDL (Contract Description Language) to describe quality of service contracts. ASL (Aspect Specification Language) is used to describe code for monitoring and adaptation of QoS.

To be able to define aspects, researchers strove for developing languages for describing aspects. Aspect description languages are devised as separate from high-level programming languages (for example Java, C++, Csharp - C#). However, a good feature is that they keep many constructions from those languages. AspectC++ was developed based on C++ in the works of Gal et al. [59, 145, 61, 60, 58]. C# in the .NET framework is somewhat different. In the .NET framework, aspects, including fault tolerance can be defined as C# attributes [141]. In the CAMEO project, they are defined in similar way as in AspectJ combined with

---

3Interface Description Language
support from the .NET framework compilers and C# attributes [127].

Aspects can be used to offer flexibility in the code that is executed at runtime by a system. Hunleth et al. [75] show how aspects and AOP are used to build middleware code that includes certain features chosen based on user requirements and added to a base configuration. The point is that these features are not necessarily known when the base implementation of the middleware is made. Dynamic AOP is present in the works of Popovici et al. [124, 123]. The idea is to use runtime hooks in the Java Virtual Machine to weave and unweave aspects in the running code. Although the framework relies on the usage of the Java language, it offers interesting directions of using aspects other than enriching code at compile time with no possibility of “getting rid” of woven code. The danger with using dynamic AOP is security violation [117]. Ways of preventing identity forging or unwanted behaviour changes are code signing and contracts, as seen by Palmer. A sort of dynamic aspect weaving and usage, mainly at runtime, is presented by Zinky and Shapiro [171]. Their approach is to define adaptive features as classes and instantiate different pieces of code at runtime depending on the adaptation needed. The concept is called aspect oriented interceptor pattern, and is not related to a certain programming language. Using the concept of “fragmented objects” Hauck et al. [72] develop a middleware AspectIX to permit configuration of aspects for interaction of a client with such an object. Fragmented object is a CORBA object that is transparently distributed over several sites and a client using it has at least one of the fragments in its address space.

Aspect orientation can be useful to handle the ever-increasing complexity of middlewares. Zhang and Jacobsen [166] propose a way of improving middleware architecture by horizontal decomposition to separate aspect modules. For this an aspect mining procedure is needed. CORBA is used as a case study. Coyler et al. [34, 33] presents aspect orientation as a way to perform the needed steps for designing manageable middleware systems: devising application-middleware independence, middleware simplicity and componentization by clearly separating crosscutting concerns from functional modules. As case study, the middleware product-line of IBM involving thousands of classes was used.

In our work we use aspect-oriented programming as a technique to lift middleware level fault tolerance mechanisms to application level. As a result, performance of the application running on top of the FT middleware can be improved, with keeping transparency and no-intervention from the application writer.

Works available in the area of fault tolerance as an aspect are mostly proposing ways of incorporating fault tolerance as an aspect in the design. Herrero et al.
[73] develop an aspect language called JReplica which is destined to describe at a very high level e.g. replication policies, actions to be performed before and after a replicated server is created, as well as error actions. The infrastructure where a JReplica description is used is residing on a communication middleware (CORBA, Java RMI) and has an interception layer. This layer acts on the request and performs actions described in the JReplica file. As an extension, the introduction of an extra stereotype in UML is proposed. The new stereotype called “Replication” would be helpful to express already in the design, the usage of fault tolerance features.

France and Georg [56] introduce the concept of aspect-oriented design (AOD). In this approach two types of models are developed during the design: a model of functionality and a number of aspect models that include each modules related to a certain concern, e.g. fault tolerance. The advantages of separation are clear when thinking about handling changes in the fault tolerance aspect. After a change the new model is weaved with the functionality model. In this approach aspects are defined application-independent in the sense that no exposure of the application is needed to define pointcuts and advices. The role model is used to define properties of aspects. Weaving the aspect model into the functional model actually means that the functional model is transformed to fulfill the role whose properties are carried in the aspect.

Fabry [48] is concerned with describing replication as an aspect. In an object-oriented context it is important what part of the object is replicated: its data (state) or behaviour. There are two other concerns that have to be considered and their separation as aspects is not always possible: naming of replicas and assignment of initial values to replicas. Both these operations are best done by the application that knows what initial values have to be assigned to the objects (i.e. provides and calls the constructor), and knows best for what are the objects used and thus what naming convention is suitable. However, in this situation, the replication aspect is not completely separable from the application.

Another non-functional property of a system is real-time performance. Tesanovic et al. [151] show how aspects are used to develop component-based real-time systems. The concept they present comprises a design method that assumes decomposition of real-time systems into components and aspects. Also, in a further contribution [150] the authors present a formalization framework for verifying properties of components, aspects and components changed with aspects. This work provides the grounds for enhancing middleware to support real-time properties in a predictable manner.
6.8 Summary

This chapter presented an elegant solution to the problem of moving some middleware level operation in the application code. A non-elegant solution would have been to leave only one option to the application writer: to make the application fault-tolerant with lower average overhead, the logging and synchronization mechanism should be coded into the application. Thus, our approach increases the potentials for middleware supported fault tolerance, at an improved performance level.

One could ask: why introducing the usage of AOP only in connection with the FT-CORBA platform? The answer is that in the FA-CORBA platform, there would be no performance gain to move middleware level operations to application level. This is due to the fact that actually the application processing part is itself embedded in the middleware, by ways of using the portable interceptor in which the algorithm operations are implemented.
Chapter 7

Computing the optimal checkpointing interval

This chapter will use mathematical modelling to obtain an optimal checkpointing interval for a passive replication mechanism. We focus on two optimization criteria:

- The checkpointing interval that leads to highest average availability of a server system
- The checkpointing interval that provides a minimal average response time for queries to the server

Checkpointing, although indispensable in failure prone systems to ensure availability, involves a trade-off. If checkpointing is performed too often, then the server is often stopped from processing requests. On the other hand, failover times can increase if the log where operations (for replay) are stored is not pruned often enough.

This chapter presents two mathematical models: first, a relatively simple model to maximize the average availability; next, an extended version of this model that includes the extra time spent to process the requests that queue up during failover, the so-called backlog processing time. This extra time has no significant influence on the maximal average availability, but it is crucial in the study of the average response time. In both models queueing theory is employed to model the client requests waiting to be processed by the server. The first model is used for computing the checkpointing interval that maximizes availability and the second for computing the checkpointing interval that minimizes response time.

This chapter has three main parts:
Chapter 7. Computing the optimal checkpointing interval

1. background information about the checkpointing procedure as it is performed in a setting resembling the FT-CORBA infrastructure

2. basic assumptions and a preliminary model (not including backlog processing)

3. the formulas used to derive the optimal checkpointing interval both for maximizing average availability and for minimizing response time, respectively.

The expression of the checkpointing interval will be determined as a function of other parameters of the system such as average service rates, average request arrival rate, and average failure rate. The idea is to support the engineer with a tool that gets as input the above parameters and provides the best checkpointing interval. Depending on the optimization criterion, the proper formulas are used.

To our knowledge this is the first time a mathematical model of a logging and failover mechanism is done at this level of detail and for the purpose of optimally defining the checkpointing interval. Moreover, it is the first time the study of backlog processing is introduced.

In the coming section we will briefly survey the existing works on checkpointing interval optimization. At the same time, we will position our work relative to previous results, thereby reinforcing the above claim.

7.1 Related work

Several researchers have studied the checkpointing trade-off in the context of fault-tolerant processing systems, and formulated the problem of the “optimum checkpointing interval”. Most of the results are obtained for settings where a long-running job (performing heavy calculations), or a process, is from time-to-time checkpointed [164, 87, 88, 29, 121, 95]. Other models include request processing or message logging systems [63, 148, 146] and request processing in mobile environments [27]. A recent survey by Elnozahy et al. [47] presents definitions, different aspects of checkpointing and logging, as well as implementation issues of transparent rollback-recovery techniques. However, the survey does not cover the aspect of checkpoint interval optimization.

Early results by Young [164] were based on simple assumptions such as fixed time between checkpoints, as well as the time taken for recording the state. This seminal work provided a simple, yet applicable formula for the value of the optimal checkpoint interval (depending only on failure rate and state recording time).
7.1 Related work

Criteria for checkpoint interval optimization are discussed by Krishna et al. [87]. Two types of systems are considered: real-time systems with hard deadlines and general-purpose systems. The criticism is directed towards two optimization criteria that are not powerful enough in relation to an application: availability measure and mean response time. Good criteria, on the other hand, envision aspects of the systems that really show how checkpointing is advantageous when failures occur. For real-time systems a recommended criterion is a measure of how well checkpointing helps tasks to not miss their deadlines under failures. For general-purpose systems a proposed criterion is a ratio between how much processes that fail gain by way of checkpointing and how much the other processes gain. In our modelling and analysis, the gain is shown when studying the average response time when no checkpointing is used in comparison with the minimum response time (see Section 8.1.1).

Coffman et al. [29] study the problem of scheduling checkpoints also in the context of fault-tolerant very long lasting computations. For different uniform failure arrival time laws, and considering the checkpoint time as constant, the aim is to maximize the time spent in useful computations during a certain time interval where checkpoints are also scheduled.

Kulkarni et al. [88] analyze the optimal checkpoint interval in a queuing system where long-running jobs queue at a server for execution. The analysis focuses on the completion time of one long-running job in presence of checkpointing and failures. In our context, incoming “jobs” are not considered as long-running, and besides, checkpointing can only delay queuing requests, and not executing ones.

Plank and Thomason [121], as well as Vaidya [153] consider the two time values called checkpoint latency (total checkpointing time), and checkpoint overhead (time taken by checkpointing from the running process) to be different from each other, but constants. Previous works consider these as equal. In our analysis both of them are random variables.

Works that model request processing server systems by queuing, usually do not analyze the call replay time part of the failover time in terms of the number of calls/messages to be replayed. The assumption of the state transfer time being a constant is kept. For example, Gelenbe [63] defines the failover time \( h(\gamma) = \alpha \gamma + \beta \), where \( \gamma \) is the checkpointing interval that has to be optimized. \( \alpha \) depends on a constant \( k \) that is the proportion of requests that has to be reexecuted after a failure. \( \beta \) is the state transfer time (constant). In that paper, it is proved that when \( \gamma \) is deterministic, then the system’s availability is maximal. In our model, instead of using a constant like \( k \), we assume an exponential distribution
for the replay time for one call, and find a distribution of the number of calls to be replayed, to further compute the failover time.

Tantawi and Ruschitzka [148] use the same type of formula for the failover time, where the two coefficients are constants. They assume general failure interarrival time distribution, as well as the possibility of failures occurring during failover. However, due to the generic assumptions they end up providing mathematically intractable computations, even if failures are not allowed to occur during failover and checkpointing. Only when considering an equidistant checkpointing strategy the problem of computing an optimum becomes tractable.

More practical works in similar contexts are conducted by Chen and Lyu [27], and Ssu et al. [146]. Chen and Lyu present a mathematical model for a checkpointing system in mobile environments. Their model is extended as compared with previous ones, because of hand-offs that can occur. They define effectiveness as a measure for how well the system performs under failure conditions, with and without checkpointing. They assume that failures can occur during recovery (repair) after a failure. The repair time is modelled as a random variable with general distribution. No explicit modelling of message logging and replay is used. Ssu et al. develop a checkpointing scheme in which the interval between checkpoints is modified during execution depending on incoming request rate as well as failure rate. Message logging is involved, to reduce failover time. A user provided upper bound on failover time is considered when deciding whether to make a checkpoint or not.

A different approach, based only on simulation, is taken by Katsaros and Lazos [79] in their work about determining the optimal state transfer policy in a replicated scenario. The study is conducted on an implementation of the FT-CORBA standard. Since in active replication state transfer to a new replica is needed only when the number of alive replicas drops under a certain value, this replication is recommended when the time to transfer the state is longer than the time to execute a request. In cold and warm passive replication state transfer and thus checkpointing are done often, and therefore it is worth to study the type of checkpointing to be employed (periodic or load based) and the interval at which it should take place. The results of the simulations show that load based protocols give rise to lower response time overheads, and lower sensitivity to size of the state, as well as increased system availability.

The area of optimizing response time by choosing an appropriate checkpointing interval has not been studied so extensively. The work of Gelenbe and Derochette [64] comprises mathematical analysis for determining the checkpointing interval
minimizing the response time of transactions, respectively maximizing availability of database systems. As a difference from our work, the time taken by a checkpoint operation is considered exponentially distributed. Also, as in the work of Gelenbe from 1979 [63], the failover time is considered simply as proportional (via a constant $k$) to the checkpointing interval, where $k$ is not computed in the mathematical model, but chosen empirically. An outcome of the work (as will be visible in our results, too) is that the optimal checkpointing intervals (rates) are rather different for the two optimization criteria. Huang and Jalote [74] present the effect of fault tolerance (in terms of replication and degree, total failures and repair time) and checkpointing on the average response time. A model is presented for computing the average response time in a primary-backup setting, where the resulting formula contains the system availability. No explicit logging of requests is considered. Furthermore, the replay is done after all requests since the last checkpoint are resent to the new primary. The replay time is simply computed as being proportional with the checkpointing interval.

In the area of real-time, the influence of checkpointing on performance has been studied mainly for hard real-time systems ([168], [129]). Both works present a formula for the worst case response time considering different failure arrival patterns and checkpointing strategies. Zhang and Chakrabarty [168] consider the rollback and state restoration time to be zero. Punnekkat et al. [129] present an optimal algorithm for checkpointing real-time tasks such that the least time is needed if a failure occurs and some task has to be reexecuted.

Even if not directly related to checkpointing optimization, queuing theory tailored for real-time systems - so called Real-Time Queuing Theory (RTQT) [93] - provides solutions for the queuing system in heavy traffic. Lehoczky [93] reduces a queue with heavy traffic to a Brownian motion. RTQT is used in the work of Zhu et al. [170] to analyze design trade-offs in a real-time system where tasks with given end-to-end deadlines are served in a network of server nodes forming a pipeline.

Our two criteria for optimization when determining the optimal checkpointing interval are similar to other researchers’ choice: the average time the server is available for processing client requests, chosen by Plank et al. [121] and Vaidya [153], and the average response time of a request, like in the work of Gelenbe [63]. Although these criteria may not be recommended for hard real-time applications [87], we consider them powerful enough for meaningful analysis in a wide range of high availability and soft real-time applications.
Chapter 7. Computing the optimal checkpointing interval

7.2 The checkpointing procedure

Chapters 7 and 8 consider the mathematical modelling of a part of our FT-CORBA infrastructure (presented in Chapter 4), namely the logging and checkpointing unit used in failover procedure (see Section 2.2.3), plus part of the application (the server object). However, we claim that the modelling and analysis performed can be applied to any other middleware, or even in contexts where middleware is not present. The goal of the optimization chapters is not to formalize the original overhead study. Rather, we aim to support the tuning of one parameter (the checkpointing interval) in order to achieve maximum availability, respectively minimum response time. Another point to note is that we analyze the primary-backup service availability as if there were sufficient number of backups available for the given failure frequency. A special case of this is the assumption of one backup and that there will be no failures until the most recently failed primary has recovered. If the number of backups is limited or the latter assumption is not valid in the circumstances then the model has to be extended to include the repair time for a server.

We now go on to describe the same checkpointing procedure that was already addressed in the Chapter 4. However, here more (and different) details are given on the operations involved therein, as these are needed to better understand the modelling that will follow. We envisage that similar operations (constituting the checkpointing procedure) are part of any such middleware or application that uses the primary-backup mechanism, and thus have a general character.

Figure 7.1 shows the general scheme of checkpointing and logging procedures. Upon arrival from the client, the update method calls are logged by a server that is dedicated for logging (Logging server on the left hand side). The main server that handles the application object is denoted by Application server. The call log includes information about the result of executing the client query on the Application server, too. This information is used by the middleware in order to avoid executing the same call more than once on the server (and is used only if a client sends the same query more than once).

