Performance and Availability Trade-offs in Fault-Tolerant Middleware

Diana Szentiványi
Anyukámnak
és apukám emlékének,
sok szeretettel és tisztelettel

To my mother, and
to the memory of my father,
with love and respect
Acknowledgments

First, I would like to thank my supervisor Simin Nadjm-Tehrani for being a great guide in my work and a wonderful friend. I could not accomplish this work without her help. The discussions with Simin gave me priceless knowledge, that I am very grateful for.

A special thanks goes to Nick Szirbik for putting me on his list in June 1998, thereby making possible all this change in my life.

I want to express my endless gratitude to Professor Petru Eles, my favourite teacher in Timișoara, for his immense patience to me during my time in ESLAB and his further support and encouragements. I also thank Petru for reading this thesis and making very helpful comments.

I want to thank the 1998-1999 ESLAB team for offering me a very good time with nice coffee breaks and discussions.

I am grateful to my colleagues in RTSLAB for offering me a great working environment and nice outings. I hope to stay with them for some more time. I owe a special thanks to Anne Moe for always sorting out the tricky administrative problems. I thank my colleague Călin Curescu for preparing the test application and for being the person next door.

I thank Torbjörn Örtengren from Ericsson Radio AB for initial discussions that led to the project TRANSORG and for providing us the test application code.

I thank Isabelle Ravot, our guest from Lausanne, for doing the work with the CORBA implementation of the robust algorithm. I also thank Johan Moe for the insightful discussions and new ideas.

I also want to thank Bodil Carlsson, Lillemor Wallgren and Britt-Inger Karlsson for helping me out with putting together the thesis book.
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, and by CENIIT (Center for Industrial Information Technology) at Linköping University.

Last, but not least I would like to thank my dear friends from here and abroad for being what they are, for always accepting me the way I am, and for believing in me. Knowing that they exist made my life happier and my work more enjoyable.

Diana Szentiványi
## Contents

1 Introduction
   1.1 Motivation ................................................. 2
   1.2 Problem description ........................................ 3
   1.3 Contributions ............................................. 4
   1.4 Thesis outline ............................................ 4

2 Fault Tolerance: Basic Notions ................................ 7
   2.1 Models for fault tolerance ................................. 7
      2.1.1 Historical perspective ............................... 7
      2.1.2 Means of achieving fault tolerance .................... 8
      2.1.3 Recognizing failures ................................ 10
   2.2 FT network services: basic notions ....................... 13
      2.2.1 Models of FT networks ................................. 13
      2.2.2 Communication in presence of replicas ............... 15
   2.3 Consensus as a basic primitive ............................ 16
      2.3.1 Basic algorithms for consensus ....................... 17
      2.3.2 Perfect failure detectors ............................ 19
   2.4 The process group abstraction ............................. 20
      2.4.1 The notion of *group* ................................. 21
      2.4.2 Replication strategies ............................... 21
      2.4.3 Typical group services ............................... 23
      2.4.4 Specification and implementation of group services .. 23
   2.5 Measures of efficiency .................................... 25

3 Availability in Distributed Systems ......................... 27
   3.1 The issue of availability ................................ 27
      3.1.1 Using the dependability model ........................ 28
      3.1.2 Using metrics to analyze availability ............... 29
7 Evaluation of the Robust Algorithm

7.1 Experiment setup .......................................................... 89
7.2 Expected results .......................................................... 90
7.3 Results ................................................................. 91
  7.3.1 The artificial application ........................................... 91
  7.3.2 Telecom application - overheads ............................... 92
  7.3.3 Telecom application - failover times ......................... 95
  7.3.4 Improvements to the algorithm ................................. 97

8 Conclusions and Future Work ........................................... 101

8.1 Conclusions ........................................................ 101
8.2 Future work ......................................................... 102
Chapter 1

Introduction

Computer based systems are very used today in areas that range from cars and airplanes to telecommunication and e-commerce applications. Distribution of functions in these systems is widely applied. It is crucial for this kind of systems to be able to continue providing services even if faults (defects) are present in the system.

Dependability is a concept with a large number of attributes, themselves being 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 [24]. An alternate complementary definition for dependability was introduced in 2001 by Avizienis et al. [3]. 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.

Availability is the attribute of dependability that describes the readiness for correct service of the system [3]. Safety is the property of a system that assures the absence of catastrophic consequences on the environment ([3]). Maintainability is the property of a system expressing the ability to undergo repairs and modifications ([24, 3]). According to Verissimo [74] maintainability is in fact the time needed by a system to recover from a failure. 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. Therefore, fault tolerance is
important for the safety and availability of critical applications like airplanes, or for the availability of, e.g., telecommunication platforms. 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 in an easy way and with low effort. Furthermore, the maintainability (ability to undergo repair) of the systems built has to be taken in consideration.

This thesis describes results concerning the incorporation of fault tolerance support in the middleware on top of which the application is built. Middleware is a general term for any programming that serves to "glue together" or mediate between two separate and usually already existing programs [76]. A different definition for middleware is given by Bakken [7]: "middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems". Besides, Bakken mentions several categories of middleware. In the work presented in the present theses, the middleware is a transparent communication media. It allows for accessing remote objects as if they were local ones.

The extension of this middleware with fault tolerance is a so called fault-tolerant infrastructure. Such an infrastructure helps the application writer to avoid implementing fault tolerance mechanisms in the application.

The future of middleware and distributed systems' building infrastructures is pointed out in a recent collection of articles [1]: "Adaptive middleware". An already existing trend is to build applications that can request different quality of service. Some examples are mobile applications, multimedia clients and servers, environment aware applications. Also, there is a need to allow building of distributed applications for real-time embedded systems [64]. A solution to the problem of adaptability is to use reflective middleware [37], to compose middleware services [71], or to specialize some existing middlewares for different application purposes.

This thesis reports about investigation results concerning the possibility of extending a well-known middleware (CORBA) to include support for fault tolerance following the FT-CORBA standard.

1.1 Motivation

It is a good practice to include fault tolerance mechanisms in the design of a system. Also, there is a need for standardizing the way of building systems that combine fault tolerance features with other, e.g. performance


1.2 Problem description

When using a fault-tolerant infrastructure, the telecom engineer, for example, is interested to know what is the cost of having the fault tolerance features embedded in the middleware that was used up to this moment and was quite well understood. In particular, it is worth looking at what is the extra time spent with servicing a request while no failure occurs (the no-failure scenario happens most of the time). Then, an interesting point is to measure the time taken for the distributed server to "reconfigure" itself after and error, also known as failover time. In earlier solutions this operation includes maybe manual restart of some server, which is more time consuming. Questions worth studying arise in relation with resource consumption in general.
(such as processing power, storage space, network bandwidth), although this problem can be found in almost any setting where a fault-tolerant system is built. Also, it is not clear to what extent the implementation of the infrastructure can influence the above mentioned measures. Another problem is to determine what other parameters (such as group size, replication style\textsuperscript{2}) can influence the performance and resource usage. It is also a question how restrictive must the failure model be in order to allow a reasonably sized fault-tolerant infrastructure to be built.

1.3 Contributions

In real-life industrial applications such as telecommunications, distributed systems find a strong area of usage. 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.

This thesis work contributes by:

- the description of how a CORBA implementation is enhanced with fault tolerance capabilities by following the CORBA standard;
- setting up a realistic (telecommunication) application on top of the extended middleware infrastructure;
- evaluating the performance of the application on top of the new middleware, by trying to find out the influence of different parameters on the timing values;
- questioning the feasibility of a CORBA standard fault-tolerant middleware and by pointing out in a comparison, the benefits and drawbacks of an alternative CORBA infrastructure.

1.4 Thesis outline

The thesis is divided in seven chapters, as follows:

- Chapter 2 presents an overview of the field of fault-tolerant computer systems. It introduces basic notions such as asynchrony, failure detectors, consensus, reliable broadcast. These notions form the theoretical basis for the later presentation in the document.

\textsuperscript{2}These notions are presented in Chapter 2
1.4 Thesis outline

- **Chapter 3** presents a set of related works in the area of availability and fault-tolerant infrastructures. The works mentioned include Java RMI and CORBA middleware based infrastructures.

- **Chapter 4** gives an overview of the building process of the FT-CORBA standard infrastructure.

- **Chapter 5** describes experimental results obtained when evaluating the built infrastructure with a realistic application running on top of it. One of the outcomes here is the pointing out of shortcomings in the FT-CORBA standard.

- **Chapter 6** presents an alternative infrastructure, not entirely complying with the FT-CORBA standard. It has, on the other hand, interesting properties like robustness to infrastructure failures and inclusion of unreliable failure detectors.

- **Chapter 7** compares and summarizes the results in chapters 5 and 6.

- **Chapter 8** concludes the thesis and indicates possible future directions.

Chapters 4 and 5 are an extended version of [68]. Chapters 6 and 7 present the extended content of [69], a work in submission during autumn 2002.
CHAPTER 1. INTRODUCTION
Chapter 2

Fault Tolerance: Basic Notions

This chapter gives an overview of the research problems and results in the area of fault-tolerant computing systems. The notions that will be introduced range from synchrony, asynchrony, replication, consensus, to group communication and fault-tolerant middleware.

In the following sections two words are often used: process and processor. When talking about fault tolerance by software replication, they can be used, in general, interchangeably. However, in some cases, one or the other term is more appropriate. Processor is used with the meaning of hardware support on which computations are performed. Process is used with the meaning of piece of software that executes the computations, together with the support on which it is running.

2.1 Models for fault tolerance

This section presents some generic notions used in the process of constructing fault-tolerant systems. A short historical view is also given, to emphasize the fact that an agreement on notions and definitions was also an issue in this area.

2.1.1 Historical perspective

Basic notions of fault tolerance were introduced during the years (starting in the 1970's). The area is large and diverse. As a consequence, in the early
years there was little unifying work to formalize the basic 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 fail-safe (crash) and fail-stop are used in one way by some authors [67, 66], and the opposite way by others [26]. An early attempt at defining common agreed upon terms appeared in the work by Laprie [40], that was later on adopted and developed as the working document for the IFIP WG 10.4 on dependable computing. In 2001 a new document about dependability concepts ([3]) appeared, bringing in few changes or additions to the basic notions.

One of the early attempts at distinguishing models for fault tolerance appeared in the works of Cristian [16]. However, it was not until late 90's that a general framework for formal treatment of fault-tolerance appeared [28].

2.1.2 Means of achieving fault tolerance

Some authors see the requirement for fault tolerance as a part of the specification of the system, i.e. derived from availability, safety, integrity, etc. [42, 5]. 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. In other works the notion of fault tolerance is related with 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 [59]. 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 [61, 43, 66, 16, 8], and whether the fault tolerance technique is employed in a particular setting, e.g. databases [9] or disk storage [29].

In most cases, however, multiple versions of a process are added to achieve fault tolerance. The same thing 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). For both types of “multiplication” we talk about redundancy, but in the process case we even use

\footnote{International Federation for Information Processing Working Group 10.4}
the term *replication*. Thus, in replicated systems we have several copies of some entities contributing to the production of the final result, that should be unique.

A well known approach of 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: N-version programming [4] and recovery blocks [6].

- When using the first technique, several versions of the same program run in parallel and there is a procedure of choosing the result of the computations among the ones produced by every version. A voter is used for this purpose.

- When employing the recovery block technique, the outcome of the specific program to be executed is passed through an acceptance test. The acceptance test is constructed according to the application. If the test is not passed, then an alternative recovery block is chosen and the possibly new version of the algorithm is executed again. The result is tested again and if it passes the test no new alternative has to be executed.

Having replication alone in a system is not enough to achieve fault tolerance. There has to exist a mechanism that tells when a backup replica must replace a failed unit. Also, there has to be a way to know when to leave out a failed node from a decision algorithm.

In this context the notion of *consensus* (the notion of consensus will be described in Section 2.3) is introduced. It relates to the need of multiple processing units to agree or reach consensus on the produced result. This notion has generated a lot of research in the distributed systems community, devising algorithms for solving the problem.

An important early result that possibly affected the future directions considerably is the theoretical result on impossibility of consensus in asynchronous systems in presence of at least one failure [27] — a theorem established by Fischer et al. in the mid-80's. In a synchronous system every system component takes one step whenever other system components take \( n(n \geq 1) \) steps. In time-free asynchronous systems this is no longer the case. Processing units can execute a step without any relation to other nodes' activity. Also, there is no known upper bound on the delay in communication amongst units. Thus, a node cannot be considered as failed only because one message from its part is delayed, as compared to a previous time. Thus, in case the node really failed, it is possible to wait forever for its message,
and if this message is needed for the other processes to be able to continue, then they will wait forever.

There has been a large body of work on clock synchronization algorithms over the years (for historical snapshots see [67, 73]). However, the impossibility result tells us that since the asynchronous distributed system has no global notion of time, there is no deterministic algorithm to distinguish between a failed process and an extremely slow process. Hence a failure detector and thereby fault tolerance measures can not be used to compensate for failures.

2.1.3 Recognizing failures

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 another component of a fault tolerance model. It can be seen as a contributing factor in achieving fault tolerance. Failure detection is the mechanism that together with replication helps achieving fault tolerance.

The notions of fault, error, and failure

Although mentioned constantly in previous sections these basic notions and mainly the relationships between them were not properly explained.

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.

Finally, 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. A failure can be seen as the deviation of the behavior of the system from the specified one. 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.

Some classifications of faults and failures are given in the literature [24, 3]. 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 hardware circuit
2.1 Models for fault tolerance

- 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

A fault that does not cause an error, i.e. it not active, is called dormant [3].

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 [17]. 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 if failures indicated in the model occur. Thus, the algorithm is designed such that 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.

A common model is the crash failure model. This includes failures of nodes by simple crash. To fail by crashing means that the unit stops executing its computation and communication, and does nothing further. Another name for the crash failure model is the fail-silent model.

A more serious type of failure is the Byzantine failure, where the unintended behaviour of a unit can be arbitrary, and potentially caused by malicious attacks. Verissimo and Rodrigues [74] describe these types of failures as a subset of arbitrary failures. Arbitrary failures appear when omission failures are combined with assertive failures and inconsistency. Basically, omission faults occur when a component misses to execute some action. Assertive faults happen when the component executes some interaction in a non-specified way. It can produce an incorrect construct of an interaction (syntactic failure), or an incorrect meaning conveyed by the interaction (semantic failure). Inconsistency of failures arises when not all recipients of, e.g. a message, experience an omission failure.

When several processing units collaborate to provide fault-tolerant services, the communication media among these units is also important to analyze. Thus, in a distributed fault-tolerant service provider, there are two major types of failures. This classification is based on the location of the
failure. In addition to the previously mentioned process failure, we have the class of network failures. These 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. 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. Thus, a new failure model, concerning the communication media, is the partition failure. 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.