Thus, for an arriving request, before logging call information, the system looks for the record of a reply to the call. If this is found, the reply is returned at once to the calling client. Thus, no operation on the application object will take place. Otherwise, the call information is recorded in the log, and the request will proceed to the Application server for being processed on the object and accomplishing the actual serving of the client’s request. After the request is processed on the Application server, the request “returns” to the Logging server to
7.2 The checkpointing procedure

write its reply in the log. This step is followed by sending the reply to the client. From time to time the infrastructure initiates a checkpointing request (by initiating a \texttt{get\_state} call on the \texttt{Application server}). This call is treated in a similar manner to the calls generated by clients.

To summarize, there are two categories of call information logging: (1) calls corresponding to the client requests and (2) calls corresponding to the \texttt{get\_state} request. There are two categories of reply logging: (1) the returned result by the client initiated calls, and (2) the result of the \texttt{get\_state} call, i.e. the completed checkpointing of the state. When storing the call information, the log is checked for replies, thus a reply logging operation has to wait in a queue. When a reply logging operation is performed, no call information logging goes on, and so on. Hence at the \texttt{Logging server} the calls are executed sequentially in a run to completion manner (i.e. no interruptions of one execution by another).

We use a combination of forked [153, 121] and sequential checkpointing: from time to time, the checkpoint procedure is initiated by a parallel process. However, this process only inserts a checkpointing request in a queue where normal requests are queuing to obtain service. As mentioned before, while the checkpointing request is serviced, no other request is processed in the server.

The checkpointing procedure has six phases:

1. waiting to mark the last (client) call record
2. marking the last call record
3. waiting to execute the “state reading” on the \texttt{Application server}
4. reading the state from the server
5. waiting to record the state

Figure 7.1: The logging and application servers and their relation
Chapter 7. Computing the optimal checkpointing interval

6. recording the state in the Logging server

In what follows we explain the above in more detail.

Upon arrival of a get state request, the last call record from the log has to be marked. This is needed because the calls that arrived before the get state request shall all be executed at the Application server before get state, and therefore their call records can be removed from the call log after state recording. The marking operation is in fact achieved by logging the get state call that was mentioned in the last section. Thus, the call to get state first spends some time at the Logging server (phase 1 above, denoted by the first $W_L$ in Figure 7.2), and then its own logging amounts to marking the previous call (phase 2 above, denoted by the first $S_L$ in Figure 7.2).

Next, the call to get state arrives at the Application server, and waits for all queuing update calls to finish execution on the object. After this, the get state request is executed on the application object: the state of the object (as left by the last update operation executed before get state) is read (phases 3 and 4 above, denoted by $W_A$ and $S_A$ in Figure 7.2).

Finally, the state that was read by get state is recorded at the Logging server, after all queuing calls at that server. Thus, before “exiting the system”, the completed get state request again spends some time in the queue of the Logging server and on it (phases 5 and 6 above, denoted by the second $W_L$ and $S_L$ in Figure 7.2).

The six phases constitute the whole checkpointing procedure, and are repeated $P$ (the checkpointing interval) time units after the last phase was completed. See Figure 7.2 for a visualization of this procedure. Note that failures can appear at any time between instance $j$ and $j + 1$ of checkpointing.

The goal of our work is to support the engineer that uses such an infrastructure by providing the optimal value of the average checkpointing interval, under certain
simplifying assumptions, after inputting the other parameters (e.g. average client request arrival rate, average failure rate, average service rates) of the system. The *optimal* checkpointing interval is the value that maximizes the average time the system is available for client request processing, or that minimizes the average response time of client requests. The two values, as the experiments will show, are different, as the optimization criteria are not overlapping. Also, some of the assumptions made in the model are common for both optimization settings and others are different.

### 7.3 Basic model

This section will present common modelling aspects when considering both optimization criteria: average availability and average response time.

Throughout this and the next chapter, capital letters (with or without subscript, e.g. $C$, $S_A$) designate random variables.

In the approximating Markov model below, $P$ is a random variable with generic distribution\(^1\) with $E[P] = p$. $p$ will be expressed as a function of different given parameters and is chosen such that the system spends the least time in failover operations, as well as checkpointing, or it provides an optimal average response time when processing client requests.

The aim of the models is to express the average time the system is available for client request processing, respectively the average response time, as a function of the “unknown” quantity $p$ (or some intermediate $x$, such that $p$ is a function of $x$), and the “known” system parameters. Once this function is found, the main goal of finding the $p$ that maximizes, respectively minimizes it, can be achieved.

Following the description in the last section, we model the server side of the application as a set of two servers (in terms of queuing theory). The first server where

\(^1\)Same as the random variable $T$ from Vaidya’s work [153].
client requests arrive to be served first (for logging purposes) is the Logging server (server 1 in Figure 7.3). The second server at which requests arrive (to perform the actual computations) is the Application server (server 2 in Figure 7.3). The two servers form a pipeline. Customers of server 1 are the requests that arrive originally from the client, as well as the checkpointing requests. These customers, when departing from server 1, become customers of server 2. Customers of server 2 when departing, return to server 1 (see the “feedback” arrow in Figure 7.3). From server 1 they depart as “reply to client” (see the arrow in 7.3 that corresponds to that arrow in the Figure 7.1).

7.4 Modelling assumptions

This section will present our assumptions for the basic model. Assumptions specific to each model will be presented in the sections dedicated to each of them.

In line with earlier works [148, 63, 164, 87, 30, 95] we assume that a failure is detected as soon as it occurs. We also assume that no failure occurs during a failover [164, 63, 95, 30]. We assume that the first failure arrives after the first checkpoint operation was completed. The average arrival rate of client requests is assumed to be higher than the average arrival rate of checkpointing requests.

We will approximate the interarrival time between two consecutive checkpointing requests with independent identically distributed random variables, with exponential distribution with average \( \frac{1}{\lambda} \). As a consequence, the random variable \( \mathbb{P} \) is not deterministic\(^2\).

A simplifying assumption we make is that the probability distribution of service times on the two servers does not depend on the type of request the server is processing. This means, for example, that the probability distribution for the service time of a call information logging request on server 1 is the same as the distribution for the service time of a reply logging request. Also, the state transfer part of the failover time is assumed to be a constant \( s \). Related to queue analysis, we assume that no infinite queues build up at any of the servers.

Client request interarrival times are independent identically distributed variables with exponential distribution. Service times on the two servers and call replay time are also exponentially distributed. Failure interarrival time distribution is also exponential, as in [164, 121, 153, 87, 63, 27].

\(^2\)However, it will be shown in Chapter 8, that the optimal value obtained for \( p = E[p] \), works rather well in a simulation setting, when a deterministic \( p \) is chosen.
Finally, in the current work we assume that all client requests arriving at server 1, will proceed to the queue of server 2 and will be processed on server 2. In other words, no client request arriving at server 1 has been resent by the client, i.e. no requests find a reply in the call log.

### 7.5 Queuing analysis

For both optimization cases we will analyze two queues: the one at server 1, and the one at server 2. What we need, are the distributions of interarrival times of customers of the two servers, as well as distributions of their service times. Since we assume that all “external” customer interarrival time distributions as well as service time distributions are exponential (see Section 7.4), all “internal” customer interarrival time distributions are also exponential. By “internal” customers we mean, customers of server 2 and customers of server 1 that are fed back from server 2. Let the average arrival rate of client requests be \( \lambda_2 \). Since all interarrival time and service time distributions for the two queues are exponentially distributed, the two queuing systems are \( M/M/1 \), and, besides, they form a Jackson network [3].

The average arrival rate for customers at server 1 can be computed as follows. The queue is composed of:

- “external” customers, given by normal (client) and checkpointing (infrastructure) requests: these have average arrival rate \( \lambda = \lambda_1 + \lambda_2 \)
- “internal” customers, given by “feedback” from server 2: these have average arrival rate \( \lambda \) (equal to the average departure rate from server 2)

Thus, the total average arrival rate of customers at server 1 is \( 2\lambda \).

The queue at server 2 contains customers that departed from server 1 that will eventually return to server 1. Hence, the average arrival rate of customers at server 2 is \( \lambda \).

Figure 7.3 shows the servers, as well as the customer interarrival and service time distributions at each of them. The text under each box (e.g. \( \mu_1 e^{-\mu_1 t} \)) shows the service time distribution on the respective server. Thus, the average service rates on the two servers are \( \mu_1 \) and \( \mu_2 \), respectively. The figure does not show the average request replay rate (this will be denoted by \( \mu_3 \)). The reason for considering the average replay time different from the average service time on the Application
server is that the replay of calls is assumed not to involve any overhead that the normal request processing would involve. Replay of requests is performed on the server object locally, in a batch, not incurring middleware overhead. As will be seen in the numerical studies section, the value of the replay rate will be chosen larger than the service rate on the Application server.

The consequence of the assumption “no infinite queues build up at the servers” is that the following relations hold between \( \lambda_1, \mu_1, \) and \( \mu_2: \lambda < \mu_2 \) and \( 2\lambda < \mu_1. \) The consequence of the assumptions “average rate of incoming client request is higher than the average rate of checkpointing request arrival” is: \( \lambda_2 < \lambda_1. \)

The average time a customer waits in the \((M/\infty/1)\) queues of server 1 and server 2, is given by classical queuing theory formulas [3], and is described by 
\[
E[W_i] = \frac{2\lambda_1 + 2\lambda_2}{\mu_1(\mu - 2\lambda - 2\lambda_2)} \quad \text{and} \quad E[W_i] = \frac{2\lambda_2 + 2\lambda_1}{\mu_2(\mu_2 - \lambda_1 - \lambda_2)},
\]
respectively.

Some variables used in our optimization analysis are shown in figures 7.3 and 7.4 and summarized in Table 7.1. The average values of the random variables \( \mathbf{N} \) and \( \mathbf{F} \) will be computed in the respective sections dedicated to the two optimization criteria. Besides, new variables specific to each model will be introduced there.

### 7.6 Optimal checkpointing for maximum availability

The goal of this modelling is to obtain the expression for the average time that the system is available for client request processing. To reach our goal let us first consider the failover time random variable.

Without loss of generality, we will perform our analysis considering a failure occurring between the time when the \( j^{th} \) (\( j \geq 1 \)) checkpointing request left the system (see arrow “to client/infrastructure” in Figure 7.1), i.e. the \( j^{th} \) state recording happened, and the time when the \( j + 1^{st} \) state recording happened, i.e. the \( j + 1^{st} \) checkpointing request left the system. Let the two points at which these state recordings take place be \( T_j \) and \( T_{j+1} \) respectively (see Figure 7.4). A failure may occur anywhere between \( T_j \) and \( T_{j+1} \), including any of the intervals corresponding to the phases of the checkpointing procedure. Let \( T \) be the random variable denoting the interarrival time between checkpoints (\( E[T] = \frac{1}{\lambda_1} \)). For reasons of symmetry we have that \( T_{j+1} - T_j = T \) (in terms of random variable distributions, it does not matter if \( T \) is written as the sum of \( P \) and the first group \( W_l, S_l, W_h, \) etc., or the second group; in other words, \( T_{j+1} - T_j = T', \) but \( T \) and \( T' \) have identical probability distributions). In sequel we will only refer to \( T \).

In this section, we will assume that after each failure (and the corresponding failover) the system is started out with no backlog. Thus, no client requests are
7.6 Optimal checkpointing for maximum availability

\[
\begin{align*}
W_L &= \text{wait time in the queue of the Logging server} \\
W_A &= \text{wait time in the queue of the Application server} \\
S_L &= \text{service (processing) time on the Logging server} \\
S_A &= \text{service (processing) time on the Application server} \\
T &= \text{time between two checkpoint request arrivals} \\
Y &= \text{distance between the moment of failure and the moment of the last state recording (T)} \\
P &= \text{time between the end of checkpoint j and beginning of checkpoint j + 1} \\
T_S &= \text{time the checkpointing request spends on the two servers (T_S = S_L + S_A + S_b)} \\
T_C &= \text{time the checkpointing request spends in the system (T_C = W_L + W_A + W_L + T_S)} \\
T_R &= \text{call replay time} \\
s &= \text{state transfer time} \\
N &= \text{number of call records in the log that have to be replayed at failover} \\
F &= \text{failover time}
\end{align*}
\]

Table 7.1: Variables used in the models

\[
\begin{align*}
E[W_L] &= \frac{\lambda_2}{\mu_1(\mu_1 - 2\lambda_1 - 2\lambda_2)} \\
E[W_A] &= \frac{\lambda_2}{\mu_2(\mu_2 - \lambda_1 - \lambda_2)} \\
E[S_L] &= \frac{1}{\mu_1} \\
E[S_A] &= \frac{1}{\mu_2} \\
E[T] &= \frac{1}{\lambda_T} \\
E[P] &= p \\
E[T_S] &= \frac{1}{\mu_2} + \frac{1}{\mu_1} \\
E[T_C] &= \frac{1}{\mu_2 - \lambda_1 - \lambda_2} + \frac{\lambda_1 - 2\lambda_2}{\mu_1 - 2\lambda_1 - 2\lambda_2} \\
E[T_R] &= \frac{1}{\mu_3} \\
s &= 1 \\
N &= 0 \\
F &= 0
\end{align*}
\]

Figure 7.4: Relation between different random variables

queued up at any of the two servers.

Table 7.2 summarizes the random variables specific to the basic model used when maximizing average availability. The average value of all the variables will be computed in the rest of this section.
Chapter 7. Computing the optimal checkpointing interval

<table>
<thead>
<tr>
<th>$T_A$</th>
<th>time the system is available for client request processing between the occurrence of two consecutive failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_F$</td>
<td>time between two failures</td>
</tr>
<tr>
<td>$C$</td>
<td>number of checkpoint requests that arrive and are executed between two failure occurrences</td>
</tr>
</tbody>
</table>

| $E[T_A]$ = $E[T_F] - E[F] - E[C] \times E[T_F]$ |
| $E[T_F] = \frac{1}{\lambda_A}$ |
| $E[C] = \frac{E[T_F] - EF}{E[T_F]}$ |

Table 7.2: Availability random variables

The failover time is dependent on the state transfer time (constant $g$) and the number of requests to be replayed. In other words, the failover time random variable depends partly on the number of call records in the call log at the moment of failure. This number (denoted by the random variable $N$ in Table 7.1) is the same as that of client requests that managed to record their call information in the log between the moment the checkpointing request left server 1 on its way to server 2 and the moment of failure (from Figure 7.4, the length of this time interval can be computed as $W_A + S_A + W_L + S_L + Y$). The calls that recorded their information in the log during this interval, are exactly the ones that arrived at the application server (server 2) during the interval. The average rate of arrival of these calls at server 2 is $\lambda_A$. Hence, to obtain $E[N]$, we can use Little’s formula [3]:

$$E[N] = \lambda_A E[W_A + S_A + W_L + S_L + Y | Y < T]$$,

where

$$E[W_A + S_A + W_L + S_L + Y | Y < T] =$$

$$= E[W_A] + E[S_A] + E[W_L] + E[S_L] + E[Y | Y < T]$$ \hspace{1cm} (1)

Using well known properties of conditional expectation described in Section 2.5.1, we obtain (remember that the pdfs of $Y$ and $T$ are $\lambda_A e^{-\lambda_A t}$ and $\lambda_2 e^{-\lambda_2 t}$ respectively):
Now, by substituting values from Table 7.1 in (1) and by using (2), we obtain $E[N]$:

$$E[N] = \lambda_1 \left( \frac{\lambda_1 + \lambda_2}{\mu_2(\mu_2 - \lambda_1 - \lambda_2)} + \frac{1}{\mu_2} + \frac{2\lambda_1 + 2\lambda_2}{\mu_1(\mu_1 - 2\lambda_1 - 2\lambda_2)} + \frac{1}{\mu_1} + \frac{1}{\lambda_1 + \lambda_2} \right)$$

$$= \lambda_1 \left( \frac{1}{\mu_2 - \lambda_1 - \lambda_2} + \frac{1}{\mu_1 - 2\lambda_1 - 2\lambda_2} + \frac{1}{\lambda_1 + \lambda_2} \right)$$