**Unreliable failure detectors**

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 (not bounded also means infinite).

However, there is a possibility to weaken the time-free assumption, without departing too much from a real setting of a distributed system. What if one simply supposes that there is some bound on message delay, but at the same time is conscious that this supposition can be wrong?

This aspect is explored by Chandra and Toueg in their formal characterization of failure detectors [14]. They actually play around with "mistake patterns" that enable them to specify the so-called unreliable failure detectors.

For example, when executing some consensus 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 state of the other members of the replicated group.

The failure detectors can make mistakes, by suspecting processes that are correct (i.e. those that have not crashed), 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 (all correct processes will eventually 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. More exactly,
these properties are eventual weak accuracy\(^2\) and weak completeness\(^3\). The main contribution of this group is to show that the impossibility result by Fischer et al. can be overcome if the failure detector properties hold for sufficiently long time for performing the algorithm.

### 2.2 FT network services: basic notions

This section describes some basic notions in the context of using distributed systems for building up replicated services. Thus, replication in space will be covered.

#### 2.2.1 Models of FT networks

Algorithms and applications for distributed systems require a model of the underlying system. The model is built by making assumptions about the physical system consisting of processing units and communication media.

As seen earlier, the main assumptions in relation with time are synchrony and asynchrony. A third model is based on *partial synchrony*. A partially synchronous system model is one in that there are intervals during the system's life-time when there exist bounds on message delays and on process execution times. This is the general idea, but there are variants to this notion, also referred to as *partial stability*, in that the bounds are not known \([23]\). 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. Further, under a certain time-related assumption, different kinds of failure models exist. These include network failures as well as process failures.

**Network failures**

When considering the communication media, two types of failure models can be considered. One is the *partition* failure model (explained in Section 2.1.3). For example, the algorithms devised by Babaoglu et al. \([6]\) are built starting from this assumption.

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

\(^2\)eventually there exists one correct process that is never suspected by any correct process

\(^3\)eventually, every crashed process is permanently suspected by some correct process
Chapter 2. Fault Tolerance: Basic Notions

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 [63]. 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.

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.

Process failures

In connection with processes, the most usual assumption is that one can fail by crashing. Other failures such as commission (Byzantine) failures, when processes fail in an arbitrary way are harder to address. In presence of failure detectors, a process can be suspected to have crashed. Further assumptions can be made about a process’ behaviour after an actual crash, or the reversal of a suspicion.

First, we can have the recovery assumption. Usually, recovery is related to a real crash. It simply means that an actually crashed process is repaired after a while, and joins further decision making steps in one way or another. Sometimes, the recovery assumption is related with a suspected crash. In this case recovery means that the suspicion can be revoked provided that the process is found to be correct later on (this assumption is used by Chandra and Toueg [14]).

The opposite model is based on the no-recovery assumption. Again, we can relate the assumption to a real crash, in which case we assume that the process is not repaired and will never participate again in decisions. On the other hand, when no-recovery is related to a crash suspicion, it means that once a process is suspected, the suspicion is never revoked. This notion is called failure belief stability by Ricciardi et al. [63]. The no-recovery assumption is used in both senses by Helary et al. ([34]).

A failure model is built by combining these assumptions. For example, Babaoglu et al. use a failure model built on crash/no-recovery/partition assumptions [6]. The no-recovery is used in its first sense (related to real crashes, i.e. it is possible to revoke a crash suspicion).

In Section 2.3, it is described how the above assumptions affect some of the algorithms.
2.2.2 Communication in presence of replicas

Because in a replicated reliable service framework, all processes have to be involved in the communication, the sending of messages is done via broadcasts\(^4\) [32].

However, since failures are expected to happen, even a broadcast activity can be affected by problems caused by failures. Thus, there is a need to characterize reliable broadcasts, and that needs the notion of message delivery (as distinct from reception) defined as follows:

- **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

From the agreement property it follows that every message is either delivered by all correct processes or by none of them, that is a desirable property specially when broadcast can be affected by failures. The other two properties imply that the underlying message delivery system does neither create, nor loose messages.

**Message ordering**

Besides being reliable, sometimes a broadcast primitive has to give also different message ordering guarantees. Thus, even if all sent messages are received by all destinations, the order in which these messages are delivered

\(^4\)When the message is sent to only a subset of a global destination set, then we talk, instead of broadcast, about multicast.
by all processes is important. Depending on the delivery order restrictions, we can distinguish different types of 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 \[32, 74\].

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 [10].

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 \textit{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) \[32, 74\].

Further, there are broadcast primitives that can give combinations of ordering guarantees. For example FIFO-causal means that messages are delivered in a causal order and for messages that are not causally dependent on each other, if they are received from the same sender, they are delivered in FIFO order \[32, 74\].

### 2.3 Consensus as a basic primitive

Consensus was mentioned earlier as a useful operation in distributed systems. The operation is usually defined in terms of the two notions \textit{propose} and \textit{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 to a point when they decide on the final value. The algorithm has to be constructed in such a way that it will be guaranteed 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
2.3 Consensus as a basic primitive

- 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 relation between a number of problems in distributed systems and consensus has been a major area of research. It has, for example been proven that some problems can be solved by using consensus (and vice-versa). Thus, some problems in distributed systems are reducible to consensus and vice-versa. Examples of such problems are the atomic broadcast problem [14], and the atomic commit problem [30]. For the first case, the authors prove the equivalence of the two problems. The second problem is actually a particular instance of the consensus problem [30]. Thus, here, the reduction is only one way.

For example, take the case of 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.

2.3.1 Basic algorithms for consensus

In this section some algorithms for solving the consensus problem are presented. Different assumptions on the distributed system are made, from the perspective of node behaviour after a crash (recovery/no-recovery assumption) and from the perspective of link failures. For the latter category, algorithms are given only for the no-partition failure model. It does not make too much sense to talk about unique decision making in a partitioned system, except for the case when it is possible to determine the primary partition.

No-recovery/no-partition model

Chandra and Toueg give solutions to the consensus problem in asynchronous systems with unreliable failure detectors [14]. The no-recovery assumption is used here in the sense that after a (real) crash a process is not repaired.
It is possible, on the other hand, to “change one’s mind” if a process is only suspected to have crashed. They prove that consensus is solvable with the weakest category of failure detectors. This type of failure detector is called \textit{eventually weak} and satisfies weak completeness and eventual weak accuracy. Two algorithms are given (in pseudocode form) to solve consensus: one for the system equipped with \textit{strong} failure detectors (those satisfying strong completeness and weak accuracy), and one for the system equipped with \textit{eventually strong} failure detectors (those satisfying strong completeness and eventual weak accuracy).

- In the first case (strong failure detectors), consensus can be solved even in the presence of $n - 1$ process failures. The weak accuracy property is satisfied. In other words, there is a correct process that will never be suspected by any correct process. 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 \textit{round} – a process waits for messages sent in the same round by the other non-suspected processes. The no-partition assumption is used, even if it is not specified explicitly. By the fact that there exists one correct process that is never suspected as being down, disjoint sets of processes that work on their own, can never appear.

- In the second case, consensus can be solved in the presence of at least a majority of correct processes. This 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. It has to be mentioned that by suspecting the coordinator to have crashed, processes move to a new round with a new coordinator. Thus, the coordinator in the previous round did not manage to decide. However, as mentioned above, this situation will not last forever. In this case, the no-partition assumption is used in its primary-partition form. By always needing the agreement of at least a majority of processes in order to take a decision, it is clear that there cannot be more than one set of processes that decide.
2.3 Consensus as a basic primitive

Recovery/no-partition model

Aguilera et al. adopt a totally different view on solving the consensus problem and come up with a new notion of failure detector suitable for the crash/recovery assumption [2]. A crashed process can be repaired, and can thus recover. The algorithm incorporates suspicion revocation, as well.

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 too restrictive for them to be permanently suspected as prescribed by the strong completeness property in the sense of Chandra and Toueg [14].

2.3.2 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 cannot 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.

Helary et al. discuss the problem of decision based on a global set of values in the presence of perfect failure detectors [34]. This can be seen as a strict(er) form of consensus – called by the authors 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. These channels, thus, could be used by processes for sending “I am alive” messages. Here, as already mentioned before, the no-recovery assumption is used, in both its senses. 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 is that failure detectors used are perfect and they do not suspect a crash if this is not real. The no-partition assumption is quite obvious here - again because of the use of perfect failure detectors. There is no mistaken crash suspicion, thus 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. [21]. The discussion here is about the weakest failure detector to solve consensus,
atomic broadcast and terminating reliable broadcast in case the number of failed processes is not restricted. Delporte et al. prove that the failure detector of class \( \mathcal{P} \) (perfect) is the weakest to solve the above mentioned agreement problems. Also, they consider 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 in a perfect failure detector. From here, the conclusion that the perfect failure detector is the weakest to solve the problems.

The Timely Computing Base (TCB) model introduced by Casimiro and Verissimo ([72, 13]) 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 \( (T_{TFD_{max}}) \) 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 \( (T_{TFD_{min}}) \) before the specified termination moment of the respective action.

2.4 The process group abstraction

Intuitively, a group can be seen as containing a set of replicas being able to do the same processing task. Here we have the difference between processor groups running a set of replicated services, and service groups.

The service groups (servers) are also processor groups, but they all run the same service. The issue is that a processor can run more than one type of service. To maintain service replication, for each service we must know the number of processors running that service, and if this drops under a certain limit (the replication limit for that service), then the service has to be started on some other processor.

One of the successful experimental platforms around the ideas of software replication were implemented in the Delta-4 project [59]. This work leaves the decision concerning the particular method 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.

2.4.1 The notion of group

In previous sections, although the word group was sometimes mentioned, it was used only in an intuitive sense. In this section, the notion will be more precisely delimited. Thus, what is meant by group in the fault tolerance 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 that process requests from clients. The important aspect is the transparency from the client's side. Thus, the client, when sending a request to the replicated server (or now, a process group), perceives it as one addressable unit.

The group actually underlying the service is not visible for the user of the service. It might be used to manage complexity, to balance load, or to provide fault-tolerant services [60]. A consequence is that the group has to manage itself to assure the one-unit image.

An important point becomes the notion of 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. This case is conforming to the partition failure model. As already mentioned, in a distributed system, it is possible to have several groups of processing units working separately. Sometimes this is permitted, sometimes it is not.

2.4.2 Replication strategies

A replica group can be constructed in several ways:

- 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 [31].
• Active replication: in this approach, all replicas process requests from the client in an active way. In this case, the notion of group is not used in its above mentioned strict sense - it is not necessary for the replicas to know who is up and current member of the group.

The restriction here is that all replicas perform deterministic computations in a state machine fashion [65]. Depending on the failure semantics of the replicas the client has to wait for the first answer from any replica, or for an answer resulting from a majority vote performed by the replica group.

In the latter case, processes do not communicate with each other explicitly – consistency among them is guaranteed by the deterministic processing of the same inputs.

• 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 [20].

• Semi-passive strategy: in this (semi-primary-backup) replication style, the primary is elected by a consensus algorithm [59].

The notion of state might need some clarification. The state (as used in “state update” or “state machine”) 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.

In all types of replication strategies, failures have to be handled such that the states of the replicas remain consistent. In the first case, 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. To choose a new primary, a membership service, that indicates the composition of the group, can be used. In case of active replication, failures of members can be masked by using a voting mechanism on the results (if Byzantine failure semantics is considered), or by sending one of the answers (the fastest one) in case of crash failure semantics of the replicas. In the former case, the number of replicas has to be $3f + 1$ if the failure of $f$ replicas has to be masked [39], whereas in the second case, it suffices with one replica more than the number of failures tolerated [65].
2.4.3 Typical group services

An indispensable service in a group setting is the one that informs about the group membership. This is used by the group members themselves 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) (sub)groups. 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 in case a new node joins the group. The operation of state transfer has to occur as an atomic action.

2.4.4 Specification and implementation of group services

Extensive work on the group membership problem can be found in the works of Cristian ([16, 18]). He gives a formal description of membership service properties both for synchronous systems and asynchronous systems.

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. [6]. They give a formal description – by introducing some quite simple properties – of a communication infrastructure for processes in a partitionable distributed system. Thus, here, the failure model is built using the partition assumption, and partition failure can be also present. In this setting, a process does not recover after a real crash. Thus, the no-recovery assumption in its first sense is used here. On the other hand, it is possible to reverse one’s suspicion about a crash. This, actually, follows from the fact that the suspicion itself can be related to process crashes as well as process unreachability.

Work by Birman et al. on the ISIS system, has minted the notion of virtual synchrony and view synchronous multicast. The first notion is re-
lated 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. The second notion is tightly coupled with the former one, and directly relates to groups. Thus, a view synchronous multicast (or broadcast) 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\(^5\), 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 [12]. Further, the procedure execution was more transaction based. Later, however, with the introduction of the notions of virtual synchrony and view synchronous multicast, algorithms and communication primitives are described more formally [10]. Also, a process group approach was used to provide fault-tolerant services [11].

2.4.5 Application platforms

Besides the ISIS system, there are some other fault-tolerant distributed system platforms.

One of these is Totem developed by L.E. Moser et al. [49]. This system provides a platform for writing distributed applications. It is structured in several layers, deals with sending messages among different process groups, enforces total order on delivery of messages, and also deals with merging of groups. An extension of the notion of virtual synchrony is introduced here: the extended virtual synchrony. In this model, care is taken even of the processes not in the current group of a certain process. Thus, 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 and can rely on the fact that the system will enforce the total order delivery. The platform is suitable for use in soft real-time systems, since high throughput can be achieved, as well as predictable latency.

Another system, developed by Van Renesse et al. is Horus [70]. It is equipped with group communication (e.g. message passing) primitives, providing flexibility to distributed applications developers.

\(^5\)“registered” is referred to as *installed* by Birman et al. ([10])
This system allows flexible composition of protocols to support groups. The application developer can use well-defined protocol components (i.e., with well-defined interfaces) to build the needed protocol stacks, while excluding unnecessary overheads. Not all protocol block configurations make sense; only those for which one block's output interface can be connected to the other block's input interface. The authors mention their successful experiences with Horus [62], as well as remaining challenges to be taken up later.

2.5 Measures of efficiency

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 efficiency 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 [14] – the second type of algorithm). 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 [38]. 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. [72, 13]
employs a partly synchronous approach by using the timely computing base
(introduced in Section 2.3.2). This offers also the possibility to reason about
execution timeliness. Cristian and Schmick discuss real-time aspects in the
context of distributed algorithms, in a client-server configuration [16, 18].
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.

Helary et al. give upper (worst case) bounds for the number of rounds
executed in their algorithms [34]. 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].
Chapter 3

Availability in Distributed Systems

This chapter describes related work in the area of availability, with emphasis on fault-tolerant middleware infrastructures. Availability is very important in distributed net centric systems that have to provide their services whenever clients ask for them. It is important to quantify a system's availability in order to be able to improve its qualities in this sense. On one hand, distributed systems can be characterized by superior availability compared to centralized systems. On the other hand, maintaining and building a distributed system is difficult. If application writers have at their disposal a middleware it is much easier to write distributed applications, because some parts of the system already exist in the underlying platform.

Most of the works mentioned in this chapter include CORBA as the middleware used, or a Java-based frameworks for describing the system.

3.1 The issue of availability

This section presents work in the area of building dependable systems. Two main approaches are highlighted: employing a dependability model in the system creation process, and using metrics for building more dependable systems.
3.1.1 Using the dependability model

An important problem when developing dependable systems is to use an appropriate model for this goal [35]. A systematic and structured design framework is needed in order to include dependability concerns from very early stages of the development process. It is difficult to standardize such a framework due to the variety of application domains that come with many different types of failures, requirements and resource constraints.

The idea of Kaaniche et al. ([35]) is to use a “dependability-explicit” development model. Some processes are distinguished in this context:

- the system creation process containing classical development steps: requirements, design, implementation, etc.
- the dependability processes (means to reach dependability): fault prevention, fault tolerance, fault removal, fault forecasting
- quality assurance and certification processes

Dependability measures are to be incorporated in all stages of the system creation process. Activities directed towards dependability can be incorporated easily in this process and can be executed iteratively, in parallel or as a sequence. In each dependability process (fault prevention, fault tolerance, fault removal, and fault forecasting), three classes of activities were identified. These groups of three activities are different from process to process. In particular, for the fault tolerance process Kaaniche et al. identified the following activities that can be incorporated in the system creation phase:

- study the system’s behaviour under fault conditions, in order to identify faults against which to protect the system
- partition the system, to provide fault isolation, and independence
- fault and error handling, to devise appropriate fault tolerance mechanisms

In the context of the fault tolerance process that can be related to the fault forecasting process, levels of degraded service provision have to be defined. Also, tolerated fault occurrence patterns have to be defined. The authors finally give a set of checklists with activities to be performed during the development of dependable systems. The requirements phase and the design phase are covered.
3.1.2 Using metrics to analyze availability

Another way of enhancing systems in the direction of dependability - in particular, availability - is by doing different measurements on the system's behaviour under failure conditions [33]. More precisely, data are collected (usually via event logging facilities provided with different operating systems) mainly about failure causes (machine related or network related), and down-time. The analysis by Iyer et al. ([33]) concerns two types of local area network computers (with UNIX and with Windows NT) and the Internet computer. It is a difficult task to write the right messages in the log, and further to properly interpret the content of the log. Also, observations have to be conducted for long times (several month) in order to obtain meaningful results. Further, in cases when a failure is not machine related it is not always possible to find out the exact cause: is the link between the client and a server failed, or is the server overloaded. For example, down-times are measured by looking at the time-stamp of the last event in the log before the reboot, and the time-stamp of the event immediately after the reboot. This is correct if the reboot did not cause itself new failures. All error events apparently caused by the same fault (error event that are very close in time) have to be collected in a single error log element. 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. States in the machine represent levels of the functionality of a machine. The arc between say, states A and B is weighted with the fraction of the total number of transactions from A that are made to state B. For example: state A is Reboot state, state B is state Functional, and state C is state Connectivity Problem [33]. The arc from A to B is weighted with 0.41. The arc from A to C is weighted 0.0722. This 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).

Moe and Carr present a solution to the problem of understanding and modifying a distributed system [47]. 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. Also, misbehaviour due to configuration errors can be identified. Their 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, system performance has not to be affected. The quantity of data collected should be right for all analysis that have to be performed.

The results presented in this section are useful when devising failure models, for example, of infrastructures on which applications are built. The next sections will elaborate more on fault-tolerant infrastructures.

### 3.2 Java-based FT infrastructures

In networked systems, distributed functions are performed on different platforms. Thus, the presence of multiple operating systems in a realistic distributed application is a fact. If the application can be written using one single programming language, then using Java is a very proper choice for ensuring interoperability.

In this section we give a brief exposure to several works building on Java Remote Method Invocation (RMI) mechanisms.

The Java RMI framework is used to provide distributed fault tolerance in the Jgroup toolkit [48]. Application server objects are replicated for fault tolerance and the client accesses the server group as a single entity. The client uses *external group method invocation*. Special client stubs are created by a *group method invocation compiler* (similar to remote method invocation compiler rmic). Server group members communicate internally by *internal group method invocations*. These are used, for example, to communicate the result of a computation to all group members.

Another example of using Java RMI is the AROMA system developed at University of California, Santa Barbara [55]. There are three ways of adding fault tolerance to a Java RMI based system: the service approach
in which replication mechanisms are implemented as packages at the application level, the integration approach in which the same mechanisms are pushed at the level of the RMI infrastructure. However, this solution would request modifications of the JRMI framework. The third alternative (used in AROMA) is the interception approach: RMIIs are captured at the transport layer of the Java RMI protocol stack, by a software component (the interceptor) embedded there. The objectives of the AROMA system are transparency and replica consistency. Replication style is chosen depending on resource availability and failover time allowed. Every RMI to a replica (e.g., in case of primary backup replication) is intercepted and multicast to the other members in the group, and the same is true for state reading messages.

The Javelin Project [54, 53] was destined to use the large computational capability of the Internet to run very large coarse-grained parallel applications. The infrastructure is built using Java language. Application writers do not have to modify application code in order to cope with interprocess communication when parallelizing complex algorithms. The system is used to solve NP-complete problems such as the Traveling Salesman Problem. Processes running parts of the search algorithm are deployed on different hosts in the network, and use a shared memory implemented with a pipelined RAM cache consistency mechanism. When executing the search in the state space, costs of different solutions are passed between processes using this memory. There is a distributed scheduling scheme that is based on the work stealing approach. An eager scheduler makes sure that if a process fails, i.e., its result is not reported, a new one is started on an idle processor, thus achieving fault tolerance. Load balancing is also achieved by the eager scheduler.

### 3.3 Fault tolerance services in middlewares

CORBA is one of the most studied middlewares that offers interesting challenges in the direction of fault-tolerant computing. This section surveys the work in the area of enhancing CORBA applications with fault tolerance.

A failure mode analysis on a CORBA service implementation is conducted by Marsden et al. [44, 45]. 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\(^1\) messages sent to the server. Thus, the reaction of the server to corrupted requests can be studied. Depending on the implementation of the service, the exceptions generated give satisfactory information or not. For example, if the type of the exception is UNKNOWN, then not much is known about what recovery action has to be taken. Other observations are related to, e.g., where in the message a bit-flip happens. If, for example, the position is inside the message side field, then the indication about how much data is expected to arrive can be wrong and the server can block forever waiting for it.

Little and Shrivastava [41] present a way of integrating a group communication service with transactions. They start with a system that supports transactions, but no process groups. Then, they enhance the use of transactions by introducing process groups. Further, the authors consider the possibility of using a widespread middleware like the Common Object Request Broker Architecture (CORBA) [57]. The CORBA standard included already from an early stage a transaction service specification. It was in the year 2000 that fault tolerance by process replication started to enter the standard. A trade-off is found 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.

Prior to the specification of the FT-CORBA extension, few works had studied alternative augmentations of CORBA with a process (object) group module. Felber et al. show some possible approaches for introducing object groups in CORBA [25]. Three different approaches are presented depending on the position of the group communication module relative to the ORB\(^2\): 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 interceptor approach is considered to be more general in the

\(^{1}\)Internet Inter-ORB Protocol

\(^{2}\)Object Request Broker
sense that several deficiencies of current CORBA systems can be overcome by interceptors: non-application components that can alter the behavior of the application without changing the application code or the code of the ORB. The "object service" approach is the most CORBA-oriented. The Object Group Service is specified as a set of IDL\textsuperscript{3} 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. The Object Group Service allows transparent group invocations by clients. Thus, clients are not aware of the fact that they are invoking methods on a non-singleton object. Also, replicated application objects do not see the replication protocols. Method invocations are done directly on server interfaces, not by using some "multicast()" primitive. The OGS is constituted from a set of CORBA services specified separately and interacting through the ORB. They are defined as components and have usage relationships between them. These components 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
- 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 process (address space). In the second model, service objects are located in a different process on the same host (different process) 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

\textsuperscript{3} Interface Description Language
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 number of members of a group. They also experimented with optimistic active replication and unreliable multicast, and those showed a much smaller latency. All these results correspond to no failure case.

Mishra et al. [46] describe a CORBA group communication service based on a service approach. The design and two implementations are presented. A UDP\(^4\) 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 are 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.

Narasimhan et al. [50] propose an interceptor approach to enhance CORBA with fault tolerance. They also suggest that interceptors can be used not only to achieve fault tolerance, but also for several other purposes: real-time scheduling, profiling and monitoring, as protocol adaptors for protocols other than IIOP and for enhancement of security [50]. The result of the research work in this direction is the Eternal System. By using it, 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 mentioned in the work: system call interception and 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 information in this proc directory reflects, thus, what happens during the execution of a process. The files corresponding to processes can be accessed via specific interfaces and the information in them modified.

\(^4\)User Datagram Protocol
\(^5\)time it takes for an update to become stable, i.e. to be received by all replicas
Fault Tolerance Services in Middlewares

This happens based on the specification of what system calls, writing or reading information from that file have to be intercepted. In case of library routine, interception is based on the possibility to add shared objects to a process when it initializes. By using such a-posteriori loaded shared objects it is possible to replace originally used library routines that resolve a symbol with new ones modified for different purposes, like e.g. to provide fault tolerance. This is possible because it is allowed for an executable to have symbols (variables or functions) resolved only at runtime. 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.

Some problems related to replica state consistency arise and are pointed out in the context of Eternal [51]. The procedure of state transfer to a new replica of an actively replicated group is also presented. The work also presents experimental results when using the Eternal system with different ORBs (VisiBroker, Orbix, TAO). Several components of the replica state are delimited. The most important and visible state constituent is the application level state. This is obtained by using method calls specified in the FT-CORBA standard, from the application itself. A second, maybe not so obvious component of the object state in the Eternal system, is the ORB/POA level state. This has two constituents: GIOP 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 is given by the infrastructure level state. This component is invisible to the replicated

---

6Portable Object Adapter
7General Inter-ORB Protocol
application and to the ORB. This 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. The state transfer operation is presented as a sequence of six steps. An important aspect is to remember calls arriving at the server while the state transfer happens. The Eternal mechanisms make sure that the proper calls will be executed on the new replica after setting the state. When using the Eternal system, overheads given by the interception, multicast and replica consistency mechanisms were measured. The measurements were done under fault-free conditions. Overheads range between 10-15%, being of reasonable size. Also, failover times were measured and plotted against size of state (a part of failover time is time for transferring state). Failover times range between 15-55ms for state sizes between 100-3500 kBytes.

A framework for fault-tolerant CORBA services with the use of aspect oriented programming is presented by Polze et al. [58]. Their 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: crash faults, timing faults, design faults. The toolkit then chooses the appropriate fault-tolerance strategy. Communication between a client and a replicated service is done via an interface object. The considered replication styles are hot, warm and cold. Hot replication is the equivalent of active replication. In warm replication a primary replica processes the requests from clients and periodically transfers its state to the backups. In cold replication, the primary replica is checkpointed periodically to stable storage. At failover, the new primary is invested with the state in the latest checkpoint. The interface object (a sort of gateway) can be equipped with evaluators to be able to detect possible computation faults in the replicas. This interface object is pointed out as a single point of failure in the system. It is supposed that it runs on a very reliable system and that it has been tested and verified. Alternative implementations of the interface object are suggested: to put it inside a library and link it together with the client code. However, this would make the client a single point of failure. Another alternative would be to move its functionality to the ORB, but then the ORB single point of failure would be a problem. Still, this concern is left to the ORB vendors, and the work focuses, in fact, on a configurable framework allowing easy enhancement of existing CORBA services with fault tolerance. 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). The consensus aspect is also mentioned - it is implemented by application voter objects.

Chung et al. present a fault-tolerant infrastructure (DOORS) built using the service approach, on top of CORBA [15]. 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 a FT-CORBA implementation, Nataraajan et al. [52] present ways of improving the performance of a 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.

Killijian and Fabre [36] describe a framework for building fault-tolerant CORBA applications by using reflection. They define a meta-object protocol, and use an open compiler, 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.

**Summary**

The methods described in this chapter are directed towards devising ways of building dependable systems with enhanced availability. These ways are: the using of the dependability model and the system availability measurements and tracing. Also, examples of concrete fault-tolerant infrastructures are presented with emphasis on their building and the functionality they offer.

In the next chapters we present an alternative view on fault-tolerant applications. The middleware (CORBA) used when building an application is extended to include fault tolerance features, according to the standard. Further, the performance/fault tolerance trade-offs are studied and explained.
Chapter 4

Building an FT-CORBA Infrastructure

This chapter describes aspects of extending an existing CORBA implementation (OpenORB [19]) with fault tolerance features. The goal of this work was to adhere to the CORBA standard and, at the same time, give a measure of the performance/fault tolerance trade-offs. The extended middleware takes care of fault detection and failover in a failure prone system by offering full transparency. The first section will give an overview of the FT-CORBA standard. The rest of the chapter will describe infrastructure building blocks and related problems.

4.1 Background

The generic CORBA Specification by the OMG\textsuperscript{1} group was extended to provide application developers with support for fault tolerance. The Fault-Tolerant CORBA Specification V1.0 was adopted in April 2000 [56]. In December 2001, the FT-CORBA specification became part of the CORBA 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 in case of failures is done depending on the replication

\textsuperscript{1}Object Management Group
strategy used.

Support is provided for use of active replication, primary/backup and stateless\(^2\) 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 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 ReplicationManager interface inherits interfaces such as PropertyManager, Object-GroupManager, GenericFactory.

- The PropertyManager interface has methods used to set the replication style, the number of replicas, the consistency style and group membership style (both can be infrastructure-controlled or application-controlled)

- 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 \texttt{create\_object()} that in case of replicated objects is called transparently by the Replication Manager in response to a call to its own \texttt{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 (IOW).

\(^2\)This type of replication is used when the server object data is accessed only via read method calls
The Fault Notifier interface is also specified. It 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 is introduced. Each fault tolerance domain contains several hosts and object groups and a separate Replication Manager is associated with it. Also, there is one Fault Notifier in a fault tolerance domain, and on every host there has to be one Object Factory and one Fault Monitor. The Object Factory creates group replicas on demand. The Fault Monitor (detector) has to detect failures of group members running on its host.