(3)

The time for call replay is given by multiplying $E[N]$ with the average call replay time ($E[T_R]$). By substituting values from Table 7.1 and using (3) the average failover time can be computed as well.

$$E[F] = E[T_R] \times E[N] + s =$$

$$= \frac{\lambda_1}{\mu_3} \left( \frac{1}{\mu_2 - \lambda_1 - \lambda_2} + \frac{1}{\mu_1 - 2\lambda_1 - 2\lambda_2} + \frac{1}{\lambda_1 + \lambda_2} \right) + s$$

(4)
(\(E[T_A]\)) can be computed from the formula in Table 7.2, by substituting \(E[C]\):

\[
\]

\[
= (E[T_F] - E[F]) \left(1 - \frac{E[T_S]}{E[T]}\right)
\]

Thus, by using (4), and by replacing \(E[T_F]\) with its value from Table 7.2, and \(E[T_S]\) with values from Table 7.1, we obtain for \(E[T_A]\):

\[
E[T_A] = \left(\frac{1}{\lambda_2} - \frac{\lambda_1}{\mu_3} \left(\frac{1}{\mu_2 - \lambda_1 - \lambda_2} + \frac{1}{\mu_1 - 2\lambda_1 - 2\lambda_2} + \frac{1}{\lambda_2 + \lambda_2}\right) - s\right) \times
\]

\[
\times \left(1 - \lambda_2 \left(\frac{1}{\mu_2} + \frac{2}{\mu_1}\right)\right)
\]

(5)

As mentioned earlier, our goal is to obtain the \(p\) that maximizes \(E[T_A]\). However, we obtained \(E[T_A]\) as a function of \(\lambda_2\). The relation between \(\lambda_2\) and \(p\) is given by the following (recall that \(E[P] = p\), and see Figure 7.4):

\[
\]

(6)

By replacing in (6), the values for \(E[T]\), \(E[P]\), \(E[S_L]\), \(E[W_L]\), \(E[S_A]\), and \(E[W_A]\) from Table 7.1, we obtain:

\[
\frac{1}{\lambda_2} = p + \frac{2}{\mu_1} + \frac{2(2\lambda_1 + 2\lambda_2)}{\mu_1(\mu_1 - 2\lambda_1 - 2\lambda_2)} + \frac{1}{\mu_2} + \frac{\lambda_1 + \lambda_2}{\mu_2(\mu_2 - \lambda_1 - \lambda_2)}
\]

which leads to

\[
p = \frac{1}{\lambda_2} + \frac{1}{\lambda_2 + \mu_1 - \mu_2} + \frac{2}{2\lambda_2 + 2\lambda_1 - \mu_1}
\]

(7)

To obtain the value of \(\lambda_2\), that maximizes \(E[T_A]\) (in (5)) one has first to study the concavity property of the function \(E[T_A]\) in terms of \(\lambda_2\) (in a shorter form, \(f(\lambda_2)\)). Then, for \(\lambda_2\) in the intervals where the function is concave (if it is somewhere) solve the equation \(f'(\lambda_2) = 0\). Finally the \(\lambda_2\) that gives the “absolute” maximum \(f(\lambda_2)\) \((E[T_A]\)) should be chosen.

It is visible from the formula in (5) that \(f(\lambda_2)\) is a rational function and it is therefore continuous on regions between its poles (values of \(\lambda_2\) that are roots
of the denominator). Thus, the concavity study can be conducted only in the regions between two poles where the function is continuous. The poles are: \(-\lambda_s, \mu_2 - \lambda_1\) and \(\frac{\mu_1 - 2\lambda_1}{2}\). As we will assume that the value of the average service rate on the Logging server is much larger than that on the Application server \((\frac{\mu_1}{\mu_2} \gg 2)\), we will have the relation \(\mu_2 - \lambda_1 < \frac{\mu_1 - 2\lambda_1}{2}\). As the values of \(\lambda_2\) (that maximize the value of \(E[T_4]\)) interesting to us are positive, and even more, smaller than \(\lambda_1\), the poles we will consider are only \(\mu_2 - \lambda_1\) and \(\frac{\mu_1 - 2\lambda_1}{2}\). Thus, the interesting intervals where to study the functions concavity are \((0, \mu_2 - \lambda_1)\) and \((\mu_2 - \lambda_1, \frac{\mu_1 - 2\lambda_1}{2})\). As \(\mu_2 - \lambda_1 > 0 (\mu_2 > \lambda_1)\), and \(\mu_1 >> 2\mu_2 (\mu_1 >> 2\lambda_1\), and thus \(\mu_1 > 3\lambda_3\), we have that \(\mu_1 - 2\lambda_1 > \lambda_1\), and as a consequence, the maximizing \(\lambda_2\) cannot be in the third interval, as we must have \(\lambda_2 < \lambda_1\). Of course, it is possible that on each of the intervals \((0, \mu_2 - \lambda_1)\) and \((\mu_2 - \lambda_1, \mu_1 - 2\lambda_1)\) the function is neither convex, nor concave.

To study the concavity of \(f(\lambda_2)\) one must find the second derivative \(f''(\lambda_2)\) and study its sign. Below we will give the first and second derivatives of \(f(\lambda_2)\):

\[
f'(\lambda_2) = \left( g - \frac{1}{\lambda_s} \right) \left( \frac{1}{\mu_2} + \frac{2}{\mu_1} \right) + \frac{\lambda_1}{\mu_3} \left( \frac{2\mu_2 - \lambda_1 - 2\lambda_1}{(\lambda_2 - \mu_2 + \lambda_1)^2} + \frac{\mu_2 - 2\lambda_2 - 4\lambda_1}{(2\lambda_2 - \mu_1 + 2\lambda_1)^2} + \frac{\lambda_2 + 2\lambda_1}{(\lambda_2 + \lambda_s)^2} \right) \quad (8)
\]

\[
f''(\lambda_2) = -\frac{2\lambda_1}{\mu_3} \left( \frac{2\mu_2 - \lambda_1 - 2\lambda_1}{(\lambda_2 - \mu_2 + \lambda_1)^3} + \frac{\mu_2 - 2\lambda_2 - 4\lambda_1}{(2\lambda_2 - \mu_1 + 2\lambda_1)^3} + \frac{\lambda_2 + 2\lambda_1}{(\lambda_2 + \lambda_s)^3} \right) \quad (9)
\]

If we would further refine the two expressions from above, both \(f'(\lambda_2)\) and \(f''(\lambda_2)\) would turn out to be rational functions. The nominator in \(f'(\lambda_2)\) is a polynomial with degree four (it has degree two only when the two poles \(\mu_2 - \lambda_1\) and \(\frac{\mu_1 - 2\lambda_1}{2}\) are equal) (a), and the nominator in \(f''(\lambda_2)\) is a polynomial with degree six (it has degree three only when the two poles \(\mu_2 - \lambda_1\) and \(\frac{\mu_1 - 2\lambda_1}{2}\) are equal) (b). The two poles can never be equal as, an opposite situation would lead to the equality \(\mu_2 = \frac{\mu_1}{2}\), and we assumed that \(\mu_1 >> 2\mu_2\) (c).

Now, in order to study the sign of \(f''(\lambda_2)\) one needs to solve - according to (a) and (c), - a sextic equation, to find the roots of the nominator polynomial. Further, in order to solve the equation \(f'(\lambda_2) = 0\) one has to solve a quartic equation (according to (b) and (c)). According to Ferrari the quartic equation has analytic
Chapter 7. Computing the optimal checkpointing interval

solutions[19]. Abel’s impossibility result [1] tells that the sextic equation does not have analytic solutions. In any case (even for the quartic equation), since computing those solutions analytically is very tedious, we chose to use the tool Mathematica [162] that can perform numerical maximization of functions, given certain constraints. In this way we skip both the analysis of the concavity of the function (to determine the region where the maximum can be found) and the solving of the equation. We simply specify the function to maximize, the constraints that have to be satisfied by the maximizing solution, and let Mathematica do the operations for us. One of our constraints was that $0 < \lambda_2 < \lambda_1$. In order for Mathematica to be able to conduct the maximization of $f(\lambda_2)$ in $\lambda_2$ and return the maximizing value of $\lambda_2$, we had to give numerical values to all other system parameters: $\lambda_1$, $\mu_1$, $\mu_2$, $\mu_3$, $\lambda_e$ and $s$.

Independently of how it is obtained (analytically and then replacing parameters with numbers, or numerically), the value of $\lambda_2$ that maximizes $E[T_A]$ is a function of all the other parameters appearing in $E[T_A]$ ($\lambda_1, \lambda_e, \mu_1, \mu_2, \mu_3$). The value of $p$ that maximizes the average availability time, can then be computed by substituting the obtained value for $\lambda_2$ in equation (7).

7.7 Optimal checkpointing for minimum response time

In this section we will present the modelling and analysis aspects involved in the computation of the checkpointing interval when the optimization criterion is the average response time of a request. The queuing model used here is very similar to the one used when optimizing average availability. However, an accurate measure of response time turns out to benefit from modelling the backlog after failures.

Since the relation between $p$ (the checkpointing interval) and $\lambda_2$ (average checkpoint arrival rate) remains the same as in equation (7), we will, at the end of our analysis here, express the average response time as a function of the unknown $\lambda_2$. All assumptions listed in Section 7.4 hold for this model as well. However, some additional assumptions are necessary.

As shown in Figure 7.5 due to the extension of the basic mathematical model, the time line $(0, \infty)$ will be sliced in groups of failover, backlog processing and equilibrium intervals. Equilibrium corresponds to normal processing, i.e. when the system is not executing failover or backlog processing. The backlog processing in the interval during which the server system (mainly server 2, as $\mu_1 >> \mu_2$) has to process all requests queued up during failover. Note that failures for which the failover time is analyzed and included explicitly in the final formula of average re-
7.7 Optimal checkpointing for minimum response time

Figure 7.5: Equilibrium, failover and backlog processing

response time, are those that arise following the end of an equilibrium interval. These are shown in Figure 7.5. Even though failures can occur also during backlog processing intervals, implying new failovers, those failover times are incorporated in the expression of the backlog processing time. Thus, they will not appear explicitly in the final formula for response time that we provide.

7.7.1 New assumptions

Two different average failure arrival rates are considered, depending on whether the failure arrives during a backlog processing interval, or during equilibrium intervals. The average rate of failures during backlog processing \( \tilde{\lambda}_f \), will be larger than the “equilibrium” failure rate \( \lambda_f \), with \( \tilde{\lambda}_f = \frac{\lambda_f \theta}{\Lambda} \). During this time server 2 is busier with request processing.

We assume that reply logging “customers” of server 1 have higher priority than other customers (client or checkpointing requests logging call information), and the special “log reading customer” that appears only at the beginning of a failover. This assumption affects the failover time that now consists of (1) state transfer time (denoted by the constant \( g \)), (2) waiting for all reply logging customers present in server 1’s queue to be served, before the call information log can be read for the replay, (3) call replay time for requests prior to the failure. Also, it has a consequence on the way of computing the average number of calls that are present in the call log at the moment of replay.

In this work we do not treat the heavy traffic case separately. We obtain one formula for the average response time and do our analysis based on that, even for large values of the average request arrival rate. The separate treatment of the latter case is subject of future investigation following the work by Lehoczky about real-time queuing theory [93].
7.7.2 Minimizing average response time

The average response time is the average time a request spends in the system given by server 1 and server 2, from its arrival in the queue of server 1 as call logging request, until it leaves server 1 as reply to the client (see Figure 7.1). The system’s life cycle can be divided in two types of phases: failover when the response flow is stopped, and equilibrium and backlog processing when the system is processing requests. Hence, the average response time is basically given by two parts: the average time spent when actual processing of the request takes place plus the average extra time spent in the queue of some server due to failover.

Now, the latter fragment will be encountered by requests that arrive in the queue of server 1 during a failover, or during equilibrium just before a failure, so that their processing is delayed by the failover activities.

Without loss of generality, we will consider two particular points as delimiters of the interval, in which a failure marking the end of an equilibrium interval occurs, denoted by $T_j$ and $T_{j+1}$ (see Figure 7.6). As in our reasoning in Section 7.6 these are the points when the $j^{th}$ ($j \geq 1$) checkpointing request left the system, i.e. the $j^{th}$ state recording happened, and the time when the $j + 1^{st}$ state recording happened, respectively (see arrow “to client/infrastructure” in Figure 7.1).

Note that, although the picture does not reflect such a situation, our assumptions do not exclude the possibility of a failure occurring during any of the six phases of the checkpointing procedure (see Section 7.2 for a reminder of what these six phases are and Figure 7.4 for a graphical representation). Remember from Section 7.6 that $T_{j+1} - T_j = T$ (where $E[T] = \frac{T}{2}$).

Table 7.3 summarizes the random variables specific to the model used when minimizing average response time. The average value of all the variables will be computed in the rest of this section.
By detailing the earlier intuitive description, we can formulate the expressions for the lower and upper bound on the value of the average response time. Both have four terms: two of them correspond to the average times spent on the servers during equilibrium and backlog processing, respectively, while the other two correspond to the average extra times spent by requests arrived during failover and just before it, respectively. We enumerate them below:

- **Term 1** expresses the average processing and wait time spent in the system by requests that arrive and leave during equilibrium, weighted by the fraction of time the system is in equilibrium while enough time is left for a request to finish execution on server 2.

- **Term 2** expresses the average processing and wait time spent in the system by a request during backlog processing (this request either arrived during a failover inside or outside the backlog processing, or was already in the queue at server 1 when the failure occurred), weighted by the fraction of time the system spends in backlog processing.

- **Term 3** expresses the average processing, wait and extra wait time spent by requests that arrive while the system is executing a failover, weighted by the fraction of time the system is executing failover.

- **Term 4** expresses the average processing, wait and extra wait time spent by requests that arrived during equilibrium, but did not have enough time to
complete their execution on server 2, weighted by the fraction of time left of equilibrium till the failure occurrence, during which there is no time to leave the failed server 2.

The lower bound expression is obtained when assuming that the number of failures that occur during backlog processing is zero. The upper bound is given when assuming that at least one failure occurs during backlog processing, so that at least one new buildup has to be handled until the backlog processing period is over.

The formula for the lower bound on the average response time (each of the four terms appearing on a separate line) is given below:

\[
\begin{align*}
E[R_T]_{\text{lower}} &= \\
&= \left( E[T_F]_{\text{lower}} - E[B]_{\text{lower}} - \left( \frac{1}{\mu_1-2\lambda_1-2\lambda_2} \frac{1}{\mu_2-\lambda_1-\lambda_2} \right) \left( \frac{2}{\mu_1-2\lambda_1-2\lambda_2} \frac{1}{\mu_2-\lambda_1-\lambda_2} \right) \right) \left( \frac{E[T_F]_{\text{lower}} + E[F]}{E[T_F]_{\text{lower}} + E[F]} \right) + \\
&\quad \quad + \left( \frac{E[B]_{\text{lower}} \left( \frac{E[R]_{\text{lower}}}{\mu_2} + \frac{1}{\mu_1} \right)}{E[T_F]_{\text{lower}} + E[F]} \right) + \\
&\quad \quad + \left( \frac{E[F] \left( \frac{E[R]_{\text{lower}}}{\mu_2} + \frac{1}{\mu_1} \right)}{E[T_F]_{\text{lower}} + E[F]} \right) + \\
&\quad \quad + \left( \frac{\left( \frac{1}{\mu_1-2\lambda_1-2\lambda_2} \frac{1}{\mu_2-\lambda_1-\lambda_2} \right) \left( E[F] + \frac{1}{\mu_2} \right)}{E[T_F]_{\text{lower}} + E[F]} \right) \quad (10)
\end{align*}
\]

The formula for the upper bound of the average response time (each of the four terms appearing on a separate line) is:
As shown in the description above, we need to compute the following values: the average failover time \( (E[F]) \), the lower and upper bound for the average number of requests that have to be processed until the equilibrium is reached after the failover \( (E[N_b]_{lower} \text{ and } E[N_b]_{upper}) \), the lower and upper bound on the average backlog processing time \( (E[B]_{lower} \text{ and } E[B]_{upper}) \), and the corresponding expressions of the average time from the end of a failover until the moment of the next failure \( (E[T_F]_{lower} \text{ and } E[T_F]_{upper}) \).