Every application object has to implement two interfaces — PullMonitorable, containing method isAlive and Checkpointable, containing methods getState and setState — for the purpose of fault detection and checkpointing. It is also possible that the object implements the Updateable interface, and thus the methods getUpdate and setUpdate, to deal with state updates. This interface is appropriate to use when the state is quite big, 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 recovery mechanisms are automatically used in case of infrastructure controlled consistency and membership. They are needed only in case of 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.

The picture in 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 in the next sections.
Figure 4.1: Deployment of the FT-CORBA Infrastructure
4.2 Failure model

As noticed in the picture, the emphasis is on nodes and not on links. Thus, the failure model considered does not include link failures. Links are supposed to be reliable. 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 (if there is one) 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 round-trip time of the request, so that the connection stays up. In this context, it can be pointed out that the no partition assumption holds. It is also the case that if a message is delivered, this message is not corrupted. Also, no duplication of messages happens on the reliable links.

It has to be noted that the application on top of the infrastructure is an asynchronous distributed system. Thus, there is no bound on message delays. There is no way to differentiate between a slow server and a crashed one. Thus, all failures reported by failure detectors are just suspicions.

In the following, the presentation will mention two conceptually close implementations of the infrastructure. The second one was devised after analyzing results of experiments performed on the first. Also, it was meant to improve the first implementation from the perspective of robustness.

The failure model includes only node failures. To be more precise, crashes of server objects are deemed to occur. Thus, it can happen that an application server object is no longer able to process requests from clients, because it crashed. In the first simplified model, 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 this first simple version of the model. Thus, failover information such as state and information about requests serviced since the last checkpoint, can be 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. In a second model, the crash failure of the Logging and Recovery Controller Object (used intensively in case of cold and warm passive replication styles) is included. And, also, the possibility for an entire (primary) host to crash and lose all data stored in memory is considered. This case is interesting for cold and warm
passive replication. The modifications induced by considering the second failure model will be described in Section 4.4.1.

As specified in the FT-CORBA standard, there is a minimum number of replicas - the value of this number is supposed to be at least two - property for an object group. As soon as the number of previously created correct (up) replicas drop under this value, one (or more) new replicas have 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.

4.3 Other assumptions

By state of an object (application) we mean internal (and possibly external) application object data - fields of a class - that can be modified by method calls. This state of the application server objects can be updated only via CORBA calls (those which can be “seen” by hooks installed outside the application, at the ORB level). The type (update or read) 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 of certain types (classes). When doing this, it has knowledge about the types of the methods for that class, and uses this information to know whether to store the call in the log, in order to be used for replay at failover, if necessary. Read-only method calls are not stored in the log.

4.4 Infrastructure building blocks

When building fault-tolerant applications by using the present FT-CORBA standard infrastructure, the application writer needs the following ingredients:

- a collection of service (CORBA) objects
- CORBA portable interceptors at the client and server side
- ORB class extensions

We describe each building block in the three following sections.
4.4 Infrastructure Building Blocks

4.4.1 Service objects

The service objects are those specified in the FT-CORBA standard: the Replication Manager, the Object Factories (implementing the interface GenericFactory), the Fault Notifier and the Fault Monitors. These are implemented as separate CORBA Objects. The infrastructure also includes a logging and recovery mechanism built using service objects. Examples of these mechanisms are the two alternative versions described later in this section, the Logging and Recovery Controllers, and the Recovery Helper.

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 Recovery Controllers (one on every host on which group members are to be created and run). The deployment of the components on hosts was summarized in Figure 4.1. The Object Factory will create CORBA Object replicas, on demand, and will start them in different processes on that machine. Every replica of an object group is on a different host (and on that host 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. Thus, by implementing the PullMonitorable interface, application objects provide a method “is alive” that is periodically called by the fault detector. As soon as this method call cannot be completed (there is no answer from the server), or the method call returns a false value, the called object is considered as failed, and the failover mechanism has to be started. In order to assure that the failure is real, the object is “killed”. This means that the CORBA server object 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” happens in the same way, just that it might not have any effect. The Fault Detector reports the failure to the Fault Notifier. The latter sends fault notifications to its consumers (one of them has to be 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).
The gateway for active replication

In case of active replication, a gateway object is also created and started on a machine from the fault tolerance domain. In the current versions of the infrastructure, the object is located on one of the hosts where the actively replicated group members reside. The gateway has the role to broadcast incoming method calls to the group members. A duplicate suppression mechanism is needed in case 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. Thus, outgoing calls from the server group members, in case of active replication, are routed through the gateway, and the same thing happens to the reply to these calls. The gateway object is also a registered fault consumer, and it will receive fault notifications about failures of members of the respective active object group.

In this context, it has to be mentioned 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 recovery service objects

In a first implementation of the infrastructure, the logging and recovery mechanism on a host was implemented by using a separate CORBA object (called Logging and Recovery Controller). This object interface offers methods for logging method call and reply information. Also, it is possible to retrieve those informations later, at failover. In the improved version, the Logging and Recovery Controller object is replaced by a Recovery Helper CORBA Object that is used only during failover. This decision was made when considering an extended failure model including possible failures of infrastructure components, in particular the logging and recovery controllers' failure. The conclusion was that no separate object used for the logging of method calls and replies is actually needed. In this way, only the recovery helper was introduced to be used for retrieving state and update call information used in the failover of the new primary. In Section 4.5.5 more details can be found about the way logging is done in the improved infrastructure.

4.4.2 CORBA portable interceptors

There are some operations that have to be performed on the request either after it was "formulated" by the client, or before it reaches the servant object
4.4 INFRASTRUCTURE BUILDING BLOCKS

(the entity that executes the actual computation). These operations can be:

- adding extra informations to the request. These informations are recommended to be "carried" in service contexts. A client has to send over to the server the group version known to it. 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 reference obtained by the client contains such a version number that is or not the same as the one known by the server. Further, the client identifier, and retention identifier of a request, as well as a request expiration time, are also mentioned by the standard. The reason for having new types of request identifiers is that the classical (normal CORBA) request identifiers change for a resending of the same request. All this information is added, at the client side, in a client portable request 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 replication, no logging of method calls is needed. Only replies are logged in order to detect requests that are resent. In case of cold passive replication, method calls are logged only at the primary, as opposed to warm passive replication in which case calls are broadcast to the backups and logged there.

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

4.4.3 Extensions of ORB classes

When building the infrastructure, some ORB classes - such as the portable interceptors - were plugged in in an elegant way, and thus leaving the CORBA implementation 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" (standard) ORB classes had to be modified.

For example, because in the new setting, the multiple profile CORBA object reference might contain an indication about which is the primary member of a group, the part of the ORB where the target address is chosen, had to be modified. Also, some new classes had to be defined. In case a request has to be stopped in the server or client interceptor and not sent further, there must exist special exceptions that are thrown. Of course, an extension is also needed for handling these types of exceptions.

As already mentioned, the unique request identifier consisting of the retention identifier and client identifier 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 when this arrives at the interceptor level, and every time the request is resent this information should be the same. For this purpose, some ORB classes 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 IDL compiler was also extended.

4.5 Logging and recovery mechanism

On every host, there is a need (in some cases) for checkpointing (recording the state of) group members, as well as storing serviced method calls. All these data are stored for later failover purposes. The method calls for which information is important to be in the store at failover time, are those applied on the object since the last state was recorded. Moreover, these are state modifier (update) method calls.

4.5.1 Checkpointing decisions

In cases of cold and warm passive replication, from time to time there is a need to read the state of the object and store it in a log. Reading of the state has to be done when no update method call is performed on the object. This means, on the one hand, that as soon as the get.state method call has to be performed on the object, it has to wait until the current update method executing on the object finishes. On the other hand, 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 obvious problems arise: must the call to get.state be done periodically?
If this is the case, then how often? This is interesting because the average round-trip time for requests is influenced by the waiting time for get_state. Also, if the state does not change so often, maybe it is better to save the state as soon as it changes. Thus, in this case we talk about “event triggered” checkpointing. The evaluation results in Chapter 5 will provide some input for making these decisions.

Another issue is related to storing only state changes instead of whole state. This means that it might be useful in some cases to have implemented the Updateable interface, with the two methods get_update, set_update mentioned in the standard. So, the application writer would have to provide a method for returning the latest state change. This way of checkpointing can be very efficient especially when the state can become very big, but through small changes. However, there is a problem with this approach: in cases where the state of the object is complex (like a tree, or list) a change to such a structure can be very difficult to express. The reason is, of course, that the state changes have to be gathered so that they can be applied easily later on to the newly created primary (by calls to set_update).

### 4.5.2 Logging of method call information

As already mentioned, update method calls have to be logged at the application server, so that later at failover they can be replayed on the new primary. 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 that after a failure, a new primary is set up, and a read operation is resent by the client. Thus, the read method will be executed again and it is possible that it returns a different answer than first time. However, this is not a problem, because the client is interested anyway in the latest reading result.

For update methods there is a different situation. In an extreme case (considered in the present implementation), the order in which method calls are executed at different replicas e.g. primary, and later backup, matters. Therefore, the order in which calls are logged at the primary (this order corresponds with the one they will be replayed at the backup), has to be the same as the order in which they are executed on the primary. Because this is the case, and that logging is done separately from execution, there has to exist a mechanism of executing update method calls one by one. This means that independent of the way the ORB at the server side decides to handle client requests, the infrastructure (extended FT middleware) has to ensure that two requests cannot execute on the server concurrently. Of
course, if the ORB uses a single thread policy (no concurrent execution of requests will thus happen), then this precaution has no effect. In the other cases (ORB uses multiple threads to handle client requests), of course, the average round-trip time for an update method will grow with the average time taken to execute all update methods previously arrived at the server (see Chapter 5).

When logging method call information for replay purposes, it is important that the information is logged eventually, or a new state reading is recorded, leading to the removal of previously logged call information. Also, when the order of method call execution matters, the order of call information in the log has to be the same as the execution order, independent of the moment chosen for logging. There are two points where call information can be logged. These are the server side interception points at request reception and at reply sending. At present, the call information is recorded when the request is received, before it is executed in the application object. If, for any reason, the logging operation would not succeed, the call can be executed further, but then a new trial has to be made in the next interception point, i.e. at reply sending. If this action does not succeed either, then the call information has to be stored somewhere in memory for a short while, and then the logging has to be retried, maybe when the next request arrives. If, finally, the call information cannot be written to the log consistently, then it might be good to try to store the new state, so that call data is no longer needed. In this context, it is good to mention that there is a trade-off between doing checkpointing with a relatively low rate, and possibly have many update method call information recorded between two checkpoints. On the other hand, if state is logged every time it changes, i.e. after an update method call, next update calls have to wait until the state is read. This could increase even more the round-trip time for those requests.

4.5.3 Logging of reply information

Logging of method 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:

- the case of update method calls - the reason here is to avoid changing the state of the server twice, by reexecuting the method. Thus, even in case when the method call does not return a value, or even more, it is a one-way method call3, there has to be a note that the method was

3The sender does not wait for an answer for the call
before executing request on servant (at receive_request in server interceptor) & after executing request on servant (at send_reply in server interceptor) \\
| try to retrieve call reply | notify queued update methods |
| wait for update and get state methods to finish | log call reply information |
| log method call information | |

Table 4.1: Logging related operations in the server interceptor

- in case of read 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 the initial moment 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 probably more important than to get the answer back with less extra delay. Independent of whether read method 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).

### 4.5.4 Using reliable total order broadcast

In the first implementation of the infrastructure, no reliable broadcast is used. Total order is implemented by assigning unique identifiers to broadcast messages, by the sender. In this way, the receivers are sure that if they receive messages in ascending order of indices, then everyone will have messages in the same total order. This is the case for the gateway used in active replication. Also, the Logging and Recovery Controller object at the primary, in a warm passive replication setting, uses a total order broadcast of logging messages to the rest of the controllers.

In the current improved version, the gateway stays unchanged. Therefore, the broadcast of messages from the level of the gateway stays the same.

In the warm passive replication setting, on the other hand, the Logging and Recovery Controllers are no longer present. Now a total order reliable
broadcast is executed from the level of server interceptors of each replica. Thus, the primary broadcasts messages with unique identifiers, while the backups wait to receive those in correct order. Further, every backup resends the messages to all the other backups (reliable broadcast). In the presented setting all correct destinations will receive a message that was sent by a correct process. This is due to the fact that links are reliable, and, thus, no message once sent is lost. Furthermore, no message identifier is skipped, because the new primary takes on the state of the failed replica. This state includes the last message identifier as well. As a consequence, there is no danger of waiting forever for a message to fill the total order sequence.

4.5.5 Using stable storage

In the first implementation, replica state, as well as method calls and replies were stored in the memory of the separate Logging and Recovery CORBA Object. This approach is feasible for method calls and replies, and small state sizes. Also, it is feasible in a first simple model where it is supposed that the Logging and Recovery Controller Object stays up after the failure of the application object. As already mentioned, state changes (updates) can be considered to be (in most of the cases) small in size. So, it would be more efficient to store those instead of entire states. Still, a substantial effort is required to extend the application to provide the state change handling methods.

Therefore, (entire) state logging has to be focused on. When state size is very big, it might be necessary to use disk storage, because of its higher capacity. In this way the state information can be retrieved even after the crash of an entire host. Further, an interesting solution is to introduce CORBA transactions for accessing the possibly replicated stable storage. It seems feasible to use the same distributed database to log state information of members from several groups, running on several different hosts. There has to be a way of choosing hosts where to place the distributed database managers. Further, those database managers have to be extended to include transactional object behaviour as indicated in the CORBA Transaction specification. They have to implement the operations that need to be called on those objects by the transaction coordinator to orchestrate the commit of transactions.

These ideas were implemented in the improved version of the infrastructure. For cold passive replication style, stable storage is used to store state and update method call information. CORBA transactions are used to implement a replicated store consisting of simple files accessed by file op-
4.5 LOGGING AND RECOVERY MECHANISM

operations. At failover, one of these files is accessed by the Recovery Helper and state and method call information is read.

For warm passive replication style, no stable storage is used, because all information needed at failover (state and update method calls) is broadcast to the active backups as soon as it is read at the primary.

**Final remark**

In our knowledge, this is the first fault-tolerant CORBA infrastructure totally compliant with the CORBA standard. The next chapter will present evaluation results when performing measurements on an application running on top of it. The main goal of this work was to provide these results, in order to perform trade-off analysis.
Chapter 5

Evaluation of the Platform

This chapter describes the following aspects related to the evaluation of the fault-tolerant infrastructure presented in Chapter 4:

- measures of round-trip times for requests in case no failures occur;
- measures of failover times for the failure case;
- studies and results concerning the influence of different parameters on the timing values;
- feedback about the FT-CORBA standard itself, as well as arguments of whether such an approach to build a fault-tolerant middleware infrastructure is feasible or not.