To compute the average failover time let us recall our reasoning about failure occurrences and replaying of calls from Section 7.6. In the present context we have again that the number of call records to replay (denoted by the random variable \( N \) in Table 7.1) is the same as that of client requests that managed to record their call information in the log between the moment the checkpointing request left server 1 on its way to server 2 and the moment of failure. However, here we assume that reply logging customers of server 1 have higher priority than call logging customers. Therefore, after the checkpointing request read the state of the application and returned in the queue of server 1 to log its reply (the state) no call logging customer will be processed anymore until the moment the checkpointing request stores the state. Thus, in the present setting, the number of client requests that manage to log their call information, i.e. arrive in the queue of server 2 after the checkpointing request arrived there and before the failure occurred is equal to the number of arrivals at server 2 during the interval \( W_A + S_A + Y \). By using Little’s formula we can compute easily the average number of calls to replay (de-
Chapter 7. Computing the optimal checkpointing interval

noted by the random variable $N$ in Table 7.1), as the number of arrivals at server 2 during the interval $\hat{W}_A + S_A + Y$, where $Y < T$.

$$E[N] = \lambda_2 E[\hat{W}_A + S_A + Y | Y < T] = E[\hat{W}_A] + E[S_A] + E[Y | Y < T]$$

Using the result of (2) from Section 7.6, i.e. $E[Y | Y < T] = \frac{1}{\lambda_2 + \lambda_2}$, and the values from Table 7.1 for $E[\hat{W}_A]$ and $E[S_A]$, we obtain:

$$E[N] = \lambda_2 \times \left( \frac{\lambda_1 + \lambda_2}{\mu_2(\mu_2 - \lambda_1 - \lambda_2)} + \frac{1}{\mu_2} + \frac{1}{\lambda_2 + \lambda_2} \right)$$

$$= \lambda_2 \times \left( \frac{1}{\mu_2 - \lambda_1 - \lambda_2} + \frac{1}{\lambda_2 + \lambda_2} \right)$$

(12)

The failover time includes, besides the call replay time (computed as $E[N] \times E[T_R]$) and the state transfer time ($s$), the time taken to process on server 1, all reply logging requests present in its queue. The average number of reply logging customers in the queue of server 1, $E[N_R]$ is computed using classical average computing formulas for discrete random variables:

$$E[N_R] = \sum_{i=0}^{\infty} i P[N_R = i]$$

The probability of the queue containing $i$ reply logging customers at a point in time is computed with classical queuing theory formulas (where we can approximate the arrival rate with $\lambda_1$):

$$P[N_R = i] = (1 - \rho_1)^i \rho_1^i$$

where $\rho_1 = \frac{\lambda_1}{\mu_1}$

It follows that

$$E[N_R] = \sum_{i=0}^{\infty} i (1 - \rho_1)^i \rho_1^i = \frac{\rho_1}{1 - \rho_1} = \frac{\lambda_1}{\mu_1 - \lambda_1}$$

By replacing values from Table 7.1 the average failover time is


$$= \frac{\lambda_1}{\mu_3(\mu_3 + \lambda_2)} + \frac{\lambda_1}{\mu_3(\mu_2 - \lambda_1 - \lambda_2)} + \frac{\lambda_2}{\mu_1(\mu_1 - \lambda_1)} + s$$

(13)
To compute the upper bound for the average number of requests to be processed during a backlog processing interval \( E[N_b,\text{upper}] \) our reasoning is as follows: If no failure occurs during the backlog processing interval, the average number of requests processed is equal to the initial backlog size that gives the lower bound \( E[N_b,\text{lower}] \). This lower bound is equal to the average number of requests that arrive in the queue of server 1 during a failover.

\( E[N_b,\text{lower}] \) is computed by using Little’s formula (considering as interval for the arrivals, the average failover time \( E[F] \)) and then (13):

\[
E[N_b,\text{lower}] = \lambda E[F] = \lambda \left( \frac{\lambda_1}{\mu_3(\lambda_1 + \lambda_2)} + \frac{\lambda_1}{\mu_3(\mu_2 - \lambda_1 - \lambda_2)} + \frac{\lambda_1}{\mu_1(\lambda_1 - \lambda_1)} + s \right) \quad (14)
\]

However, during backlog processing (i.e. before the processing of the initial \( E[N_b,\text{lower}] \) requests is finished), failures can occur at the average rate \( \tilde{\lambda}_f = \lambda_f \frac{\mu_1}{\lambda_1} \). After each failover, while the system performs the backlog processing of the \( E[N_b,\text{lower}] \) accumulated requests, new failures occur, and thus “new sets” of \( E[N_b,\text{lower}] \) requests accumulate. An upper bound for the average number of requests that are present in the queue can be calculated by using the formula (with \( u(E[N_b,\text{upper}]) \) given below):

\[
E[N_b,\text{upper}] = E[N_b,\text{lower}] + u(E[N_b,\text{upper}]) \quad (15)
\]

Intuitively, \( u(E[N_b,\text{upper}]) \) can be seen as the average number of requests (from the backlog) that are still unprocessed when a failure occurs at time \( t \). An approximation of \( u(E[N_b,\text{upper}]) \) can be computed as follows. Note that we will only focus on the requests accumulated in the queue of server 2, the bottleneck, as \( \mu_1 \gg \mu_2 \). Given a time moment \( t_0 \), let the number of requests waiting in the queue of server 2 at \( t_0 \), be \( n(t_0) \). The number of requests in the queue of server 2 waiting to be processed at time \( t_0 + \Delta t \) (\( n(t_0 + \Delta t) \)) when no failures occur between \( t_0 \) and \( t_0 + \Delta t \), is approximately equal to the number of requests at time \( t_0 \) minus the number of requests that have left after being processed on the server, plus the number of requests that arrived in the queue. Considering that the rate of arriving is \( \lambda \), while the rate of departing is \( \mu_2 \), we have thus:

\[
n(t_0 + \Delta t) \approx n(t_0) + \lambda \Delta t - \mu_2 \Delta t = n(t_0) - (\mu_2 - \lambda) \Delta t
\]

The value of \( \Delta t \) for which the requests in the backlog have “vanished” is the value for which \( n(t_0 + \Delta t) \) becomes zero, i.e. \( \Delta t = \frac{n(t_0)}{\mu_2 - \lambda} \).
Now, at any time between $0$ and $\frac{\mu_2 - \lambda}{\lambda t}$, failures can occur. Thus, the average number of requests still remaining to be processed when a new failure occurs is given by the formula:

$$u(E[N_B]_{\text{upper}}) = \int_0^{\frac{\mu_2 - \lambda}{\lambda t}} (E[N_B]_{\text{upper}} - (\mu_2 - \lambda)t) \lambda e^{-\lambda t} dt$$

$$u(E[N_B]_{\text{upper}}) = E[N_B]_{\text{upper}} \left(1 - e^{-\lambda t} \frac{\mu_2 - \lambda}{\mu_2 - \lambda} \right) +$$

$$+ e^{-\lambda t} \frac{\mu_2 - \lambda}{\mu_2 - \lambda} \left( E[N_B]_{\text{upper}} + \frac{\mu_2 - \lambda}{\lambda t} \right) - \frac{\mu_2 - \lambda}{\lambda t}$$

$$= E[N_B]_{\text{upper}} - \frac{\mu_2 - \lambda}{\lambda t} + \frac{\mu_2 - \lambda}{\lambda t} e^{-\lambda t} \frac{\mu_2 - \lambda}{\mu_2 - \lambda}$$

(16)

Now, by replacing $E[N_B]_{\text{lower}}$ with its expression from (14) and $u(E[N_B]_{\text{upper}})$ with its expression from (16) in (15), by removing $E[N_B]_{\text{upper}}$ from both sides of the equality in (15), and rearranging the equation such that the unknown ($E[N_B]_{\text{upper}}$) ends up on the left side, we obtain the equation in $E[N_B]_{\text{upper}}$:

$$\frac{\lambda(\mu_2 - \lambda)}{\lambda_t \mu_2} - \frac{\mu_2 \lambda \lambda}{\lambda(\mu_2 - \lambda)} =$$

$$\lambda \left( \frac{\mu_2 - \lambda}{\lambda_t \mu_2} - \frac{\lambda_1}{\lambda_2 (\lambda + \lambda_2)} - \frac{\lambda_1}{\mu_3 (\mu_2 - \lambda_1 - \lambda_2)} - \frac{\lambda_1}{\mu_1 (\mu_1 - \lambda_1) - s} \right)$$

If we further simplify with $\frac{\lambda(\mu_2 - \lambda)}{\lambda_t \mu_2}$, we obtain the equation:

$$e^{-\frac{\mu_2 \lambda \lambda}{\lambda(\mu_2 - \lambda)}} = 1 - \frac{\lambda_t \mu_2 \left( \frac{\lambda_1}{\mu_1 (\mu_1 - \lambda_1)} + \frac{\lambda_1}{\mu_2 (\lambda_1 + \lambda_2)} + \frac{\lambda_1}{\mu_3 (\mu_2 - \lambda_1 - \lambda_2) - s} \right)}{\mu_2 - \lambda}$$

(17)

By solving equation (17) in $E[N_B]_{\text{upper}}$ we obtain:

$$E[N_B]_{\text{upper}} =$$

$$\frac{(\mu_2 - \lambda)}{\lambda t \mu_2} - \ln \left( \frac{\mu_2 - \lambda}{\lambda t \mu_2} - \frac{\mu_3 (\lambda_1 + \lambda_2) + \lambda_1}{\mu_3 (\mu_2 - \lambda_1 - \lambda_2) + s} \right)$$

(18)

$E[N_B]_{\text{upper}}$ will be used to determine the upper bound on the average backlog processing time ($E[B]_{\text{upper}}$). When no failures occur during backlog processing,
the average backlog processing time \(E[B_{\text{lower}}]\) is given by the average time taken to process the initial backlog on server 2, i.e. \(E[B_{\text{lower}}] = \frac{E[N_{b_{\text{lower}}}}}{\mu_2}\). By taking into account that failures indeed might occur during backlog processing, the upper bound on the average backlog processing time is given by the sum of \(E[N_{b_{\text{lower}}}]\) and the sum of all average failover times and all average times for processing accumulated requests for further failures until the original backlog processing interval ends. \(E[B_{\text{upper}}]\) is given thus by the formula below, where \(E[N_F]\) is the average number of failures occurring during backlog processing:

\[
E[B_{\text{upper}}] = E[B_{\text{lower}}] + E[N_F] \left(\frac{E[N_{b_{\text{lower}}}}}{\mu_2} + E[F]\right) \\
= \frac{E[N_{b_{\text{lower}}}}}{\mu_2} + E[N_F] \left(\frac{\lambda E[F]}{\mu_2} + E[F]\right) \\
= E[F] \left(\frac{\lambda}{\mu_2} + E[N_F] \left(\frac{\lambda}{\mu_2} + 1\right)\right) \\
(19)
\]

We have that

\[
E[N_F] = \sum_{i=1}^{\infty} P[N_F \geq i] \\
(20)
\]

Let \(n_j\) denote the number of calls in the backlog when processing resumes after failure \(j\) (\(j > 0\)). We can approximate \(E[n_j]\) \(\approx E[N_{b_{\text{upper}}}]\). Let \(p_{f,j}\) be the probability that a new failure occurs before the remaining \(n_j\) requests are processed, i.e. \(p_{f,j} = P[N_F \geq j + 1 | N_F \geq j]\). It follows that

\[
p_{f,j} = E\left[\int_0^{n_j} \lambda e^{-\lambda t} dt\right] = E\left[1 - e^{-\frac{\lambda n_j}{\mu_2 - \lambda}}\right] = 1 - E\left[e^{-\frac{\lambda n_j}{\mu_2 - \lambda}}\right]
\]

Let \(p_f\) be the probability of a failure occurring during a time interval needed for \(E[N_{b_{\text{upper}}}]\) requests to vanish from the backlog \(\frac{E[N_{b_{\text{lower}}}}}{\mu_2 - \lambda}\), i.e. \(p_f := 1 - e^{-\frac{\lambda E[N_{b_{\text{upper}}}}}{\mu_2 - \lambda}}\).

By Jensen’s inequality \(71\), because \(e^{-\frac{\lambda n_j}{\mu_2 - \lambda}}\) is a convex function, we have that

\[
p_{f,j} \leq 1 - e^{-\frac{\lambda E[n_j]}{\mu_2 - \lambda}} = p_f
\]
Chapter 7. Computing the optimal checkpointing interval

Now we can compute $P[N_F \geq i]$. We have that

$$P[N_F \geq i] = P[N_F \geq i | N_F \geq i-1]P[N_F \geq i-1]$$

$$= \prod_{j=1}^{i} P[N_F \geq j | N_F \geq j-1]P[N_F \geq 0]$$

$$= P_{x, i-1}P_{x, i-2}...P_{x, 0} \leq P_{x}^{i}$$

Therefore, the average number of failures occurring during backlog processing is upper bounded:

$$E[N_F] \leq \sum_{i=1}^{\infty} P_{x}^{i} = \frac{P_{x}}{1 - P_{x}} = e^{E[N_F]_{\text{upper}}} \cdot \frac{\lambda x}{\mu^2 - \lambda} - 1 \quad (21)$$

Now, by replacing the upper bound on $E[N_F]$ (from (21)), and $E[F]$ (from (13)) in (19), we obtain:

$$E[B]_{\text{upper}} = \left( \frac{\lambda_1}{\mu_3(\lambda_x + \lambda_2)} + \frac{\lambda_1}{\mu_3(\mu_2 - \lambda_1 - \lambda_2)} + \frac{\lambda_1}{\mu_1(\mu_1 - \lambda_1)} + s \right) \times$$

$$\times \left( \frac{\lambda}{\mu_2} + \left( e^{\frac{\lambda x}{\mu^2 - \lambda}} - 1 \right) \left( \frac{\lambda}{\mu_2} + 1 \right) \right) \quad (22)$$

The value of the average time till the next failure, corresponding to the situation when the average number of failures during backlog processing is larger than zero ($E[T_F]_{\text{upper}}$) will be computed by using the two probability distributions for the failure interarrival: exponential with rate $\lambda_x$ (during equilibrium), respectively with rate $\tilde{\lambda}_x = \lambda_x \frac{\mu}{\lambda}$ (during backlog processing).

$$E[T_F]_{\text{upper}} = E \left[ \int_{0}^{B} t \tilde{\lambda}_x e^{-\lambda_x t} dt + \int_{B}^{\infty} t \lambda_x e^{-\lambda_x t} dt \right]$$

$$= E \left[ \frac{1}{\tilde{\lambda}_x} - (B + \frac{1}{\tilde{\lambda}_x}) e^{-\lambda_x B} + (B + \frac{1}{\lambda_x}) e^{-\lambda_x B} \right]$$
7.7 Optimal checkpointing for minimum response time

Using Taylor expansion [103] as far as the second degree, about \( B = E[B]_{\text{upper}} \), we obtain:

\[
E[T_F]_{\text{upper}} \approx E \left[ \frac{1}{\lambda_f} - \left( E[B]_{\text{upper}} + \frac{1}{\lambda_f} \right) e^{-\lambda_f E[B]_{\text{upper}}} + \left( E[B]_{\text{upper}} + \frac{1}{\lambda_f} \right) e^{-\lambda_f E[B]_{\text{upper}}} \right] \\
= \frac{1}{\lambda_f} - \left( E[B]_{\text{upper}} + \frac{1}{\lambda_f} \right) e^{-\lambda_f E[B]_{\text{upper}}} + \left( E[B]_{\text{upper}} + \frac{1}{\lambda_f} \right) e^{-\lambda_f E[B]_{\text{upper}}} 
\]  

(23)

When the average number of failures during backlog processing is zero, the value of the average time till the next failure (\( E[T_F]_{\text{lower}} \)) is simply computed by subtracting the average failover time from the average time between two failures (\( \frac{1}{\lambda_f} \)). So,

\[
E[T_F]_{\text{lower}} = \frac{1}{\lambda_f} - \left( \frac{\lambda_1}{\mu_3(\lambda_f + \lambda_2)} + \frac{\lambda_1}{\mu_3(\mu_2 - \lambda_1 - \lambda_2)} + \frac{\lambda_1}{\mu_4(\mu_1 - \lambda_1)} + s \right) 
\]

We will not expand further the expression of \( E[R_T]_{\text{upper}} \) or \( E[R_T]_{\text{lower}} \) as it would yield a formula that is not followable.

The expression used to minimize the average response time is \( E[R_T]_{\text{upper}} \). Like in the previous section where the optimization criterion was the average availability, we will consider \( E[R_T]_{\text{upper}} \) as a function in \( \lambda_2 \) (the average checkpoint arrival rate), \( g(\lambda_2) \). Although we do not expand the expression of \( E[R_T]_{\text{upper}} \) further (by replacing \( E[I_B]_{\text{upper}} \) and \( E[B]_{\text{upper}} \) with their expression from (18) and (22)), from equations (18), (22) and (11) it is visible that the average response time can be written as a function of the seven parameters: \( \mu_1, \mu_2, \lambda_1, \lambda_2, \lambda_f, \mu_3, \) and \( s \). Considering, as in the previous section, all parameters beside \( \lambda_2 \), as so-called constants, we obtain \( g \) in only one unknown. It has to be mentioned that the relation between \( p \) (checkpointing interval) and \( \lambda_2 \) remains the same as that in equation (7):

\[
p = \frac{1}{\lambda_2} + \frac{1}{\lambda_2 + \lambda_1 - \mu_2} + \frac{2}{2\lambda_2 + 2\lambda_1 - \mu_1} 
\]

To obtain \( \lambda_2 \) (and then \( p \)) that minimizes the average response time, we will use only the formula for the upper bound on the average response time. The lower bound expression will be used only in expressing the constraint on the obtained value of \( \lambda_2 \) that minimizes the expression in (11). One has to first study the convexity of the function \( g(\lambda_2) \). Then, for \( \lambda_2 \) in the intervals where the function is convex solve the equation \( g'(\lambda_2) = 0 \).
Chapter 7. Computing the optimal checkpointing interval

We will not further pursue the computation of the analytic formulas to obtain the minimal average response time. However, we use the tool Mathematica to perform the minimization of the average response time. To gain interesting insights, we also consider the average backlog processing time as a function $h(\lambda_2)$ and study the value of $\lambda_2$ that minimizes it.

Independently of how it is obtained (analytically and then replacing parameters with numbers, or numerically), the value of $\lambda_2$ that minimizes $E[R_T]$ is a function of all the other parameters appearing in $E[R_T]$ ($\lambda_1$, $\lambda_3$, $\mu_1$, $\mu_2$, $\mu_3$, $s$). The value of $p$ that minimizes the average response time, can be computed by substituting the obtained value for $\lambda_2$ in equation (7).

7.8 Summary

Let us summarize the models obtained in this chapter in terms of the goals we defined earlier:

- obtaining the formulas for average availability ($E[T_1]$) and the approximation of the average response time ($E[R_T]$), described as functions of the parameter $\lambda_2$ that is the average checkpoint arrival rate; the functions are $f(\lambda_2)$ and $g(\lambda_2)$ in sections 7.6 and 7.7 respectively
- studying the relation between the parameter $\lambda_2$ and $p$, the checkpointing interval
- finding a straightforward way to obtain the optimal $p$ given the optimization criteria:
  - if the optimization criterion is the average availability, maximize $f(\lambda_2)$, then replace the maximizing value of $\lambda_2$ in the formula of $p$
  - if the optimization criterion is the average response time, minimize $g(\lambda_2)$, then replace the minimizing value of $\lambda_2$ in the formula of $p$

One might ask why the computation of the maximum availability did not need a model to include backlog processing. The answer is that we do not consider the backlog processing to affect availability. During backlog processing, in fact, the server is available for client request processing.

In the next chapter we will present interesting relations between the above functions (e.g. $\lambda_2$ that maximizes (minimizes) the average availability (average
response time), $E[T_A]$, $p$, $E[R_T]$, etc.) and some of the system parameters $\lambda_1$, $\lambda_2$, $\mu_2$, $\mu_3$ and $\mu_3$. Besides, we will present the results obtained after simulating the basic mathematical model used for maximizing average availability.
Chapter 8

Analysis of the optimization models

This chapter presents our findings when studying the different quantities involved in our analytical models of Chapter 7, as a function of the average client request arrival rate and average failure rate. It also presents results of our simulations devised to validate our mathematical model itself, as well as the approximation we made in connection to the checkpoint interarrival time.

Some examples of the interesting findings presented in this chapter are:

- the minimum response time does not present significant differences as the system is exposed to differing failure rates,
- the optimal checkpoint arrival rate behaves similar independently of failure rate and of optimization criterion,
- the availability of the system when the response time is minimal is not too far from the maximum availability; similar results were seen for the response time
- the backlog processing model can be used to find the sensitivity of the maximum average response time to load variations

This chapter has two main parts:

- an availability / response time trade-off study based on replacing parameters with numbers in the formulas obtained in Chapter 7 (section 8.1);
• simulation results that validate the basic mathematical model that was used to compute the checkpointing interval in the basic model (section 8.2)

Throughout the chapter the study of maximum availability is based on the basic model and the study of the minimum response time is based on the refined model with backlog processing.

8.1 Numerical studies

In this section we summarize the results of our numerical studies based on the two models used to derive the formulas in Section 7.6 and 7.7, using the tool Mathematica. The results are based on numerical solutions that allow us to plot interesting variables against variations in other parameters such as average client request arrival rate, failure rate and call replay rate. These studies provided a deeper understanding of the system modelled in the previous sections.

In the series of studies that we will show below, $\mu_3$ (average service rate on server 1) and $\mu_2$ (average service rate on server 2) are fixed at values 50 and 5, respectively. Fixing them was reasonable since they are infrastructure respectively application related. With those two quantities fixed, it is meaningful to study how the system handles different external conditions such as different average client request rates ($\lambda_1$), and different average failure rates ($\lambda_f$). The choice of the two values (50 and 5) was based on the need to impose a large ratio between the average service rate on the Logging server (server 1) and the average service rate on the Application server (server 2). Also, the call replay rate $\mu_3$ will be fixed at the value 12, a value larger than that of $\mu_2$.

The studies included experiments for different values of $\delta$ (state transfer time). Our observations confirmed our expectations: by varying $\delta$ the curves obtained kept the same shape, while only moving on one of the axes with constant values. All figures in the next sections use $\delta = 0$.

In order to obtain the value of the checkpointing interval ($p$) for which $\mathbb{E}[T_A]$ ($\mathbb{E}[R_T]$) is maximal (minimal), given all other parameters as fixed, we used Mathematica to perform the calculations as follows. We fixed, i.e. assigned numerical values to all parameters of $\mathbb{E}[T_A]$ ($\mathbb{E}[R_T]$) that were considered as given ($\lambda_1$, $\mu_2$, $\mu_1$, $\mu_3$, $\lambda_f$, $\delta$), and asked Mathematica to maximize (minimize) the expression of $\mathbb{E}[T_A]$ ($\mathbb{E}[R_T]$) in $\lambda_2$. The constraints we introduced in Mathematica when maximizing (minimizing) the given function, were related to our assumptions about no infinite queues building up at the two servers ($\lambda_1 + \lambda_2 < \mu_2$, and $2\lambda_1 + 2\lambda_2 < \mu_1$),
and the average rate of checkpointing being less than the average rate of incoming client requests ($\lambda_2 < \lambda_1$).

The goals of the numerical studies were:

- to find out the behaviour of the minimum average response time when considered as a function of $\lambda_3$, and how the average response time when no failures and/or no checkpointing happens, relates to it.

- to find out how sensitive the average response time is in terms of variations in the load. More precisely, given an infrastructure that is optimized for a given load in terms of request arrival rates, we studied how long the maximum average response time for the service can become if the load is increased beyond the predefined load parameter of the system.

- to find out the behaviour of the maximum average availability when considered as a function of $\lambda_3$, respectively $\lambda_4$, and the extent (especially the scale) of the dependency. Besides, we wanted to see how a fixed $\lambda_4$, respectively $\lambda_3$ influenced the studied behaviour.

- to find out the behaviour of the average checkpoint arrival rate ($\lambda_2$) that maximizes (minimizes) the average availability (response time), when considered as a function of $\lambda_3$, respectively $\lambda_4$, for given $\lambda_4$ respectively $\lambda_3$; when considering $\lambda_2$ as a function of $\lambda_4$ we expect to find out especially the value of $\lambda_4$ for which the desired relation that (on average) “checkpointing is done more often than failures occur” (i.e. $\lambda_2 > \lambda_4$) does no longer hold.

- to find out the relation between the $\lambda_2$ that minimizes the average response time ($E[R_T]$) and the $\lambda_2$ that minimizes the average backlog processing time ($E[B]$), when considering them as functions of $\lambda_2$ with a fixed $\lambda_4$.

- to find out (if it exists), the average client request rate ($\lambda_4$) for which our assumption $p > E[T_c]$ (with $p$ and $E[T_c]$ obtained for $\lambda_2$ that maximizes the average availability, respectively minimizes the average response time) no longer holds, while “no infinite queues at the servers” are built up; this for given service rates ($\mu_1$ and $\mu_2$) and a given average failure rate ($\lambda_4$) and average call replay rate($\mu_3$). This is an undesirable point since the system is close to continuously checkpointing in order to keep up with the failover time requirement. Note that, in case of minimizing $E[R_T]$ we consider only $E[T_c]$, i.e. the average time the checkpointing request spends in the system in
an equilibrium period. We believe that even this study can offer us enough information about how the client request arrival rate influences the relation between the two studied quantities.

- to find out how the two optimization criteria relate to each other. In this context, we looked for the relation between the value of $\lambda_2$ that maximizes $E[T_a]$ respectively minimizes $E[R_T]$, when considered as a function of $\lambda_1$ and fixing $\lambda_2$. Also, we tried to find out how large the difference was between the maximum average availability, and the availability when using $\lambda_2$ that minimizes the average response time (note that the formula for availability stays the same when using the slightly changed model when the optimization criterion is the average response time). Further, we wanted to find out the difference between the minimum average response time and the response time given by using the $\lambda_2$ that maximizes the average availability.

Our most valuable insights after deeper studies on the mathematical model are as follows (see also the table below).

The most interesting finding was that for similar request arrival rates, the minimum response time for different failure rates did not present too large differences. Besides, the values were not far from those considered when the system was not exposed to failures at all and no checkpointing was executed. When failures were assumed to occur, but no checkpointing was done, the value of the minimum response time was found to tend rapidly towards very large values. For an infrastructure configured to checkpoint with a rate that would minimize the average response time for a given load, the deviation from the expected minimum response time is larger, the larger the load gets.

The maximum availability decreases when client request arrival rate ($\lambda_1$), respectively failure rate ($\lambda_2$) grows. However, the rate of availability decrease is higher for a higher $\lambda_2$. Similar behaviour is observed when varying $\lambda_1$ and observing the changes in the rate for availability decrease.

There is a value of the request arrival rate from which the value of the maximizing (minimizing) checkpointing interval decreases so much that it becomes much smaller than the average time the checkpointing request spends in the system.

The shape of the maximizing (minimizing) checkpoint arrival rate is independent of failure rate: there is a peak after which the minimizing checkpoint rate starts decreasing. For the maximizing checkpoint arrival rate this is valid for all failure rate studies. However, the minimizing checkpointing rate, for higher failure rates, increases as the request arrival rate grows.
8.1 Numerical studies

<table>
<thead>
<tr>
<th>Highlights</th>
<th>Covered in section(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>significant relations involving the optimized quantities – average response time or average availability – when considered as functions of the average client request rate $\lambda_1$</td>
<td>8.1.1 and 8.1.2</td>
</tr>
<tr>
<td>relations involving the maximizing (minimizing) checkpoint arrival rate $\lambda_2$ considered as function of the request arrival rate $\lambda_1$</td>
<td>8.1.3</td>
</tr>
<tr>
<td>behaviours of the maximizing (minimizing) checkpoint arrival rate $\lambda_2$ as function of the failure rate $\lambda_2$: to reinforce our findings here, this part also contains studies of the maximized, respectively minimized quantities as functions of $\lambda_2$</td>
<td>8.1.5 and 8.1.6</td>
</tr>
<tr>
<td>relations involving the maximizing (minimizing) checkpointing interval as function of the request arrival rate $\lambda_1$</td>
<td>8.1.4</td>
</tr>
<tr>
<td>trade-off availability (response time) by e.g. relating the maximizing and minimizing $\lambda_2$</td>
<td>8.1.7</td>
</tr>
</tbody>
</table>

Table 8.1: Main findings of numerical studies

When studying the maximizing (minimizing) checkpoint arrival rate as function of the failure rate, our main insight was that sometimes there is a value of the failure rate from which on the average more than one failure occurs between two checkpoint arrivals. This value is larger, the larger the request arrival rate gets.

The findings are divided in five main parts covered in the next subsections and summarized in Table 8.1.

8.1.1 Relating response time to request arrival rate

Four studies were conducted about the average response time as a function of average client request arrival rate $\lambda_1$:

1. study of the minimal average response time,
2. study of the average response time when no failures occur (and therefore no checkpointing is needed),
3. study of the average response time when failures occur, but no checkpointing is done,
4. study of the sensitivity of the average response time to load variations.

Valuable insights were gained after the studies. First, some unexpected results were pointed out regarding the average response time (as function of $\lambda_2$) as the system is exposed to different failure rates. Second, we found that when the system is configured to obtain a minimum average response time, the sensitivity of the
average response time to load variations is not highly dependent on the load for which the system was configured.

The graph in Figure 8.1 shows the minimal average response time as function of $\lambda_4$ for four different values of the failure rate $\lambda_f$. The fifth curve on the graph represents the dependence of the average response time on $\lambda_4$ when no failures occur (and thus no checkpointing is executed either). One would expect that the curve for the average response time when no failures occur is much lower than the other four curves. We can see that this curve indeed never goes above the others, but is nevertheless very close to them. The unexpected phenomena depicted in the graph is that the curves are so close to each other for most values of $\lambda_4$, i.e. the minimal average response times for very different average failure rates are almost identical. Also, we can notice that the value of the average response time is almost constant until $\lambda_4$ reaches a relatively high value ($\approx 4$), quite close to the limit $\mu_2 = 5$.