5.1 Experiment setup

Earlier work in the area [25, 51], presented experimental results when using artificial applications and test cases. These are easy to set up, however, not always provide correct and general enough information. Also, infrastructure properties are quite easy to separate from application-induced ones in this case.

For the FT-CORBA infrastructure presented here, a realistic telecom application was used to perform most of the 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 (see Section 5.1.1 for a more detailed description). However, initial tests were performed by using a very simple made-up application.
The simple application consisted of a client calling the four operations of the server, in a loop. The functionality of the four methods is very basic. Round-trip time measures were obtained by averaging over the 200 iterations of the loop in the client.

In case of the realistic, more complex scenario, there were six methods available. Within a test case provided as part of the application, the server is accessed by one client calling three of the methods of the server, and several other clients calling other three methods. All these operations are performed in a loop. The tests were conducted for loops of 100, 200, 400 and 800 iterations. Four of the methods (method 1, method 2, method 3, and method 5) were called once every iteration. Method 4 was called 4 times every iteration, while method 6 was called 2 times every iteration. Therefore, averaging of round-trip time results was done over those numbers (multiplied by the number of iterations). The variation in number of iterations was applied in order to average out the effects of network load and other uncontrollable elements. The obtained overhead measure is, thus, an average value and it just gives indications about the extra time spent by the request on its way from client to server and back, while passing through an augmented middleware. The same is true for failover time values. These are very useful especially for comparison between several replication styles.

The parameters that were assumed to mostly influence the timing values are the replication style, group size, and checkpointing interval (for the primary-backup case). Therefore, experiments were conducted using different replication styles: cold passive and warm passive as well as active replication style. Because the application object can change its state by means of non-read operations (i.e., it is not stateless), the stateless passive replication style was not used. This was the case for both the artificial and the realistic application.

The group size ranged over 3, 4 and 5 replicas. Checkpointing was chosen to be done only periodically, with checkpointing intervals of 1s, 5s, 10s.

We performed our experiments using a fault tolerance domain containing seven SUN Ultra SPARC workstations, running SunOS 5.8. The machines were connected in a local area network, where we did not have control over the link traffic. Also, we did not control the processor load on those workstations, i.e. other applications than our experiments ran on them, as well. The infrastructure elements were deployed as follows: the Replication Manager was running on one of the computers, the Fault Notifier on a second one, while the Object Factories, Fault Monitors and Logging and Recovery Controllers were running on all of the five remaining machines. Replicas of one single group were created on 3, 4, respectively 5 of the five machines,
5.2 Round-trip measurements

Overhead values were obtained by measuring round-trip time of requests in the non-replicated case and in the replicated scenarios, and subtracting values in the former category from the ones in the latter. The round-trip time includes the time spent in the client interceptor (when adding information to the request), the time spent on traveling from the client side node to the server side node and until the request is taken for processing. Also, the time spent in the server side interceptor is traced, and the approximate computation time is counted from leaving the interceptor, until it is entered again when sending the reply to the client. Finally, the time taken to travel back from the server to the client node is traced. All time slices that are not overlapping\(^1\) are added together to give the round-trip time. Figure 5.1 shows the round-trip time slices in a generic scenario where a request follows a normal path from the client to the server and back.

Thus, measurement probes were placed in the client side interceptor, as well as in the server side interceptor. Furthermore, the time tracing code is placed in interceptor methods that will most probably be called at the client and server side. In order to avoid the probe effect, the same type of interceptors are used in the non-replicated case as well, but only to add and to extract the unique request identifiers (specified in the FT-CORBA standard), and to place the time tracing probes.

---

\(^1\)Overlapping time slices are only encountered when using active replication
roundtrip times: $t_1 + t_2 + t_3 + t_4 + t_5 + t_6$

$t_1$ - time spent in client interceptor at send request

$t_2$ - approximate time spent by the request travelling from client to server

$t_3$ - time spent in the server interceptor at receive request

(takes into account the time spent in logging method call information as well as the method synchronization time)

$t_4$ - approximate server processing time

$t_5$ - time spent in the server interceptor at send reply

$t_6$ - approximate time spent by the server reply travelling from the server to the client plus the time spent in the client interceptor at receive reply

Figure 5.1: Round-trip time slices - illustration
5.3 Expected results

This section describes the intuitions before running the experimental studies. These will be followed up by the real measurements in the following section. The number of replicas was expected not to affect overhead or failover times in case of cold passive replication. Of course, if the number of replicas drops under a certain minimum, new replicas have to be created and this can increase the failover time. In case of warm passive replication the expected result would have been that the round-trip time was slightly influenced by the number of replicas, because of the extra time spent in broadcasting method calls to the other replicas’ logging & recovery controllers. In fact, the broadcasting was implemented by use of CORBA one-way method calls. In a first implementation, no total order reliable broadcast was used. However, even in later versions where messages are delivered reliably and in the same order, the sender is not expected to be blocked by waiting for a reply, because one-way method calls are used in this case, as well. Hence, the overhead should not increase dramatically. Similarly, in case of active replication, it was expected that the number of replicas should not influence the round-trip time, since the gateway returns the first (fastest) result to the client.

The rate of checkpointing (inverse of the checkpointing interval) can influence the round-trip time for update operations. This is only relevant for cold and warm passive replication. As mentioned earlier, it is only in the passive style that the call to the checkpointing operation, get_state, is made.

Failover times are expected to be different for cold and warm passive replication as compared with active replication. This is due to the fact that in case of primary-backup replication, at failover, a number of non-read requests have to be replayed. Also, in case when a new replica has to be created for maintaining the minimum number of replicas in the group, the failover time is expected to be much larger than in the simple failover case. The latter is valid for all types of replication styles.

5.4 Results

This section describes experimental results, providing information about overhead percentages and failover times for different replication styles, group sizes, as well as different checkpointing intervals.
5.4.1 The artificial application

After running experiments involving the made-up application the overhead found was quite large. This happened when using the first non-optimized version of the infrastructure. For example, for one of the methods, the measured round-trip time for a non-replicated scenario was 24ms. The round-trip time when using cold-passive replication, a group of three replicas and checkpointing interval of 5s, was 46ms. Thus, the overhead was 22ms (92%). When looking closer, it can be realized that for a trivial application with low response time, showing a large overhead is inevitable.

It has to be noted that for the trivial server's case, the client is also very simple. Thus, the client calls the methods of the server one after the other, avoiding any concurrent calls incident on the server. Of course, there is competition when the checkpointing operation (call to get\_state) coincides with an update method call. In this case the update request is delayed until the get\_state is completed. Also, the server does not make outgoing calls as a result of client requests. This allows it to avoid the usage of the gateway as a duplicate suppressor, when active replication style is applied.

All these observations made clear that a more realistic and complex application is needed.

5.4.2 The telecom application

A shortcoming of the server structure is the existence, in some situations, of internal threads that can change the state of the server by using direct calls on the servant object. This behaviour can lead to state inconsistencies in case of cold or warm primary backup replication. The reason is that the non-CORBA method calls are not captured in the interceptor, and thus, they are not replayed at failover. Therefore, the state of the new primary will not correspond with the real state of the dead primary. Furthermore, the call to get\_state can only be blocked by an update method call that is captured in the server interceptor. In the test cases used in order to measure overheads, we made sure that these situations had not arisen.

Further, in some cases the server exhibits a client behaviour - it has outgoing calls. This implies that for the active replication style it will use the gateway's duplicate suppression mechanism.

5.4.3 Overhead values and failover times

In the realistic application the observed overhead percentages were lower in general (over all experiments), as compared to the ones in the artificial
5.4 Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Method 1</td>
<td>130ms</td>
<td>53%-66%</td>
<td>55%-79%</td>
<td>3400%-5000%</td>
<td></td>
</tr>
<tr>
<td>Method 2</td>
<td>61ms</td>
<td>110%-134%</td>
<td>77%-128%</td>
<td>770%-3360%</td>
<td></td>
</tr>
<tr>
<td>Method 3</td>
<td>65ms</td>
<td>74%-106%</td>
<td>62%-92%</td>
<td>270%-360%</td>
<td></td>
</tr>
<tr>
<td>Method 4</td>
<td>80ms</td>
<td>100%-151%</td>
<td>76%-168%</td>
<td>2100%-5100%</td>
<td></td>
</tr>
<tr>
<td>Method 5</td>
<td>133ms</td>
<td>59%-93%</td>
<td>44%-111%</td>
<td>1740%-4300%</td>
<td></td>
</tr>
<tr>
<td>Method 6</td>
<td>106ms</td>
<td>68%-94%</td>
<td>37%-98%</td>
<td>800%-3100%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Summary of overhead percentages

application. For example, for one method, the round-trip time in the non-replicated case was 106ms. In the replicated server scenario, on the other hand, using cold-passive replication style, a group of three replicas, and a checkpointing interval of 5s, the round-trip time measured was 156ms (47% overhead).

The time traces were analyzed and slices of the round-trip time were devised. In this way, it was possible to notice some parts that were dominant (occupied the largest percentage of the total time). For most of the server methods (5 out of 6) appearing in the test case, the largest slice encountered was the one corresponding to the wait time of an update method call. As mentioned earlier, update methods have to be executed one by one. Thus, the current update method incoming at the server has to wait until all update and get state methods that already arrived before it, finish their execution.

For the above mentioned example, the wait time was 74ms (out of 156ms), meaning 47% of the whole round-trip time. The other slices occupy much smaller percentages of the round-trip time.

A summary of the overhead percentages for cold and warm passive, as well as active replication can be found in Table 5.1.

The values in the first column are absolute values. They indicate the round-trip time values for every method, measured in milliseconds.

In case of cold and warm passive replication, each cell summarizes results obtained after experiments using three group sizes (3, 4, 5 replicas), and three checkpointing interval values (1s, 5s, 10s) - total of nine experiments. In case of active replication, each cell summarizes results obtained for three different group sizes (3, 4, 5 replicas). The round-trip time values used in Table 5.1 were obtained by averaging over a loop of 100 iterations (i.e. the values for method 1, 2, 3, and 5 over 100 calls, the values for method 4

\textsuperscript{2}A row corresponds to one method call
over 400 calls, and the values for method 6, over 200 calls). For comparison purposes, these values were enough, since the round-trip times when using the loops with 200, 400, 800 iterations, ranged around the same values.

As noticed in Table 5.1, the overhead in case of active replication is very large. This is the situation for the realistic application. In case of the artificial application, on the other hand, the overhead stays between acceptable ranges. For example, for one of the methods the round-trip time in the non-replicated case is 24ms, while in the presence of a replicated 3 member group server, the round-trip time is 66ms. The significant difference is given by the fact that for the artificial application there is no overhead associated with the gateway's duplicate suppression mechanism. Moreover, in case of the realistic application, some methods do not encounter the very large overheads of other methods (see method 3). This is due exactly to the fact that, e.g. method 3 does not call other servers' methods, so it does not use the gateway as a mediator object. These conclusions are drawn when looking at the approximate server processing time component of the round-trip time, in case of active replication. The values are predominant in all methods' cases, except for method 3.

The expanded table with round-trip time and wait time values\(^3\) (in parenthesis) for cold and warm passive replication styles is shown in Table 5.2. It can be noticed - by looking at values in cells on the same row and in columns corresponding to the same value of the checkpointing interval - that the difference between cold and warm passive replication under no-failure conditions is not so big. Also, the slight influence of the checkpointing interval on the wait time can be noticed (the larger the checkpointing interval, the smaller the wait time, in general). The bulk of the wait time is given by execution times of earlier arrived update method calls.

### 5.4.4 Failover times

Failover times are given as approximate time intervals. Thus, the traces contain information about when a failure notification arrived at the Replication Manager (some time after the failure was detected), and about the time the failover action of the Manager ended. The interval between these two times approximates the time that the application server is not available for processing requests. This time denotes the failover time of the server. Of course, the time spent by the Replication Manager recovering a new replica as primary (in case of e.g. primary-backup replication) includes the time taken to set the state of the replica, and to replay requests. Also, the failover

\(^3\)All values are given in milliseconds
<table>
<thead>
<tr>
<th></th>
<th>Cold Passive</th>
<th></th>
<th>Warm Passive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1s</td>
<td>5s</td>
<td>10s</td>
</tr>
<tr>
<td>201(31)</td>
<td>220(28)</td>
<td>211(21)</td>
<td>208(31)</td>
</tr>
<tr>
<td>215(35)</td>
<td>233(29)</td>
<td>221(26)</td>
<td>210(36)</td>
</tr>
<tr>
<td>208(26)</td>
<td>203(16)</td>
<td>206(29)</td>
<td>209(32)</td>
</tr>
<tr>
<td>116(54)</td>
<td>124(42)</td>
<td>115(40)</td>
<td>128(55)</td>
</tr>
<tr>
<td>138(57)</td>
<td>108(33)</td>
<td>127(50)</td>
<td>141(65)</td>
</tr>
<tr>
<td>139(60)</td>
<td>119(40)</td>
<td>127(49)</td>
<td>133(59)</td>
</tr>
<tr>
<td>125(47)</td>
<td>111(34)</td>
<td>115(35)</td>
<td>115(37)</td>
</tr>
<tr>
<td>117(39)</td>
<td>105(28)</td>
<td>110(29)</td>
<td>113(44)</td>
</tr>
<tr>
<td>108(41)</td>
<td>117(28)</td>
<td>108(37)</td>
<td>113(39)</td>
</tr>
<tr>
<td>174(108)</td>
<td>148(80)</td>
<td>148(87)</td>
<td>181(110)</td>
</tr>
<tr>
<td>214(140)</td>
<td>149(67)</td>
<td>152(89)</td>
<td>201(130)</td>
</tr>
<tr>
<td>193(129)</td>
<td>141(72)</td>
<td>160(97)</td>
<td>197(124)</td>
</tr>
<tr>
<td>234(104)</td>
<td>192(75)</td>
<td>200(89)</td>
<td>236(115)</td>
</tr>
<tr>
<td>280(156)</td>
<td>202(73)</td>
<td>213(95)</td>
<td>237(132)</td>
</tr>
<tr>
<td>245(134)</td>
<td>191(74)</td>
<td>211(100)</td>
<td>257(134)</td>
</tr>
<tr>
<td>178(99)</td>
<td>156(74)</td>
<td>156(76)</td>
<td>188(107)</td>
</tr>
<tr>
<td>210(126)</td>
<td>151(56)</td>
<td>164(87)</td>
<td>206(119)</td>
</tr>
<tr>
<td>194(115)</td>
<td>145(59)</td>
<td>175(98)</td>
<td>205(115)</td>
</tr>
</tbody>
</table>

Table 5.2: Round-trip time and wait time details (given in ms)
Table 5.3: Failover times (given in ms) for different replication styles

<table>
<thead>
<tr>
<th></th>
<th>Cold Passive</th>
<th>Warm Passive</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Checkpointing intervals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1s 5s 10s</td>
<td>1s 5s 10s</td>
<td></td>
</tr>
<tr>
<td>Failover Times</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 replicas</td>
<td>5040 7550 7800</td>
<td>2900 3500 5000</td>
<td>70</td>
</tr>
<tr>
<td>4 replicas</td>
<td>4100 7400 10700</td>
<td>1700 6600 18700</td>
<td>73</td>
</tr>
<tr>
<td>5 replicas</td>
<td>4100 3700 10700</td>
<td>1900 2500 7100</td>
<td>71</td>
</tr>
</tbody>
</table>

The number of replicas did not affect overheads in case of cold and warm passive replication. The overheads were pretty much the same for the two replication styles. In case of active replication, the group size mattered, as indicated by the large range for the overhead percentages. On the other
5.5 Lessons learnt

Average round-trip 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 mentioned earlier. The wait time for an update method call is built up by execution times of update operations.
Figure 5.2: Time for failover vs. number of requests to replay

Figure 5.3: Time for set state vs. state size
that arrived just before that method call and are executing or queuing on
the server.