The graph in Figure 8.2(a) presents the case when the system is configured to give the minimum average response time for a given load ($\lambda_4 = 1$ and $\lambda_3 = 2.25$ respectively). Thus, the system is configured to checkpoint with a rate $\lambda_2$ that minimizes the average response time for the respective loads at a given failure rate ($\lambda_f = 0.01$). We see that as the load is increased the average response time does not vary that much (almost constant) for both configurations until the load reaches a limit. We also see that the two configurations are behaving similarly below their respective limits. After the load reaches this limit (for one configuration
Figure 8.2: Load sensitivity of the average response time when system configured for minimal response time

around four times the assumed load, and for the second configuration twice the assumed load) average response time diverges. This is due to increase in failover time, which is now so large that the value of $\mathbb{E}[N_{B\text{upper}}]$ (involving a logarithm) cannot be computed, due to the expression in the logarithm becoming negative.

A similar study was performed, this time varying failure rates. Figure 8.2(b) shows the two curves corresponding to different failure rates. The picture shows, somewhat expected, that if the failure rate is low ($\lambda_f = 0.00001$), it takes a large variation in the load to reach the worst-case average response time (more than twice the assumed load), compared with the case where the failure rate is high, $\lambda_f = 0.01$, where $\lambda_f$ the worst-case average response time is reached earlier.

One would think that something like the study of the 99th percentile response time would give an insight on what are the worst-case response time behaviours of a system, given the (minimal) average response time. These figures can easily be approximated (by an upper bound), by using Markov’s inequality [3] (in $P[X \geq t] \leq \frac{E[X]}{t}$, if we set $P[X \geq t] = 1 - 0.99 = 0.01$, we find that $t$ – the 99th percentile of the random variable $X$ – is upper-bounded by $100 \times E[X]$). Therefore we believe that they are too pessimistic to be useful in analysis of distributed
service-oriented architectures.

Figure 8.3 shows the behaviour of the average response time when failures occur, but no checkpointing is executed. In this situation, the call log size is not reduced from time to time. In fact, the size of the log can grow unbounded. As the graph illustrates, the average response time grows to infinity as the average client request arrival rate grows. Figures 8.3(a) and 8.3(b) show that the extent of growth is larger, the lower the failure rate gets.

8.1.2 Relating availability to request arrival rate

Next, we study maximum average availability in relation to $\lambda_f$ for different failure rates. By availability % (y axis in Figure 8.4(a) and in Figure 8.4(b)) we mean the normalized value of the maximum average availability between two consecutive failures ($\max(\mathbb{E}[T_A])$). The normalization is done by multiplying the absolute value of the average maximum availability by the failure rate times 100. This roughly reflects the total time that the server is up over an extended time period, and helps in comparing different failure rates. An example: assume that, with failure rate=0.09 one obtains the value 11 for maximum $\mathbb{E}[T_A]$, while with failure rate=0.01 one obtains the value 89 for maximum $\mathbb{E}[T_A]$. In both cases all other
Figure 8.4: Maximal average availability % (y axis) plotted against average client request rate $\lambda_1$ (x axis)

parameters are the same. At first sight $89 > 11$, but by looking closer, one sees that $11$ is out of $\frac{1}{0.09} \approx 11.1$, while $89$ is out of $\frac{1}{0.01} = 100$.

Figures 8.4(a) and 8.4(b) show that the behaviour of maximum average availability in relation to $\lambda_1$ is similar for all chosen failure rates: it decreases, as $\lambda_1$ grows. However, for $\lambda_f = 0.001$ and $\lambda_f = 0.00001$ the decrease happens with a much smaller slope than for the larger $\lambda_f$'s.

### 8.1.3 Relating checkpoint arrival rate to request arrival rate

This section will present the checkpointing rate $\lambda_2$ that maximizes the average availability ($E[T_A]$) as a function of $\lambda_1$. Also, it will discuss the $\lambda_2$ that minimizes the average response time ($E[R_T]$) and the average backlog processing time ($E[B]$), as functions of $\lambda_1$. The latter study is interesting as the average backlog processing time ($E[B]$) forms part of the expression of $E[R_T]$. One may ask why it is interesting to study these quantities as functions of $\lambda_1$? The answer is that we wanted to find out how increasing average client request arrival rates are handled by the system in presence of checkpointing requests and failures, and how the average checkpoint rate should behave to obtain maximal availability and minimal response time, respectively. The next question is how the behaviour of these quantities relate to a chosen $\lambda_f$ (failure pattern)? Would we get similar curves no matter what/how small
Chapter 8. Analysis of the optimization models

Figure 8.5: \( \lambda_2 \) (y axis) plotted against \( \lambda_1 \) (x axis)

is the average failure rate?

We discovered two interesting phenomena: the optimizing average checkpoint arrival rate \( \lambda_2 \) presents similar behaviour independent of the optimization criteria and of failure rates.

The graph in Figure 8.5(a) shows the behaviour of the value of \( \lambda_2 \) that maximizes the average availability, when considered as function of \( \lambda_1 \). Independent of the failure rate, the value of \( \lambda_2 \) grows until a point where it starts falling. The variation and the magnitude of \( \lambda_2 \) is different from one failure rate to another. For example, when \( \lambda_f = 0.001 \), the variation is approximately between 0 – 0.025; for \( \lambda_f = 0.09 \) the variation is between 0 – 0.24.

The graph in Figure 8.5(b) presents the variation of the value of \( \lambda_2 \) that minimizes the average response time, when considered as a function of \( \lambda_1 \). We studied the influence of the value of \( \lambda_f \) on the shape of the obtained curves. We found that the behaviour of \( \lambda_2 \) is approximately independent of failure rate. Like for \( \lambda_2 \) that maximizes availability, until a certain value of \( \lambda_1 \) (\( \approx 2.75 \)), the value of \( \lambda_2 \) increases and then it starts decreasing. The variation and the magnitude of \( \lambda_2 \) is different from one failure rate to another. For \( \lambda_f = 0.00001 \), the variation is between 0 – 0.025; for \( \lambda_f = 0.01 \) the variation is between 0.025 – 0.15. For quite high failure rates (\( \lambda_f = 0.01, \lambda_f = 0.09 \)) the value of \( \lambda_2 \) is almost constantly increasing. Thus the magnitude and the variation are both larger.

As shown by the graphs in figures 8.5(a) and 8.5(b) there are two situations...
when the lowest values for $\lambda_2$ are obtained:

- when the average arrival rate of client requests ($\lambda_1$) is low, i.e. there are low service demands from the client; in such a case, checkpointing must not be done so often, since at failover, the average number of calls to replay will not be very large

- when the average client request arrival rate ($\lambda_2$) is high, i.e. client requests arrive more densely; in such a case, checkpointing must not be done often, because it interferes too much with incoming calls. In other words, the checkpointing request has to wait, on the average, a lot in the system before the state is recorded.

Next we study the value of $\lambda_2$ that minimizes the average response time compared to the one that minimizes the average backlog processing time. We would expect the two values to be close to each other, as the reduction of the average backlog processing time would imply lowering the average response time as well. Figure 8.6 presents, however, a different picture (typical for all failure rates studied): independent of the failure rate the value of $\lambda_2$ that minimizes the average backlog processing time ($\mathbb{E}[\mathcal{B}]$) is significantly larger than the value that minimizes the average response time ($\mathbb{E}[R_T]$). We can notice however, that the behaviour of $\lambda_2$ that minimizes $\mathcal{B}$ is also increasing-decreasing, plotting approximately the function $\frac{\mu_2}{\mathbb{E}[\mathcal{T_c}]} - |\frac{\mu_2}{\mathbb{E}[\mathcal{T_c}]} - x|$.

This shows that to minimize the average backlog processing time, the system would want to do as much checkpointing as allowed by the constraints $\lambda_2 + \lambda_1 < \mu_2$ and $\lambda_2 \leq \lambda_3$. The explanation for this tendency is that the average backlog processing time is actually only determined by the average failover time, that is in turn reduced by frequent checkpointing.

### 8.1.4 Relating checkpointing interval to request arrival rate

This section presents our findings about the behaviour of the maximizing (minimizing) checkpointing interval ($p$) as a function of $\lambda_1$ in relation with the average time the checkpoint request spends in the system ($\mathbb{E}[\mathcal{T_c}]$). The main insight we gained was that there (sometimes) exists a value of $\lambda_1$ from which the inequality $p > \mathbb{E}[\mathcal{T_c}]$ no longer holds. $\lambda_1$ (the turning point) for which this happens, is the maximal average rate of incoming client requests, which the two servers with the given average service rates can handle. In practice, the relation $p < \mathbb{E}[\mathcal{T_c}]$ means...
that, on the average, the checkpointing request stays in the system (the queues of the two servers and the servers themselves) more than the time that passes from when the state gets stored until the next checkpointing request arrives. Note that the “no infinite queue” assumption can still hold, even when $p < E[T_c]$.

Interestingly enough, we found that the relation between the two quantities is similar, independent of the optimization criteria we use to obtain $p$. Moreover, after studying the relation for different failure rates $\lambda_\varepsilon$ and different call replay rates $\mu_3$ we discovered even how these parameters influence the exact position of the turning point. In what follows we will show a set of curves corresponding to the case where the optimization criterion was the average availability. The same conclusions hold for the studies of the response time as well.

Figure 8.7 shows the relation between the maximizing checkpointing interval $p$ and the average client request rate $\lambda_1$ given a fixed failure rate $\lambda_\varepsilon$. The effect of the variations in the average call replay time are shown by changing $\mu_3$ (average call replay rate) and keeping everything else constant, see parts (a) and (b) of Figure 8.7. We can conclude, somewhat expected, that as the call replay rate decreases the turning point moves to the left, with the possibility of disappearing when the call replay rate grows.

The effect in the decrease in failure rate on the behaviour of the two quantities – the maximizing $p$ and $E[T_c]$ – was studied by varying the failure rate and keeping everything else constant. The result of this study can be seen by comparing part
8.1 Numerical studies

Figure 8.7: Maximizing $p$ and $E[T_c]$ (y axis) plotted against $\lambda_1$ (x axis) when $\lambda_f = 0.09$

(a) $\mu_3 = 12$

(b) $\mu_3 = 6$

(a) of figures 8.7 and 8.8 and also part (b) of the same figures. The graphs lead us to the result that when the failure rate decreases, the turning point is moved to the right.

The interpretation of the decrease of optimizing $p$ as seen in the diagrams is as follows: when $\lambda_1$ goes over a certain value, i.e. the average rate of incoming requests is higher than that value, checkpointing must happen very often, to reduce the failover time. If checkpointing were not done so often, then, at failover, there would be more calls in the log to replay. However, depending on the value of the call replay rate $\mu_3$, the request arrival rate $\lambda_1$ that can be handled by the system while still preserving the relation $p > E[T_c]$ varies. In the studies illustrated above the value of $\lambda_1$ at the point where the relation between optimizing $p$ and average checkpoint time $E[T_c]$ changes, is lower in 8.7(b)(8.8(b)) compared to 8.7(a) (8.8(a)). The reason is that in the first case, the average time to replay a request is longer ($\frac{1}{8} = 0.125$ time units in 8.7(b) as compared with $\frac{1}{12} = 0.0833$ time units in 8.7(a)).
Chapter 8. Analysis of the optimization models

Figure 8.8: Maximizing $p$ and $E[T_c]$ (y axis) plotted against $\lambda_1$ (x axis) when $\lambda_f = 0.01$

8.1.5 Availability maximization and failure rate

Next we try to answer the question: from which failure rate (value of $\lambda_f$) the assumption about the maximizing $\lambda_2$ and $\lambda_f$ ($\lambda_2 > \lambda_f$) ceases to hold? Our studies in this section shed light on this question and at the same time give new insights about the relationship between $\lambda_1$ (average client request arrival rate), $\lambda_f$ and the maximizing $\lambda_2$.

We choose two arbitrary values for $\lambda_1$ (1 and 4) and study the variation with the failure rate in three contexts:

- trade-off between checkpoint arrival rate and failure rate (Figure 8.9)
- trade-off between number of calls to replay and failure rate (Figure 8.10(a))
- trade-off between maximum availability and failure rate (Figure 8.10(b))

The rationale behind choosing the two values for $\lambda_1$ was that we wanted to have one relatively small value as compared with the average service rate of server 2 (1 compared to 5) and one relatively high (4 compared to 5).
Finally, we consider the effect on the maximum availability when varying $\lambda_{\text{f}}$ (Figure 8.10(b)) that the descriptions below show is closely related to the maximizing $\lambda_{\text{f}}$. Our observations, for $\lambda_{\text{f}} = 1$ followed by $\lambda_{\text{f}} = 4$ are given below:

- when $\lambda_{\text{f}} = 1$ (smaller average client arrival rate) we note that as soon as failures become more frequent, $\lambda_{\text{f}} > 0.09$, the desired relation $\lambda_{\text{f}} > \lambda_{\text{f}}$ does no longer hold (see Figure 8.9(a)). This means that, on average, more than one failure can occur between two checkpoints. Further, for high failure rates (not far from $\lambda_{\text{f}}$), the value of $\lambda_{\text{f}}$ becomes very small, tending toward zero. This means that we could perhaps avoid checkpointing altogether in order to obtain maximum availability. Since client request arrivals are relatively sparse ($\lambda_{\text{f}} = 1$, as compared e.g. with a service rate on the Application server, $\mu_{2} = 5$), the tendency towards smaller $\lambda_{\text{f}}$ in Figure 8.9(a) is not to avoid interference by reducing checkpointing. Rather, it shows that failover time does not grow as much, even if checkpointing is done very seldom. To clearly illustrate this, Figure 8.10(a) shows the relation between the failure rate ($\lambda_{\text{f}}$) and the average number of calls that are found in the log (to be replayed) at failover time. It can be seen that as failures come more often, this number decreases (not due to checkpointing, but rather due to the low rate of incoming calls!). Figure 8.10(b) shows that for $\lambda_{\text{f}} = 1$, the normalized
availability of the system decreases with higher failure rates, but it decreases at a slow rate. Thus, the overhead added by checkpointing more often might be higher than the cost of replaying more calls.

- when $\lambda_1 = 4$ (a value closer to $\mu_2 = 5$) we note that the availability of the system tends towards very small values ($< 30\%$) at higher failure rates (see Figure 8.10(b)). Besides, the desired relation $\lambda_f < \lambda_2$ starts not holding much later here (when $\lambda_f > 0.25$) as shown in Figure 8.9(b). Moreover, the optimal average checkpoint arrival rate starts decreasing towards 0 at much higher failure rates (compared to when $\lambda_1 = 1$), meaning that checkpointing is always necessary. Here, due to the higher rate of incoming calls, with no checkpointing the failover time would be much larger. Figure 8.10(a) (the upper curve) shows that at highest failure rates the number of calls to replay does not drop below 9 (which is roughly ten times as high as the case with $\lambda_1 = 1$), while, as Figure 8.10(b) shows, the availability decreases rapidly towards values $< 30\%$.

The next section will answer similar questions about the relation between $\lambda_2$ and $\lambda_f$ in the context of minimizing the average response time. Surprisingly, it will be noticed that the corresponding graphs look similar.
8.1 Numerical studies

8.1.6 Response time minimization and failure rate

Our studies about the behaviour of the minimizing $\lambda_2$ offered insights about the value of $\lambda_2$ from which the relation $\lambda_2 < \lambda_2$ no longer holds. To reinforce our findings we also studied the minimal $E[R_T]$ as function of $\lambda_2$ (Figure 8.12).

Figure 8.11 shows how the value of $\lambda_2$ behaves as the frequency of failures grows, and also how the way it varies is influenced by the average client request arrival rate $\lambda_3$. We considered two values for $\lambda_3$: 1 and 4.

A summary of our insights will be given below:

- the value of $\lambda_2$ from which the minimizing checkpoint arrival rate $\lambda_2$ becomes smaller than the average failure arrival rate, grows as $\lambda_3$ increases. This can be seen in Figure 8.11 parts (a) and (b).

In a situation like in Figure 8.11(b) when $\lambda_3 = 4$ (close to the limit $\mu_2 = 5$) it is always the case that $\lambda_2 < \lambda_2$. This latter situation is what we would like to have.

- for low values of the client request arrival rate, the minimizing $\lambda_2$ tends towards zero, as $\lambda_3$ increases (see Figure 8.11 for $\lambda_3 = 1$). The explanation could be that as the average client request arrival rate is rather low here, the failover time does not grow so much, since the number of requests to replay is low. Somewhat unexpected, the average response time increases for lower failure rates, and decreases as the failure rate grows (see Figure 8.12). The latter is most likely due to the fact that as the failure rate grows, the average distance between the failure occurrence moment and the moment of the nearest state recording, i.e. $E[Y]$ shrinks and thus the average failover time decreases.

8.1.7 Average availability-average response time trade-offs

In the last part of the numerical studies we will describe our findings when investigating how, for example the value of $\lambda_2$ that maximizes $E[T_k]$ relates to the value of $\lambda_2$ that minimizes $E[R_T]$. Further, results of the main comparison of the two optimizations will be revealed: how does the average availability of the system when minimizing average response time relate to the maximum average availability. Of course, we expected that the optimized maximum availability would be higher than the availability obtained from optimized response time. What we did
not know before the studies was how large the difference between the two values was. Analogously, insights will be presented about the relation between the optimized average response time and the average response time obtained when maximizing availability. All studies were conducted when considering the mentioned quantities as functions of $\lambda_1$. 

Figure 8.11: Minimizing $\lambda_2$ (y axis) plotted against $\lambda_f$ (x axis)

Figure 8.12: Minimal $\mathbb{E}[R_T]$ (y axis) plotted against $\lambda_f$ (x axis) for $\lambda_1 = 1$
Figure 8.13: \( \lambda_2 \) that minimizes \( E[R_T] \) and that maximizes \( E[T_A] \) (y axis) plotted against \( \lambda_1 \) (x axis)

Figure 8.13 presents comparatively the variations of the two checkpoint rates \( \lambda_2 \)'s (the one minimizing \( E[R_T] \) and the one maximizing \( E[T_A] \)) as functions of \( \lambda_1 \) for different (low and high) values of \( \lambda_f \). We can see that the tendency is similar irrespective of whether the system is exposed to low or high failure rates: to obtain minimum response time, the rate of checkpointing has to be higher than to obtain maximum availability, given identical external conditions (same \( \lambda_1 \) and same \( \lambda_f \)).

To further find out how significant the difference in \( \lambda_2 \)'s is, we examined the quantities \( E[T_A] \) and \( E[R_T] \) as follows:

1. Availability: comparing maximal \( E[T_A] \) and \( E[R_T] \) obtained for \( \lambda_2 \) that minimizes \( E[R_T] \) and

2. Response time: comparing minimal \( E[R_T] \) and \( E[R_T] \) obtained for \( \lambda_2 \) that maximizes \( E[T_A] \)

Interestingly enough we found that the two values from 1, respectively from 2, are not so far from each other.

For \( E[T_A] \) this is mostly visible when the failure rate is relatively high. Figures 8.14(a) and 8.14(b) show that the value of average system availability when optimizing the average checkpoint arrival rate to obtain minimal average response
time, is not much smaller than the maximum average availability. Even for low failure rates, the difference is not large if we consider that the whole variation of $E[T_A]$ while $\lambda_1$ varies from small values to the largest ones is not larger than 1% (see figures 8.15(a) and 8.15(b)).

We can therefore conclude that, especially for high failure rates, if the checkpointing interval is such that the average response time is thereby minimized, the availability of the system is also high, close to its maximum.

For $E[R_T]$ the difference is really small both for low failure rates and high failure rates as shown in figures 8.16(a) and 8.16(b). This again suggests that the average response time for a request does not get much worse when we adjust the checkpointing interval to make the system maximally available for request processing.

### 8.2 Simulations

This section will present the results of our simulations conducted to validate the basic mathematical model. When building the mathematical model we approximated the checkpoint interarrival time with an exponentially distributed random variable. This assumption was needed to make possible the computation of the maximizing checkpointing interval $p$ based on the (random variable) inputs. As a conse-

![Figure 8.14: Behaviour of maximal $E[T_A]$ and $E[T_A]$ with $\lambda_2$ that minimizes $E[R_T]$, for high failure rates](image)
8.2 Simulations

The main goal of this section is to check how significant this approximation in the mathematical model is. That is, are all the observations in Section 8.1 valid for the real world? In particular, if an engineer takes the result of the computations (the computed maximizing $p$ based on a set of fixed input parameters) does this indeed produce maximum availability or not? This section will show a validation of this approximation in the mathematical model. Besides, simulations are performed to validate the mathematical model itself.

8.2.1 Validation approach

Two simulation models were built within the SIMULA programming environment each with a different purpose. SIMULA provides a discrete event scheduling of the arrival of requests, servicing of the queues, etc [160]. The first simulation model designated to validate the mathematical model, was built using all the assumptions from the mathematical model. We ran simulations with parameters ($\lambda_1, \mu_1, \mu_2, \mu_3, \lambda_2, s$) exactly like in the numerical studies from Section 8.1.2 and studied the average availability as a function of $\lambda_1$. For a combination of those values (e.g. $$\lambda_1 = 0.00001$$ and $$\lambda_1 = 0.001$$)

![Figure 8.15: Behaviour of maximal $E[T_A]$ and $E[T_A]$ with $\lambda_2$ that minimizes $E[R_T]$, for low failure rates](image)

(a) $\lambda_1 = 0.00001$  
(b) $\lambda_1 = 0.001$

Figure 8.15: Behaviour of maximal $E[T_A]$ and $E[T_A]$ with $\lambda_2$ that minimizes $E[R_T]$, for low failure rates
Chapter 8. Analysis of the optimization models

Figure 8.16: Minimal $E[R_T]$ and $E[R_T']$ with $\lambda_2$ that maximizes $E[T_A]$ (y axis) plotted against $\lambda_1$ (x axis).

$\lambda_1 = 0.75$, $\mu_1 = 50$, $\mu_2 = 5$, $\mu_3 = 12$, $\lambda_4 = 0.01$ and $s = 0$, eight values for the average checkpoint arrival rate were considered in turn: one equal to the value that maximizes $E[T_A]$ obtained from the analytical model, while the other seven corresponding to values of the average checkpoint interarrival time ($\frac{T_A}{\lambda}$) equal to 25%, 50%, 75%, respectively 125%, 150%, 175%, 200% of the value that maximizes $E[T_A]$. These values for $\lambda_2$ were needed, on one hand, to confirm that the simulated value is close enough to the computed value of maximal $E[T_A]$ for the same values of parameters. On the other hand, we wanted to show that the simulated maximum availability is indeed the maximum, compared to the simulated availabilities obtained for values of $\frac{T_A}{\lambda}$ larger and smaller than the maximizing one.

The second simulation model was used to validate the approximation within our mathematical model. It is based on the same assumptions as in the mathematical model except the one related to the probability distribution of the checkpointing interarrival time random variable. In the simulations each checkpointing request is scheduled exactly $p$ units after the previous one ends, modelling a deterministic $P$. We considered eight values for $p$ for each value of $\lambda_1$: one corresponding to the maximizing value of $p$ obtained from our mathematical analysis, while the other seven corresponding to values equal with 25%, 50%, and 75%, respectively with 125%, 150%, 175%, 200% out of the maximizing value.
To check whether the simulation runs confirm our intuitive expectations concerning the approximation of the checkpoint interarrival, we compared the following sets of simulations:

- the simulated system availability (as function of the average client request arrival rate $\lambda_2$) using a deterministic value of $p$ equal to the computed optimal value from the Mathematica model.

- the system availability (as function of $\lambda_1$) using a range of other values for $p$, both below and above the $p$ that maximizes $E[T_A]$.

To confirm the validity of the model we would expect that for all choices of checkpointing interval when $p$ was different from our maximization result, i.e. both below and above the $p$ that maximizes $E[T_A]$, the simulations would show lower availability compared to the availability with the maximizing $p$. Hence, the approximation in the mathematical model would be shown to lead to a valid result in the (simulated) reality. These will provide evidence that if our other assumption are valid then we can expect that the mathematical studies lead to results applicable in reality.

To repeat the studies in the same framework as the mathematical analysis, we began by choosing failure rate $\lambda_f = 0.01$ and performed a set of 100 experiments for each value of $p$ (respectively $\lambda_2$) - eight such values were considered respectively, and for each selected $\lambda_1$. In case of $\lambda_f = 0.01$ we chose 18 values for $\lambda_1$. Thus, in total we ran $(8 \times 18) \times 100$ experiments. Each experiment consisted of a simulation run lasting between 1000 – 5000 logical time units. For each experiment we computed the time the servers are not available for request processing. This is the sum of time intervals while the checkpointing request is processed on the Logging server (for marking the call records that have to be deleted, and for storing the state) and Application server, as well as time intervals when the system performs failover. This sum was subtracted from the total simulation logical time (e.g. 1000) essentially computing the average value for $T_A$. For each of the 100 experiments the availability percentage was computed by dividing this value ($E[T_A]$) by the length of the simulation interval (e.g. 1000). Finally, for each value of $p$ (respectively $\lambda_2$) and of $\lambda_1$ the average of the 100 availability percentages was obtained.
8.2.2 Validation results

This section will present the results of the analysis of the two simulation models. First, the model designated to validate the mathematical model will be analyzed. Then, we will show results concerning the validation of the approximation we made about the arrival of checkpointing requests.

The diagram in Figure 8.17 shows that for \( \lambda_f = 0.01 \) our mathematical model is validated as the value of the average availability obtained in simulations is within an (acceptable) distance of \( \approx 5\% \) from the value of the maximum \( \mathbb{E}[T_A] \). Figure 8.18 further illustrates this, as the (simulated) average availability for the maximizing \( \lambda_2 \) is indeed the maximum in comparison with values of the average availability obtained for other seven larger or smaller values of \( \lambda_2 \).

Figures 8.19(a), 8.19(b) as well as 8.20(a) and 8.20(b) present evidence leading to the conclusion that our approximation about the interarrival of checkpointing requests is validated as well.

The diagram in Figure 8.19(a) shows how the availability of the system varied with \( \lambda_2 \) when using the maximizing \( p \) (as obtained after the maximization of \( \mathbb{E}[T_A] \) performed in Mathematica) and other values of \( p \) (obtained by decreasing the maximizing \( p \) with 25\%, 50\%, and 75\% of its value respectively). Note that for all values of \( p \) different from the optimal value, the availability of the system is lower than for \( p \) equal to the optimal \( p \). Remember that decreasing \( p \) means increasing the frequency of checkpointing. Since the availability of the system decreases with
8.2 Simulations

Figure 8.19: Simulated average availability % (y axis) plotted against $\lambda_1$ (x axis) when $\lambda_f = 0.01$

lower values of $p$, this part of the simulation study confirms our intuitions.

Figure 8.19(b), on the other hand, shows that when increasing $p$ from optimal $p$ (with 25%, 50%, 75%, respectively 100% of its value) the availability of the system is not much different from runs with period equal to optimal $p$. Moreover, some simulation runs provide availability slightly above the supposed optimal availability. One could argue that where the sign is different (e.g. higher availability with $1.25p$) the fluctuation is acceptable if the size of the difference is small.

To confirm that sizes of the differences in availability given the chosen $p$s is within reasonable limits, we plotted the same graphs (availability against $\lambda_3$ when $\lambda_f = 0.01$), but using our mathematical model. Figure 8.20(b) shows that we have similar increments in availability as the $p$ is reduced when starting from values higher than optimal $p$. The values for availability percentage in the two pairs of Figures are thus quite similar.

All simulations of this section were repeated for $\lambda_f = 0.001$ and $\lambda_f = 0.09$. The validity of our approximation from the mathematical model, as well as the mathematical model itself was exhibited again with similar results as in the curves already presented in this section.

Performing validations of the mathematical model by using the realistic tele-
Final remark on serialization

In chapters 7 and 8 we presented results based on modelling the queuing of the checkpointing request in the queues of the two servers (Logging and Application), having the same priority as the regular client requests. This obviously led to a load (client request arrival rate) dependent behaviour of the time spent by the checkpointing request in the system, i.e. from entering the queue of the logging server, until exiting the system after recording the state. Also, the behaviour of the checkpointing rate that maximizes (minimizes) the quantity considered for optimization, considered as function of the load, was such that it presented low values for both high and low loads.

One could consider modelling the situation when the checkpoint request has higher priority than regular client requests, and thus studying nonpreemptive priority queues. The idea is that the checkpoint request, as soon as it arrives in the queue of e.g. server 1 (both when logging its call information, and when recording the state), waits for the currently executing request on that server, after which it

(a) checkpointing more often than with the maximizing p

(b) checkpointing less often than with the maximizing p

Figure 8.20: Average availability % (y axis) plotted against $\lambda_1$ (x axis) when $\lambda_r = 0.01$

com application provided by Ericsson AB, is subject for future work.
immediately receives service.

We constructed new formulas taking account of the above change and for the same optimization criteria we made some observations. No significant changes occurred to the behaviours of the optimizing checkpointing rate, or the optimized quantities. Regarding the average availability as optimization criterion, we found a (somewhat expected) change in the behaviour of the maximizing average checkpoint rate. Its tendency is to grow as the client request arrival rate grows, even for high failure rates. This is explained by the fact that in this model the checkpoint time is not load dependent, and thus the only effect of growing load is to increase the average number of calls to replay, and the only way to reduce this is to perform checkpointing more often. The changes to the maximum availability as function of load or failure rates were insignificant.

Regarding the average response time as optimization criterion, there were insignificant changes to the behaviour of the minimizing checkpoint rate. On the other hand quite significant changes were noticed in the behaviour of the minimum response time with the growth of the load: for high values of the client request arrival rate (cf. the service rate of the application server), the minimum response time is lower than in the case where no priority of the checkpointing is used.
Chapter 9

Conclusions and future work

The work presented in this thesis comprises of two main parts:

- empirical studies with implementations of CORBA enhanced with support for fault tolerance, and

- mathematical studies revealing the optimal checkpointing interval to be incorporated in a fault-tolerant middleware with support for passive replication. The latter point was studied with two different optimization criteria:
  - maximum average availability
  - minimum average response time

The driving force and the glue in this work was to offer an application writer easier ways to develop distributed fault-tolerant applications and to configure them for best performance.

The first step was to build a middleware that can transparently handle fault tolerance related operations, below the application level. Two infrastructures were built: one of them (called FT-CORBA) deals only with failures of the application. The other (called FA-CORBA) uniformly handles failures of the application as well as the infrastructure, hence labelled fully available. Obviously, the application writer would gain a lot by using these services, needing only to concentrate on the functionality of the application. However, he/she might have doubts about whether it is appealing to use such infrastructure, considering factors such as performance overhead and gains in failover time. To overcome possible hesitations with respect to performance overheads in absence of failures, we conducted
availability-performance trade-off studies by performing measurements on a tele-
com application running on top of the middlewares. We were aware of the fact that
it is difficult to generalize our findings based on only one application. However,
all the earlier published studies on the support for fault tolerance based on CORBA
had used synthetic experiments for evaluation. We believe that this work is at least a
step in the right direction. This reservation is important since also the performance
related insights are obviously affected by the ORB implementation over which the
platforms were built. Again, we believe that the wisdom from relative compar-
ison of the different replication styles carries over to other ORBs. A different ORB
is expected to increase/decrease the overheads in all replication styles by similar
(relative) proportions.