Another aspect of interest is the synchronization of update method execu-
tions. In the present implementation the granularity of this synchroniza-
tion is at method level. Therefore, the server ORB has to know what the type
of a certain method is (update method, or reader method). Obviously, the
granularity of this part of the synchronization, also affects the round-trip
time of a method call. To reduce the round trip time one has to come up
with a way to capture the application writers’ knowledge with respect to
mutual exclusive accesses. This is a potential future optimization case.

In the present implementation, because of the way checkpointing of state
is done, there is no support for handling of application created threads that
can modify the object’s state by calling methods directly on the servant.
Such modifications are not reflected in the logged method calls, and thus
cannot be replayed once there is a crash. Therefore, using such design pat-
terns is not advisable in sensitive applications.

Further, 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 find the application server object crashed before the “official”
Failure Detector found and reported it: in case, e.g., when the monitoring
interval is relatively long (30s), and meanwhile the server thread crashes.
This is a problem in case of cold and warm passive replication. Thus, even
though the client cannot reach the server (primary) possibly because of 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 Replication Manager about the
failure, and the latter one modifies the monitoring intervals for the failure
detectors.

5.6 Why use an FT-CORBA infrastructure?

Building and testing the infrastructure has facilitated evaluating the concept
of FT-CORBA infrastructure 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 ORB to support fault tolerance the way the FT-CORBA standard devises it.

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 solution could be to equip the backups with the possibility to announce 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. In this context, it can also be mentioned that to implement unreliable failure detectors is not straightforward either. The reason is that the recovery from failure is handled by a separate unit (the Replication Manager) rather than the server group itself. Thus, 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, turning the detector into a perfect one. 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 as 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.

The most critical point is perhaps that all service building blocks of the 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 infras-
structure itself to replicate CORBA objects that are part of the infrastructure [51]. 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.

In the next chapter we present a way of dealing with the problem of infrastructure single points of failure, in an elegant way. A less rigid failure detection mechanism is introduced, that deals with failure suspicions, as well.
CHAPTER 5. EVALUATION OF THE PLATFORM
Chapter 6

An Alternative Infrastructure

As seen from the previous chapter, 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. As a second problem, the state consistency of the replicas is endangered by some application constructs such as internal threads. These can contain operations that change the state of the object without using interceptable CORBA method calls. Of course, this problem is relevant when checkpointing and method call information logging are needed, thus for passive replication styles. In this chapter we propose an approach for dealing with the first problem. Comments regarding the second problem will be given in Section 7.3.4.

Dutta et al. [22] describe an algorithm used by a server group when processing client requests, that is built to tolerate the existence of several primary processing nodes. This work does not depend on any particular middleware, but would be interesting to apply in a CORBA setting such as ours. The difference between the algorithm in [22] and active\(^1\) replication is that the servers can have non-deterministic behaviours. The existence of the mentioned internal application threads is similar to non-determinism, and so the mentioned shortcoming of the FT-CORBA infrastructure seems to be removed by using this algorithm.

The algorithm is different from active replication, in another way, too. There is no parallel independent processing of requests after agreeing on

\(^1\)state machine
their total order, but 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 maybe obtaining two different results. Also, the order of request processing is not agreed upon. The result is finally 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.

In this context, there are no additional infrastructure units handling fault detection and failover. The server replicas are equipped themselves with unreliable failure detectors (recall from Section 2.1.3), that are queried from time to time about the identity of a correct process that can be the “leader” that processes requests. However, as the failure detections are imperfect, there may be two or more servers that consider themselves to be leaders at a point in time.

We call this algorithm robust in the sense that it can deal with failures of servers, as well as of other processing units including failure detectors, in a robust way. The algorithm is independent of an underlying platform, and has so far been presented as a pseudocode. This chapter presents a CORBA based implementation of the robust algorithm. The concept of infrastructure is kept, i.e. offering transparency, and letting the application writer to concentrate on application functionality, and extend the code with very few elements. Measurement results will be listed and a comparison with the standard infrastructure will be presented based on similar experiments.

6.1 Description of the robust algorithm

The system model used by Dutta et al. [22] is as follows: the distributed system is represented by a set of processes, that can fail by crashing. After crashing, a process does not recover. Links, in this model, are reliable. Thus, there is no duplication of messages, no spurious messages, and any message sent on a link is eventually received by its destination, if this one is correct (not failed). It is supposed that at least a majority of processes are correct, being thus able to participate in decision rounds.

Application servers are replicated for fault tolerance. These replicas are the processes participating in the robust consensus algorithm. The scenario is set up so that there are clients sending requests to a server group. The group should have one (unique) leader, i.e. one process in charge of servicing the client requests. The interesting point, and the reason why a consensus algorithm is needed, is the possibility for two or more processes to “believe” that they are the leader. Furthermore, server behaviour is non-deterministic.
This means that it is not enough for replicas to agree on the total order of processing of requests. Thus, there exists the possibility that several replicas will service requests, in possibly different orders, or will service the same request, but obtain different results.

As mentioned in Section 2.1.3, it was proven by Chandra and Toueg [14] that the weakest failure detector (called eventually weak) is enough for solving the consensus problem. This property is exploited when devising the present consensus algorithm. Two primitives are involved: eventual leader, and eventual consensus. The final goal is to have an algorithm that has the properties of consensus. None of the two primitives has these properties, but using them together, makes it possible to solve consensus. Furthermore, it is an advantage to have primitives of smaller granularity.

The eventual consensus has three properties: agreement meaning that no two processes will decide differently, validity, meaning that if one process decides a value, that value was proposed by some process, and termination, meaning that every process that proposes either crashes or decides a value; furthermore, if a single process proposes an infinite number of times, it eventually decides. It has to be noted that the termination property is weaker than the same property for consensus. The difference is given by the possibility of a process to propose infinitely many times and eventually decide once.

The eventual leader has also three properties, with names identical to the eventual consensus' properties, but with different meanings. Thus, here validity means that eventually the leader (as seen by the rest of processes) is correct, agreement means that eventually all correct processes see a unique leader, and termination means that a correct process cannot be blocked forever when it performs the leader election operation (i.e. it wants to find out the identity of the leader).

6.1.1 The algorithm - client side behaviour

The client is equipped with a leader election unit. This is used to find out who is the current leader in the server group, so that the client sends its request for processing to the right replica. If the presumed leader crashes or becomes too slow after the request of the client is sent, and the reply does not come back within a certain time, the client resends the request to a new presumed leader (lines 3,4). See Figure 6.1 for the pseudocode description, cited from [22].

By the properties of the eventual leader primitive and the eventual consensus primitive, the client is guaranteed to eventually receive the expected
0: submit(Request req) {
1:     while (true) {
2:         set timer
3:         send([Request, req] to the current leader
4:         wait until received [Reply, res] or timer expires
5:         if received [Reply, res]
6:             return res
7:     }
}

Figure 6.1: Client side behaviour

answer. This will be unique, even in case two replicas produce different answers, because it is returned by the server group after running the above mentioned consensus algorithm.

6.1.2 The algorithm - server side behaviour

Each replica member of the server group has the potential to process incoming requests from the client. This is because it has its local copy of the application object. In the best case, clients send their requests to one and the same replica, seen as the leader. In a not so good case, two (or more) clients can send their requests to different perceived leaders. An even worse case is when a client has to send its request twice or more times and to different leaders. In any case, the replicas not directly applying the request on the application object (i.e. those that do not receive that request, on a send occasion), will only witness the update (if any) enforced by the request execution on the application server state. Witnessing means that the outcome (state update plus result) of the request is written in a certain position in a register, upon receiving it from the leader replica. Every server replica, including the leader itself is equipped with such a register. It contains a set of application object state changes enforced by request executions, together with the return values of those executions. The position mentioned is called the position in the total order of state updates of the server group and every replica has its local view (\( \text{num} \) in the pseudocode in Figure 6.2) of the number of this position. Total order of updates means that all replicas have identical content of their registers. A witness does not change the state of its local copy of the application object, when filling the register with an update, neither does it increase the value of \( \text{num} \). Beside the register, every replica has a store of replies for already executed requests. The store is used only
6.1 DESCRIPTION OF THE ROBUST ALGORITHM

Request req
Reply res ← nil
Update upd ← nil
Leader leader ← new Leader
Decision decision ← nil
Integer num ← 1
Register register ← new Register

1: upon receive [Request, req] from c do
2:  while true do
3:    if store.isCommitted(req, num) = [true, decision]
4:      send[Reply, decision.res] to c
5:      break
6:  [res, upd] = execute(req)
7:  prop ← [req, res, upd]
8:  decision = abort
9:  while decision = abort and leader.elect() = p_i do
10:    decision ← register.propose(num, prop)
11:    if decision = abort then break
12:  store.setCommitted(num, decision)
13:  num ← num + 1
14:  O.update(decision, upd)
15:  if decision = prop then
16:    send[Reply, decision.res] to c
17:    break

Figure 6.2: Server side algorithm pseudocode executed by process p_i

when a replica receives a request, i.e. when it is seen as a leader by at least
the client. It will check the store in order to detect requests that already
have their answer there, and will not reexecute them.

The following paragraphs will describe the steps executed by a server
replica, from that replica’s point of view, as soon as it receives a request
from the client. The pseudocode form of the server side algorithm, cited
from [22], is given in Figure 6.2 in order to help the reader, when references
to lines in that code are made. One aspect has to be clarified here: any
replica that receives a request from the client is a possible leader. Every
server replica is equipped with an unreliable failure detector, as part of the
leader election unit. When queried about the identity of the current leader,
the leader election unit will return the process with the lowest index out of
a set of processes that are seen to be correct. Of course, the process making
the query will be among those processes, but maybe it will not have the
lowest index.

After a request (req in the pseudocode) is received from the client, the
store is checked (line 3). If the result (decision.res) is in the store, it is
sent to the client (line 4). If not, the request is tentatively executed on the application object (line 6). Tentatively means that no durable update to the state of the object is made yet. Meanwhile, the update (if any - upd in the pseudocode), together with the return value (if any - res in the pseudocode) of the request execution are obtained (line 6). The consensus algorithm is executed, only if the process sees itself as a leader\(^2\) (lines 9,10).

The following three paragraphs will describe situations that motivate the use of consensus. In the first two cases, the agreement mainly helps filling in the registers in a consistent way, to respect the total order requirement. The third situation illustrates the need to use consensus to recreate the leader's state on failover.

**Case 1:** if two server replicas received the same request (i.e. the client was obliged to send the request twice to the server group for reasons not important here), since server replicas have non-deterministic behaviours, then it can happen that the two end up having different states, or obtaining different results after executing the request. This will be avoided by using consensus.

**Case 2:** if two server replicas receive two different requests from two different clients, the order in which the updates are applied to the server replicas' state can differ. It can also happen, that each of the two server replicas apply only the update given by the execution of the request it received. Note that “order of updates” was mentioned. We consider, that the order in which requests are executed matters, or that actually the order in which two state changes are applied has to be the same in all replicas, in order to keep states consistent. This scenario can be generalized to more than two clients, and should also be avoided by consensus.

**Case 3:** if a server replica suddenly becomes the leader from some client's point of view, and it receives a request from that client, then it can happen that this request enforces an update on the non-up-to-date server replica state. The update has to be written in the distributed register, exactly after the last one. However, the current value of the replica's local variable indicating the next free position in the register (num), does not coincide with the actual value of this. Therefore, the server will be trying to write in a position that is already occupied. This situation should be avoided, and it will be done using the propose primitive.

The next three paragraphs will describe the way the above problematic situations are solved:

\(^2\)That is, in the set of correct processes returned by its leader election unit, the querying process is the one with the lowest index
two (or more) leaders writing the outcome of the same request to possibly the same position in the witness and self registers. This position in the register is known by every replica by consulting the local variable num\(^3\). The non-winner leader will modify its local variable so that next time, if it has to propose an outcome for possibly a different request, it will try to put it in a potentially free position.

In case 2, the consensus solves again a competition, but in this case, two positions in the registers have to be filled with outcomes. One of the leaders will "win" the writing right to the first position. As a consequence, the other has to change its state according to the update written by the winner, and then execute the request that itself received, and finally write the update caused by this execution, in the second free position in the registers.

In case 3, the consensus helps the new leader, to read the values from the register, while updating its local variable indicating the free position (num in the pseudocode), until it reaches the right value. Also, it changes the state of the replica according to the updates read. Agreement is needed due to the fact that this leader might not be unique while updating.

In a nutshell, the algorithm can be summarized as follows:

As long as the client and the server group members manage to see the same replica as the leader, the agreement algorithm is executed once for every request sent during this time. The registers of all witness replicas as well as the leader replica are filled up with outcome values, and the local variable of the leader is modified so that it indicates the real free position in the register.

As soon as the current leader crashes and a request is sent to a new presumed leader, the agreement algorithm can be executed more than once per request, and it provides help in getting replica states consistent. The same situation can arise in case the current leader is only suspected to have crashed, several requests being sent to other server replica(s). This second possibility relates to situation a) when the client, seeing a different process as leader, sends the request there. The problem here can be that only the client sees that replica as leader, while the server group itself has a different view. Thus, from the request receiving replica's point of view, the leader can be someone else. Therefore, it will only execute tentatively the request, but it will not initiate the agreement algorithm, and it will not answer the client. As a consequence, the client will be timed out again, and it will have

\(^3\)Note that this variable can have different values from replica to replica, since the witnesses do not update it. Only the server replica that has its proposed outcome championed will put it in the registers in the position given by its local variable, followed by incrementing this variable.
to make a new trial.

We now proceed by reviewing the pseudocode in detail. In order to resolve the cases above in the desired ways, the \textit{propose} primitive used to perform the agreement, contains two phases (see Figure 6.3 for the pseudocode description of the \textit{readWrite} and \textit{propose} primitives respectively, cited from \cite{22}):

- In the read phase, the leader tries to find out if the position (given by the value of its local variable \texttt{num}) in the register is already written or even read by some other process. This operation is done by sending a \texttt{READ} message to all replicas (witnesses and leaders) - see line 2. Then the sending leader waits for a majority of answers (line 3). On the other side, upon receiving a \texttt{READ} message, two types of answers can be sent: either an answer acknowledging the present leader, or an answer that can make the leader abort the whole agreement algorithm (not-acknowledge message). The latter occurs when the considered position in the register of one of the witness processes was written or even read by a process that sent a \texttt{READ} or \texttt{WRITE} message with a round number (\texttt{k} in the pseudocode of the \textit{propose} and \textit{readWrite} primitive) higher than that of the current sender. The former occurs in all other cases. Note that even if the leader is acknowledged it is possible that some other process was there reading or even writing the value (in this latter case, the written value is also sent together with the acknowledge answer). So, the waiting leader, upon receiving the majority of answers, it can act in two ways: if all answers are of type acknowledge, then it will proceed to the next phase. If at least one of the answers is not-acknowledging, then it will abort the operation, and start all over again (see line 9 in Figure 6.2). Of course, this will happen only if it is still a leader.

- In the write phase that takes place only if the read phase did not lead to an abort, the leader tries to write the value it proposed, or read from the registers of its witnesses, to the approved free position in those registers. Note that the so-called approved position coincides with the value of the process' local variable \texttt{num}. The write operation can also end with an abort, for the same reasons as the read operation: one witness process received a \texttt{WRITE} request to the same position, with a round number larger than the one currently sent. In such a case the while loop at line 9 in Figure 6.2 will be entered again, if the process is still the leader.

The message flow between the parties involved in the request sending
At process $p_i$
class Register
RRegister register $\leftarrow$ new RRegister
Integer $k \leftarrow 1$–$n$
Outcome propose(Values $v$)
\[ k \leftarrow k+n \]
return (register.readWrite($k,v$))

At process $p_i$
class RRegister

Integer $read_i \leftarrow 0$
Integer $write_i \leftarrow 0$
Values $value_i \leftarrow \bot$
Values $v* \leftarrow \text{perp}$

1: Outcome readWrite(Integer $k$, Values prop)
2: send [READ,$k$] to all processes
3: wait until receive [ackREAD,$v_j,ts_j,k$] or [nackREAD,$k$] from $\lceil \frac{n-1}{2} \rceil$ processes
4: if received at least one [nackREAD,$k$] then
5: decision $\leftarrow$ abort
6: else
7: select the [ackREAD,$v_j,ts_j,k$] with the highest $ts_j$
8: $v* \leftarrow v_j$
9: if $v* = \bot$ then
10: $v* \leftarrow \text{prop}$
11: send [WRITE,$v*,k$] to all processes
12: wait until receive [ackWRITE,k] or [nackWRITE,k] from $\lceil \frac{n-1}{2} \rceil$ processes
13: if received at least one [nackWRITE,k] then
14: decision $\leftarrow$ abort
15: else
16: decision $\leftarrow v*$
17: return decision

18: upon receive [READ,$k$] from $p_j$ do
19: if $write_i > k$ or $read_i > k$
20: send [nackREAD,$k$] to $p_j$
21: else
22: $read_i \leftarrow k$
23: send [ackREAD,$value_i,write_i,k$] to $p_j$

24: upon receive [WRITE,$v_j,k$] from $p_j$ do
25: if $write_i > k$ or $read_i > k$
26: send [nackWRITE,k] to $p_j$
27: else
28: $write_i \leftarrow k$
29: $value_i \leftarrow v_j$
30: send [ackWRITE,k] to $p_j$

Figure 6.3: Consensus primitives propose and readWrite
and answering scenarios is depicted in figures 6.4, 6.5, and 6.6.

Figure 6.4 shows the message flow in a normal (failure free) run of the client-server and server-server communication.

Figure 6.5 illustrates one type of failure scenario, when the client has to resend its request to the server group after being timed out in waiting for a reply. The server replica being seen as the leader by the client does not manage to send the answer to the client on time. The client has to send the request again, and by this time it sees a different process as the leader. It happens, as the chart illustrates, that the other replicas (including that process itself) in the group see the same member as the leader. This can be the case both when the previous leader really crashed or it became very slow. Also, the situation is such that the previous leader managed to read the register, but not write it.

The chart in Figure 6.6 describes the scenario where the first leader becomes too slow, but only from the client's point of view, and only for a while. Thus, when resending the request for the first time, the new receiver replica will not send the answer back, since it does not recognize itself as leader. The client will be timed out again, therefore resends its request now to the original replica. And the answer is finally received in time from this one.

After the execution of the agreement algorithm, if the process is no longer the leader, it will stop the execution in this point\(^4\) (line 11 in Figure 6.2).

\(^4\)Note that to reach line 11, either decision has to be not abort or the process stops
6.1 DESCRIPTION OF THE ROBUST ALGORITHM

Otherwise, it will proceed to change its state according to the decided update, and store the result in the local store. The outcome of the agreement algorithm (i.e. application object state change and request execution result) can be the same as the one proposed by the leader. In this case the reply will be sent to the client. Otherwise, the leader will return to the beginning of the loop (line 2 in Figure 6.2) where the effect of the execution of the currently received request has to be completed, i.e. it will try to send the reply of the current request to the client. This situation can arise repeatedly during one request execution, thus leading to multiple walks through the while loop. It is a typical behaviour of a replica that is taking over. Let us review a possible concrete scenario:

The leader just crashed after it executed 15 requests, thus changing the state of the application object 15 times. A client, discovering the identity of the new leader sends request number 16 (the one that will cause the 16th state change) to this. Say, that this new leader is in its initial state, therefore believing that the first free position in the registers is number 1. Upon the receipt of the request it will start the while loop.
Figure 6.6: Client-Server message sequence chart when the leader becomes too slow(2)
• First iteration - it checks, as usual, the store with replies, but obviously will not find the reply there. As a consequence it will execute tentatively the request on the object, obtaining an update and a result. Next, it tries to write this values in position 1 in the registers. After several turns in the while in line 9, when the status is no longer abort, the content of position 1 in the register is obtained. The new leader changes its state accordingly, remembers the request result in the store, increments the variable containing the local view on the first free position in the register (num becomes 2) and wants to send the reply to the client. Since the value read from the register is not the same as the one proposed by the new leader, it will not send anything to the client, but will return to the beginning of the while loop in line 2.

• Second iteration - same steps as in the previous one are performed, except that now the content of position 2 will be read and num will become 3. The state of the object is modified with a new increment.

• .................

• Fifteenth iteration - now num is 15; this is the last written position in the registers. The update read is applied on the object. Now the object has an up-to-date state, i.e. the one of the former leader.

In iteration number 16, the request is again tentatively executed, but now the outcome proposed by this replica for position number 16 is the first one to be written there. Thus, now the replica will send the correct reply to the client to request number 16.

Note that the request was tentatively executed 16 times, this being really necessary only once (iteration 16). We will come back to this later in the evaluation section when discussing failover times.

6.2 The CORBA based implementation

The algorithm discussed in the previous section is implemented by reusing some of the elements introduced with the FT-CORBA standard infrastructure. Thus, portable interceptors are used again. Also, client requests have attached a unique identifier that is the pair (retention_id, client_id). Furthermore, the get_state and set_state methods are used again, even if not in the same way as in the FT-CORBA case. In addition, in this new setting,
Chapter 6. An Alternative Infrastructure

The get_update and set_update methods are implemented. ORB classes have to be modified further, and some of the old modifications can be reused.

The architecture of the new infrastructure can be seen in Figure 6.7. We call this infrastructure universal construction based, due to the original title of the Dutta et al. paper.

The acronyms in Figure 6.7 have the following meanings:

- **LEU** - Leader Election Unit
- **CO** - Consensus Object
- **ASO** - Application Server Object
- **ACO** - Application Client Object
- **UC-ORB** - ORB extended for fault tolerance by using the robust algorithm (universal construction)
6.2 The CORBA-Based Implementation

The dotted line boxes containing ASO suggest that the server interceptors access the application object and call methods on it from their level. The dashed arrows that go between two consensus object boxes, or two leader election unit boxes, represent message flow by using simple socket connections, not involving CORBA calls. The dashed arrows that appear inside a “host” box, suggest internal operations within a process (e.g. in the client box, the dashed arrow represents a portion of the flow of a method call from the client to the server, that spans from the ORB to the client interceptor). The solid arrows represent flow of CORBA calls or other method calls (e.g. leader() or propose() ) that are not called via CORBA, but are done from one box to another box.

The scenario captured in Figure 6.7 is the desired one when clients send their requests to one single process that is the leader. Several failure scenarios could be drawn: instead of having one box with “Leader” to have two of them, and one client sending to one such process, and the other to the second one. Also, it is possible that the client sees as “Leader” box number 2, while this one sees as leader box number 1.

The weak failure detector is set up as a separate server thread, attached to the application CORBA object. Further, this failure detector server is part of a leader election unit, that is supposed to receive I-am-alive messages from other leader election units at other application replicas. Note that it can be assumed that the leader election unit (and thus failure detection unit) sends “true” I-am-alive messages on behalf of the application server replica. Thus, the non receiving of a I-am-alive message denotes the non-availability of the application server, although the two units are separated. When trying to introduce the outcome of a request processing in the total order, the leader election unit is queried for the present leader. This will return the process with the lowest index that is correct. The I-am-alive messages are sent as simple datagrams and as soon as such a datagram is received at the server, a timer is reset.

The client, in order to enable its leader election unit to receive I-am-alive messages from server replicas, used to determine who the leader is, has to “register” with the server group. This is done by sending an I-am-in message to all members of the replica group. This message has piggy-backed the address in the client where the I-am-alive messages have to be sent, i.e. the address of the client’s leader election unit. This registration system is also used in case when a server replica needs to use a third tier server, i.e. the server replica becomes a client. When initializing a server replica using the infrastructure, a list of third tier servers is parsed, and if there are any, the server replica registers itself as a client at each of them.
The only references (addresses) that are published are those of the application server replicas. Therefore, the I-am-in messages are sent via CORBA dynamic request invocations to these addresses. Further, the leader election unit information sent with these messages is communicated at the request receive interception point to the same unit at the server side. The sending of I-am-alive and other infrastructure related messages is done by using simple datagram socket connections, and not involving CORBA specific overhead.

6.2.1 Using portable interceptors

In the new infrastructure the role of client side portable interceptors is similar to the one in the FT-CORBA infrastructure: adding unique request identification information to the client request. Group reference version information is not relevant for this new setting.

Portable server side interceptors are used mainly for two operations:

- to intercept the I-am-in registration messages from clients, and to inform the appropriate leader election unit about the addresses where the I-am-alive messages have to be sent later;

- to intercept client requests and perform all the operations devised in the algorithm.

The tentative execution of the request on the server replica is simulated by three steps:

1. the state of the replica is read by using get\_state (1);

2. the method call is executed on the server replica, so that a result, if any, is returned and the state of the replica is changed;

3. the change of the object's state is read by using the call to get\_update.

Note that in our implementation, in case the request is read-only, the server replica does not execute the agreement algorithm. After the request execution (if this is needed) it sends the reply to the client and stops, returning to the point where it waits for a new request to arrive.

In case the process proposing the request outcome stops being a leader, then the state read at (1) will be restored on the replica by using set\_state. This also happens if some other leader's update has to be enforced on this replica. First the state is restored and then the state change is applied on the object.
6.2.2 Modifications of ORB classes

ORB classes had to be modified or extended, like for the FT-CORBA infrastructure. Some old modifications could be kept, but some had to be replaced. This is the case, e.g., of the code in which the address where the request is sent (target address) is chosen. 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 by extracting port and host information from the IOR. In the present setting, the modification consisted of introducing the leader election algorithm execution as the part returning the target server address.

Furthermore, the client side ORB part had to be modified to handle the timeout in a transparent way. The timeout is used when setting the timer (see line 2 in the client behaviour pseudo code). In the ORB, there is a routine that handles the case when a reply does not return before a certain timeout (if such a timeout is used), but it only throws an exception that is caught at the client application level. We did not want this to happen, so instead of throwing this exception, we had to resend the request in a transparent way following the client-side behaviour of the algorithm.
CHAPTER6. AN ALTERNATIVE INFRASTRUCTURE
Chapter 7

Evaluation of the Robust Algorithm

This chapter will present results obtained after conducting experiments with the CORBA infrastructure using the robust algorithm. The text essentially follows the structure in Chapter 5 on evaluation of the standard FT-CORBA platform. Finally, a comparison between the two infrastructures is also presented.

7.1 Experiment setup

The setup used for performing our experiments was chosen in the same way as for the standard platform. Namely, measurements were done first on the trivial client-trivial server type application. Round-trip times were averaged over a number of 200 iterations. When using the real telecom application, round-trip time values were obtained also as averages, over a number of 100, 200, 400 iterations. Some methods were called once per iteration, others two or four times. The no failure case was the basis of round-trip time measurements. The request pattern was kept identical to the previous experiments. Of course, reference values for the round-trip times were obtained from measurements in a scenario where clients use the services of a non-replicated server application.

To measure failover times, a failure scenario was considered. On failure of the current leader, a new replica was elected as leader and forced to bring its state up to date as soon as a first query arrived. Experiments were conducted for different numbers of updates to be set in the recovering leader.
The dependence of the failover time on these numbers will be highlighted in this chapter.

In case of the present infrastructure, the parameters supposed to influence the obtained results were:

- the number of replicas in the group - both for round-trip times and failover times. The group size ranged over 3, 4, and 5 replicas.

- the rate of incoming requests - mostly for failover times.

Note that the rate of incoming requests had an influence on failover times in the first infrastructure as well. The difference is that in case of the robust algorithm, requests are not actually replayed, but their result, if any, and update are read from a majority of replicas. Further, the update must be set in the failing-over leader.

The experiments were run on similar machines connected in the same way as in case of the FT-CORBA standard compliant infrastructure.

### 7.2 Expected results

This section describes what our expectations were before doing any measurements, about how the different parameters will influence timing values. The next section will provide quantitative measures about how big this influence is.