After the performance studies, as in some cases the overhead of request roundtrip
time using middleware support was large, some way of reducin
g it had to be stud-
ed. The solution we found implied lifting some middleware level mechanism to the
application level, thereby changing the application code. Here we were careful not
to make optimizations that would resemble putting FT support directly in the appli-
cation. The goal was to keep a minimal (and automated) change of the application.
We found that a possible technique to use was aspect-oriented programming. We
adopted AOP as an elegant way to extend application code with application knowl-
edge that affects the fault tolerance mechanisms’ performance. We do not exclude
the possibility of existence of other solutions, but we found it interesting to apply a
new technique like AOP in such a context.

Even after offering the application writer our results of empirical studies, doubts
can still arise. This is due to the sometimes complex relation between the results
and different infrastructure parameters that makes it impossible to configure the
system only based on such empirical results, e.g. optimizing different performance
metrics. To offer help to the application developer in this direction, we consid-
ered one infrastructure parameter: the checkpointing interval. The second part of
the thesis presented mathematical analysis underlying the provision of optima that
lead to maximal average availability and minimal average request response time,
respectively.

After this short summary and motivation, we will present our conclusions re-
arding all the parts.
9.1 Concluding remarks

Our conclusions can be grouped in three parts. These are given by the experience with the building and evaluation of the FT-CORBA and FA-CORBA platforms, the extension involving aspect-oriented programming, and the formalization of the checkpointing interval optimization. We will begin with remarks about the empirical studies involving the two middleware platforms.

Varying the number of replicas has generated conclusions that are likely to carry over to other middleware. The number of replicas made a significant difference only for active replication and the FA styles. For the FA-CORBA platform, the group size made a difference in both overheads and failover times. This was due to the need for executing the consensus primitive in both failure free and failover scenarios. The checkpointing interval as well as the corresponding measure in the FA-CORBA platform, the pruning interval, influenced the two metrics. When transforming the FT-CORBA middleware to support aspect-oriented programming based extension of the application code, the number of replicas did not change the results so much. This was not surprising, as the replication style reproduced in this setting was only the passive style. The checkpointing interval, on the other hand, influenced the overheads to a certain extent. Still, a clear relation between the checkpointing interval parameter and the overheads was not possible to obtain from empirical studies.

To sum up, providing fault tolerance embedded in middleware, and keeping transparency for application writers involves a price. This price is the extra delay experienced by a client while receiving a reply from a replicated server. The price becomes higher in an infrastructure in which failures of non-application units are handled together with those of application objects (employing a distributed algorithm).

The experience with building the FT-CORBA platform shows that it is feasible to implement the FT-CORBA standard by extending an existing ORB. On the other hand, with no other failure handling mechanisms, except the application object related ones, the infrastructure units have to run as processes that do not fail. Ad hoc replication might be employed to cope with the problem. Still, the rigid failure detection mechanism is inherent to this platform: timers are explicit and visible at a very high level. In contrast, the FA-CORBA platform ensures that failures of infrastructure units do not impair the availability of the application; of course, at the cost of a higher overhead. Besides the higher overheads, there is another drawback in this approach: a majority of replicated units must be up at all times.
This requires some initial hardware investment beyond a redundant pair. Note that if the group has two replicas, none of them may fail! In the FT-CORBA platform, no such restrictions exist. At the extreme (although not really desirable) the system could run with only one replica. Thus, e.g., seven out of eight replicas may fail, and the application will still not block, as opposed to the case in FA-CORBA.

The failover comparisons show that active replication in the FT-CORBA platform is unlikely to be worthy of attention due to its high overheads. In our experiments, the extremely short failover time (70ms) cannot be considered essential in a high-assurance system if the no-failure roundtrip time is around 6 seconds (after adding the overheads). Another limiting factor might be the need to ensure replica determinism. In such a competition, the FA-CORBA platform will win, since non-determinism of replicas is tolerated. In both cases, the available resources further influences the decision, since the number of replicas in the group gives the maximum number of failures tolerated.

When looking at overheads, the two passive replication styles in FT-CORBA score well: no need for determinism, and failover is in range of seconds. Warm passive replication does not exhibit larger overheads than the cold passive style, but gives a shorter failover time, especially if the checkpointing interval is low. The bottom line is that the quest for higher availability lies in the likelihood of failure in the infrastructure units. As long as the costs for hardening such units is not prohibitive, the FT-CORBA platform may be considered as adequate since the overheads are smaller and the failover times have comparable values.

What provides more reasons to recommend the FT-CORBA infrastructure is the successful incorporation of aspect-oriented programming in connection with the transparent extension of application code, to improve performance. The success of using aspects should be seen in the modification of the application code with no programmer intervention leading to a gain in performance of server request processing. In some cases, as shown in our performance studies, the overhead percentage dropped to half of the overhead rate in the regular FT-CORBA implementation. As mentioned before, there is a further potential for reducing the application writer input to only writing application code (i.e. no added effort due to FT and aspect-orientation support requirements). This could be implemented if standard techniques for static analysis are used to deduce the variable usage in methods instead of the programmer writing the component description. In the context of modifying the application code by weaving aspect code, security problems can be considered. However, in the present setting, we assume that the middleware together with the aspect generating software is provided by a trusted party. Also,
9.1 Concluding remarks

the application writer who augments his/her code with the aspects generated based on the component description does not impair security more than he/she would if no external code would be weaved in the application. Thus, the new usage of the (aspect-supporting) FT-CORBA middleware will presumably not impair the security of the application more than the original FT-CORBA infrastructure.

After performing the large series of empirical studies with the realistic application on top of the built middleware, we found it important to provide a formal framework to the application developer. Among the parameters in the infrastructure that could be varied to obtain best performance were the number of replicas, the checkpointing interval, and even the replication style. We chose to study ways of choosing the checkpointing interval such that some quantity is optimized, as indicated by the earlier literature. Our work in this direction is, to our knowledge, the first in the setting with a separate unit for logging calls and replies. We analyzed two mathematical models of a server replicated using the primary-backup mechanism and employing logging of client calls. The basic model is used in the computations of the checkpointing interval that maximizes the average availability of the system. A more refined model is used to find out the formula of the checkpointing interval that leads to minimal average response time of requests. We used queuing theory to build the models, based on assumptions about client request arrival rates, failure rates and other system parameters. We provided the formula for availability, respectively response time as functions of average checkpoint arrival rate \( \lambda_2 \). The checkpointing interval \( \rho \) is also a function of \( \lambda_2 \). Further, when inputting numerical values for the remaining system parameters, any numerical tool (e.g. Mathematica) can be used to maximize system availability or minimize average response time, and obtain a value for the maximizing, respectively the minimizing \( \lambda_2 \). The value of the maximizing (minimizing) \( \rho \) is then obtained using the calculated \( \lambda_2 \).

Our observations revealed significant relations between different parameters in the replicated server system. As expected, the maximum availability decreases when client request arrival rate \( \lambda_1 \), respectively failure rate \( \lambda_\ell \) grows. However, the slope of availability decrease is higher for a higher \( \lambda_\ell \). Similar behaviour is observed when varying \( \lambda_1 \) and observing the changes in the slope for availability decrease. Surprisingly, for low \( \lambda_1 \) and high failure rates, one is better off avoiding checkpointing to obtain maximum availability. On the other hand, when \( \lambda_1 \) is high, getting close to the application service rate \( \mu_2 \), checkpointing is always necessary. This is because for high \( \lambda_1 \), and growing failure rates, the main impairment to availability is the time spent for replaying requests during failover. Here, check-
pointing has to be done in order to cope with the rapid increase of the number of call records in the log.

Similar statements are valid when the optimization criterion is the average response time. Still, surprisingly, the value of the minimal average response time is not much different for different failure rates and same client request arrival rate. Also unexpected was the comparison of the average checkpoint rate (and thus checkpointing interval) that minimizes the average backlog processing time with the \( \lambda_2 \) that minimizes response time. The difference between the two values (for similar client request arrival rate and failure rate) was found significant. Finally, other surprises were found when studying minimal response time versus maximal availability. We found that the difference between the values of the minimal response time (for given request arrival rate and failure rate) and the response time obtained when checkpointing with an average rate that leads to maximal availability (under similar given external conditions) is not at all significant. Also, the difference between the availability of the system when checkpointing with a rate to minimize response time is not far from the maximal availability obtainable for given the client request rate and failure rate.

The whole study has illuminated and deepened our understanding of the trade-offs involved between availability and performance. Still, a lot of work remains to be done and a lot of new directions are still open for exploration.

9.2 Future work

The areas covered in this thesis are pretty large and diverse and open different directions for future work. An extension of the work with the FT handling middlewares is to choose some other technology than CORBA. This change could lead to studies about the extent to which the gathered experience with CORBA and fault tolerance can be applied to other middlewares. Some steps in this direction has started in a new European project (DeDiSys). Besides, the base ideas of the DeDiSys project provide an opportunity to study new configurations of middleware infrastructure. Thus, if until now, in both the FT-CORBA and FA-CORBA infrastructures full consistency of replicas was to be ensured, an idea is to see at what extent consistency constraints can be relaxed to provide higher availability. In this context, a special feature would be to consider network partition failures, and allow request servicing in different parts of the replicated application, at the cost of reducing consistency.

Experimenting with only one application does not lead to general knowledge. One extension of this work consists in setting up one or more different applications
and conduct experiments for enriching the trade-off studies with new results. To make the FT-CORBA platform to fairly compete with the fully available platform we envisage extending that work by replicating the infrastructure units as well.

Using aspect-oriented programming was successful in combination with the FT-CORBA middleware for passive replication. We can see a continuation in this sense by applying this new and elegant technique to solve performance problems in the FA-CORBA infrastructure and for other replication styles as well.

The main contribution of the work regarding the checkpointing interval optimization is that it looks deeper at the failover time in terms of probability distribution of the number of calls to replay. However, it considers the distribution of replay time to be independent of the call to replay. An immediate extension in this direction is to consider different probability distributions depending on the call to replay. Other extensions of the work are also possible. For example one could consider different service time distributions (exponential or not) for the different requests served. A related extension is to consider different interarrival time distributions for different types of requests. We can also think about extending the model to include the state transfer time modelled as a random variable. Last, but not least, one can consider request arrivals as non-independent.

Some of the above extensions are tractable mathematically, some are not. For example, by changing only one of the distributions from exponential to non-exponential, the mathematical computations become almost impossible. However, considering different request types to have different arrival rates, while keeping exponential distribution of the interarrival time can be an extension to consider. The same is valid for considering different service rates for different request types.

As in the work on optimization we did not consider the heavy traffic case, in the future we plan to investigate this aspect. We will base our analysis on the work by Lehoczky [93]. Also we plan to provide simulations of the refined mathematical model to validate the model used for minimizing average response time.
Bibliography


<table>
<thead>
<tr>
<th>No</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>No 14</td>
<td>Anders Haraldsson</td>
<td>A Program Manipulation System Based on Partial Evaluation, 1977</td>
<td></td>
<td>91-7372-144-1</td>
</tr>
<tr>
<td>No 17</td>
<td>Bengt Magnhagen</td>
<td>Probability Based Verification of Time Margins in Digital Designs, 1977</td>
<td></td>
<td>91-7372-157-3</td>
</tr>
<tr>
<td>No 18</td>
<td>Mats Cedwall</td>
<td>Semantisk analys av processbeskrivningar i naturligt språk, 1977</td>
<td></td>
<td>91-7372-168-9</td>
</tr>
<tr>
<td>No 22</td>
<td>Jaak Urmi</td>
<td>A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978</td>
<td></td>
<td>91-7372-188-3</td>
</tr>
<tr>
<td>No 33</td>
<td>Tore Risch</td>
<td>Compilation of Multiple File Queries in a Meta-Database System 1978</td>
<td></td>
<td>91-7372-232-4</td>
</tr>
<tr>
<td>No 51</td>
<td>Erland Jungert</td>
<td>Synthesizing Database Structures from a User Oriented Data Model, 1980</td>
<td></td>
<td>91-7372-387-8</td>
</tr>
<tr>
<td>No 54</td>
<td>Sture Hägglund</td>
<td>Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980</td>
<td></td>
<td>91-7372-404-1</td>
</tr>
<tr>
<td>No 77</td>
<td>Œsten Oskarsson</td>
<td>Mechanisms of Modifiability in Large Software Systems, 1982</td>
<td></td>
<td>91-7372-527-7</td>
</tr>
<tr>
<td>No 94</td>
<td>Hans Lunell</td>
<td>Code Generator Writing Systems, 1983</td>
<td></td>
<td>91-7372-652-4</td>
</tr>
<tr>
<td>No 97</td>
<td>Andrzej Lingas</td>
<td>Advances in Minimum Weight Triangulation, 1983</td>
<td></td>
<td>91-7372-660-5</td>
</tr>
<tr>
<td>No 109</td>
<td>Peter Fritzson</td>
<td>Towards a Distributed Programming Environment based on Incremental Compilation, 1984</td>
<td></td>
<td>91-7372-801-2</td>
</tr>
<tr>
<td>No 155</td>
<td>Christos Levcopoulos</td>
<td>Heuristics for Minimum Decompositions of Polygons, 1987</td>
<td></td>
<td>91-7870-133-3</td>
</tr>
<tr>
<td>No 174</td>
<td>Johan Fagerström</td>
<td>A Paradigm and System for Design of Distributed Systems, 1988</td>
<td></td>
<td>91-7870-301-8</td>
</tr>
<tr>
<td>No 192</td>
<td>Dimitr Drankov</td>
<td>Towards a Many Valued Logic of Quantified Belief, 1988</td>
<td></td>
<td>91-7870-374-3</td>
</tr>
<tr>
<td>No 214</td>
<td>Tony Larsson</td>
<td>A Formal Hardware Description and Verification Method, 1989</td>
<td></td>
<td>91-7870-517-7</td>
</tr>
<tr>
<td>No 221</td>
<td>Michael Reinfrank</td>
<td>Fundamentals and Logical Foundations of Truth Maintenance, 1989</td>
<td></td>
<td>91-7870-546-0</td>
</tr>
<tr>
<td>No 239</td>
<td>Jonas Löwgren</td>
<td>Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991</td>
<td></td>
<td>91-7870-720-X</td>
</tr>
<tr>
<td>No 244</td>
<td>Henrik Eriksson</td>
<td>Meta-Tool Support for Knowledge Acquisition, 1991</td>
<td></td>
<td>91-7870-746-3</td>
</tr>
<tr>
<td>No 252</td>
<td>Peter Eklund</td>
<td>An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991</td>
<td></td>
<td>91-7870-784-6</td>
</tr>
<tr>
<td>No 258</td>
<td>Patrick Doherty</td>
<td>NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991</td>
<td></td>
<td>91-7870-816-8</td>
</tr>
<tr>
<td>No 260</td>
<td>Nahid Shahmehri</td>
<td>Generalized Algorithmic Debugging, 1991</td>
<td></td>
<td>91-7870-828-1</td>
</tr>
<tr>
<td>No 264</td>
<td>Nils Dahlbäck</td>
<td>Representation of Discourse-Cognitive and Computational Aspects, 1992</td>
<td></td>
<td>91-7870-850-8</td>
</tr>
<tr>
<td>No 265</td>
<td>Ulf Nilsson</td>
<td>Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992</td>
<td></td>
<td>91-7870-858-3</td>
</tr>
<tr>
<td>No 270</td>
<td>Ralph Rönquist</td>
<td>Theory and Practice of Tense-bound Object References, 1992</td>
<td></td>
<td>91-7870-873-7</td>
</tr>
<tr>
<td>No 273</td>
<td>Björn Fjellborg</td>
<td>Pipeline Extraction for VLSI Data Path Synthesis, 1992</td>
<td></td>
<td>91-7870-880-X</td>
</tr>
<tr>
<td>No 276</td>
<td>Staffan Bonnier</td>
<td>A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992</td>
<td></td>
<td>91-7870-896-6</td>
</tr>
</tbody>
</table>