As already mentioned, in this setting, the group size is expected to very much affect round-trip times, as well as failover times. Basically, for both cases, there is a reading and writing phase from and to a majority of processes. Thus, the bigger the number of processes, the longer the time spent in communicating with their majority.

Also, the time spent in getting and setting the state of the replica is expected to influence round-trip times, and especially failover times. Therefore, in an indirect fashion, the state size should also influence the timing values. The failover times are expected to be more sensitive to the state size because, during failover of the new leader, the state reading and writing happens more often than in case of the normal request processing phase in the current leader.

As for the FT-CORBA infrastructure, the rate of incoming requests can influence the failover time, because of the number of requests that have to be replayed on the new leader. In case of the present infrastructure, however, every new leader has to bring its state up to date by applying the effect of all requests executed so far by the server group. “All” is mentioned here
7.3 Results

This section presents the quantitative values for round-trip times and failover times obtained after performing the experiments described earlier. Also, it will report on memory, or disk space usage for storing the outcomes of requests. The storage aspect becomes significant in this case. It has to be noted, that when performing the experiments on the telecom application, with 800 loop iterations, the storage got full before the end of the test. This is the reason why results were only considered for the case of 100, 200 and 400 iterations.

As in the case of the FT-CORBA platform, the results are average values, mostly used for comparison purposes. Thus, they are indicative of a trade-off, but cannot be relied upon as a kind of profiling information. For such a quantitative measure other contributing factors need to be known in detail such as processor load and link traffic.

7.3.1 The artificial application

In case of the trivial application, after running the experiments with 200 loop iterations, the overheads came out to be pretty large (as for the standard platform). The values for overheads ranged between 200%-268%. We could notice that the main component of the round-trip time, and also overhead, was the time taken by the consensus. 58 to 64% of the total round-trip time was represented by this part. The wait time given by the method call synchronization is 0 in this case, because methods are called in a sequence and so they do not compete at the server.

Next, we performed failover experiments. The results shown in Table 7.1 will be used for comparison with the real-life application. The values in the first column indicate the number of updates that had to be set before the new recovering leader can start processing requests. The values in the columns on the right are given in milliseconds and correspond to the failover
Chapter 7. Evaluation of the Robust Algorithm

Table 7.1: Failover times for the trivial application (given in ms)

<table>
<thead>
<tr>
<th>replayed reqs.</th>
<th>3 replicas</th>
<th>4 replicas</th>
<th>5 replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>9000</td>
<td>8400</td>
<td>8500</td>
</tr>
<tr>
<td>150</td>
<td>13500</td>
<td>19900</td>
<td>15800</td>
</tr>
<tr>
<td>225</td>
<td>23700</td>
<td>26000</td>
<td>23800</td>
</tr>
</tbody>
</table>

times in case of a groups of 3, 4 and 5 replicas. It can be noticed that the failover time values grow almost proportional with the number of updates to be set in the failover replica, and almost unchanged with variations in number of replicas.

As in case of the previous platform, we chose to run the telecom application as well, in order to observe timing values at a bigger scale.

Sections 7.3.2 and 7.3.3 will give details of no-failure and failure scenarios respectively.

7.3.2 Telecom application - overheads

The round-trip times for the non-replicated case, as well as overhead percentages for different group sizes, in the replicated server scenario, are presented in Table 7.2. It has to be noted that the same test case was used as for the standard FT-CORBA infrastructure. Therefore, the method calls for which the timing values are given, are the same as in the first presentation.

The first column in Table 7.2 contains round-trip time values when experimenting with the non-replicated server. The timing values are given in milliseconds. For every method call, three values are given in the first column and in the three columns on the right. The three values correspond to results obtained when averaging over 100, 200, 400 iterations respectively. Here, we present them separately (as opposed to the first platform), because we found big differences in overhead and round-trip time values when the number of calls increases during the experiment. The reason for the differences noticed can be explained as follows: for the client request interarrival time imposed in the test case, requests are processed at the server side such that the wait (method synchronization) time increases very much for requests that arrive later. Therefore, the average result will be much bigger for numbers of iterations 200, 400 as compared with 100, since the round-trip times for requests in the "middle" are more or less the same. The differences can be noted in the three rows corresponding one to each method. It is clear that higher
number of iterations leads to higher overheads monotonically. The three columns on the right correspond to the three group sizes used (3 replicas, 4 replicas, and 5 replicas, respectively).

As the results show, the overheads are pretty large. The dominant part of the round-trip time in case of the replicated server is given by the time taken for synchronization of method executions. As in case of the FT-CORBA platform, for the same reason, methods have to be executed by the leader, in a serialized manner.

The overheads are larger in this case, because the time spent in executing one single request is larger than before, due to the consensus part, executed every time a request is processed. As our measurements show, this part occupies also a significant percentage of the round-trip time. We can notice that, even if the overhead does not always grow with the number of replicas, the time spent in executing the consensus grows slightly with the number of

<table>
<thead>
<tr>
<th>Method</th>
<th>nr.iterations</th>
<th>Non repl</th>
<th>3 replicas</th>
<th>4 replicas</th>
<th>5 replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>221ms</td>
<td>89%</td>
<td>124%</td>
<td>139%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>182ms</td>
<td>156%</td>
<td>185%</td>
<td>203%</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>156ms</td>
<td>244%</td>
<td>294%</td>
<td>319%</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>95ms</td>
<td>223%</td>
<td>253%</td>
<td>287%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>84ms</td>
<td>314%</td>
<td>389%</td>
<td>415%</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>96ms</td>
<td>361%</td>
<td>433%</td>
<td>456%</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>91ms</td>
<td>200%</td>
<td>240%</td>
<td>285%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>69ms</td>
<td>365%</td>
<td>446%</td>
<td>492%</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>75ms</td>
<td>472%</td>
<td>533%</td>
<td>594%</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>174ms</td>
<td>104%</td>
<td>137%</td>
<td>200%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>103ms</td>
<td>355%</td>
<td>400%</td>
<td>464%</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>103ms</td>
<td>440%</td>
<td>510%</td>
<td>580%</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>208ms</td>
<td>120%</td>
<td>166%</td>
<td>232%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>169ms</td>
<td>265%</td>
<td>287%</td>
<td>343%</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>162ms</td>
<td>345%</td>
<td>388%</td>
<td>432%</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>183ms</td>
<td>134%</td>
<td>164%</td>
<td>228%</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>126ms</td>
<td>331%</td>
<td>378%</td>
<td>423%</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td>121ms</td>
<td>427%</td>
<td>483%</td>
<td>554%</td>
</tr>
</tbody>
</table>

Table 7.2: Summary of overhead %s in the alternative infrastructure
replicas. Table 7.3 shows the dependence of the time spent in the consensus on the number of replicas in the group. Results are shown for the same set of experiments that were involved in the round-trip measurements. The values given in parenthesis indicate the percentage of the total round-trip time occupied by the consensus component. It can be noticed, that these values stay almost constant for a certain method call. The timing values, as well as the percentages are given as averages of the values obtained when experimenting with loops of 100, 200 and 400 iterations\(^1\).

The very first experimental results (not described in this work) were obtained when using get\_state both for reading the state before the tentative execution, and after it, as part of the request outcome. This alternative turned out not to be efficient, while reading only state changes (using the call to method get\_update) seemed better and more natural. Thus, the register with request outcomes stores for every request, the state change, instead of entire states. The numbers presented in the tables in the result sections reflect the timing behaviour of the infrastructure when using get\_update, i.e. state change readings.

We implemented the get\_update and set\_update methods for the Activity Manager application, as well as in the trivial application. There were two reasons for doing this:

- the time consideration
- the memory consideration

It turned out that a more important problem than time, arises when using get\_state: the memory (or other storage space) problem when storing state changes.

\[^1\text{Here, the details for each loop are not given, as differences were small enough to be ignored.}\]

<table>
<thead>
<tr>
<th>Method</th>
<th>3 replicas</th>
<th>4 replicas</th>
<th>5 replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method 1</td>
<td>92(30%)</td>
<td>120(22%)</td>
<td>135(23%)</td>
</tr>
<tr>
<td>Method 2</td>
<td>60(17%)</td>
<td>77(19%)</td>
<td>85(19%)</td>
</tr>
<tr>
<td>Method 3</td>
<td>54(16%)</td>
<td>74(19%)</td>
<td>84(20%)</td>
</tr>
<tr>
<td>Method 4</td>
<td>54(12%)</td>
<td>72(13%)</td>
<td>81(13%)</td>
</tr>
<tr>
<td>Method 5</td>
<td>60(10%)</td>
<td>81(12%)</td>
<td>89(12%)</td>
</tr>
<tr>
<td>Method 6</td>
<td>54(10%)</td>
<td>77(12%)</td>
<td>88(12%)</td>
</tr>
</tbody>
</table>

Table 7.3: Time spent in consensus (given in ms)
and reply values in the register. The problem is due to the fact that all outcomes for all executed requests have to be stored.

Thus, even though the effort demanded from the application writer to describe the get update method is bigger than for describing get state, it might be worthwhile in this case.

There is also a disadvantage when using checkpointing with state updates. All updates after an “initial” state have to be kept in the log; there is no possibility of removing updates, since they are only partial “states” with no identity on their own. So, at failover, if all updates existing since an initial state have to be applied on a new recovering replica, the time taken by the operation can grow to infinity.

7.3.3 Telecom application - failover times

Table 7.4 contains examples of failover times for groups of different sizes. The curve in Figure 7.1 indicates the dependency of the failover time on the number of updates that have to be set in the new leader, in order to get its state up-to-date. As mentioned earlier, the number of updates is the same as the number of non-read-only methods executed on the previous leader, until it crashed.

The curve shows an almost linear dependency of the failover time on the number of methods executed on the crashed leader. Of course, it is possible that after executing a large number of requests, the size of the update grows as well. Still, this growth is not so large, so the time taken to enforce an individual state change (according to that update) on the replica does not grow significantly.

The results presented in the table, refer to failover times that contain repeated tentative execution (see Section 6.1.2) of one and the same request (call of method 4). We think that this is not a loss of generality, because we may consider that average execution times of methods are not so different from each other. Also, the other components of the failover time do not
Figure 7.1: Time for failover vs. number of updates to set in the new leader
depend on the method call that arrives when the server is failing over.

In can be noted that the failover time when considering a new leader, consists of the time taken by applying all updates generated since the first request ever processed by the replicated server. If we consider the crash-recovery model, where a server can restart from a non initial state after a crash, then the failover (in this case recovery) time will stay in a finite range, because a new leader is possibly one replica that already processed some requests, as opposed to the first situation where the failing-over replica has to process an ever increasing number of requests.

The failover times for the trivial application and for the telecom application have comparable values (see tables 7.1 and 7.4). This emphasizes more the fact that failover times are not so much dependent on method execution times. Of course, there can be a difference caused by the sizes of state changes to be applied in the failing over leader.

7.3.4 Improvements to the algorithm

The outcome of a request execution consists of update value and reply. Outcomes for all requests executed by a server replica (the leader) have to be stored at every replica. The way the algorithm is described [22], it is suggested that the storage place should be in memory. However, even if a disk is used to write outcomes to, by possibly consuming more time, after a number of requests the space can be filled up. After a number of requests, the algorithm does not work as wished for.

The authors provide a solution to the problem as follows: the outcome of a request is not written in the registers at witness replicas, but actually used to modify the state of their local copy of the application object. Furthermore, the local store with replies to requests is also updated. As a consequence, the memory, or disk storage is not used for the registers, but only for the stores, thus saving some space. The failover times could also be improved, as a new leader starts processing requests from a state very close to that of the crashed leader.

Another solution to the above problem could be the pruning of the registers, from time to time. The pruning could be done if a series of updates applied, starting from a state baseline, would be replaced by the equivalent state. This replacing could happen, from time to time, due to the way the get_update method was implemented: when the get_update method is called the first time after a new state is set in the object (either the initial state or the state in an explicit call to set state), it returns the current state. The problem is that the outcome contains the update value, as well as the request
reply (return value of the method call). Therefore, in order to be allowed to “remove” some of the outcomes by replacing the series of updates with the equivalent resulting state, it must be made sure that the replies within those removed outcomes are not needed. This means that it has to be assured that those requests will not be resent by the client.

Another problem here is that the pruning of the register must happen in the same way at all correct replicas. In other words, when the register is reduced in size by replacing a set of updates following a state value, by the equivalent state value, and the current value of the total order variables are modified, the two operations must happen in the same way at all replicas, because the leader will use the answers from a majority of the correct replicas. It is not known which this majority will be. Even if the coordination of operations in all replicas would be possible, it has to be decided beforehand when the pruning should take place (after how many updates). Also, this pruning should start and finish when no new requests are handled - at a well-defined “cut” of the distributed system.

The most natural moment when the pruning can happen is when a new leader is recovering and is sending out its first update when executing the first request arrived after the crash of the former leader. When the new leader succeeds in proposing its update value, and request return value, the update is actually a state, because it is the first time get.Update is called on the object after a set.state. Thus, immediately after receiving this value from the leader, the witness replicas being aware of the state form of the update, could prune their registers.

We also conjecture that the pruning would help reduce the failover times as well, because the register containing updates and/or state of the leader would be much shorter. Therefore, the current value of the variable indicating the first free position in the registers, would be smaller.

In the current implementation that follows the way the algorithm is designed, the pruning process does not follow in a natural way. Therefore, in our experiments when the number of requests grows over around 7000, the storage space becomes full.

The work by Dutta et al. [22] mentions the crash-recovery model as a possible choice when implementing the algorithm. They also mention that using the crash-recovery model would imply the need for stable storage. Finally, they argue that implementing the register by using stable storage, is equivalent to a register kept in memory at every correct replica. Our experiments did not involve restarting a crashed leader, although it would have been interesting to see whether the failover time does not converge to a certain value, over a larger time scale.
Another aspect related to failover, is the restriction already mentioned in Section 5.5 about lessons learnt in the FT-CORBA infrastructure. Also, for the second platform, it is recommended that the application does not contain internal threads that might change the state of the application with no possibility of capturing this in the server interceptor. This is because, at failover, some update that happened uncaptured in the server interceptor, could be not “seen”, and so the state of the new leader will not correspond to the one of the crashed leader, at the moment of its failure. Thus, this restriction remains valid in the second architecture, and should be seen as a “price” to pay for the interoperability that a middleware provides for the application writer.

Summary

Chapters 6 and 7 presented an alternative fault-tolerant CORBA infrastructure. A robust algorithm was implemented by using CORBA (for the client-server application part) and plain sockets for the infrastructure communication part. This platform does not involve extra vulnerable components. It involves, on the other hand, unreliable failure detectors to be queried about the identity of the current leader. As mentioned in the beginning of Chapter 6, the algorithm is robust, so that even if such extra processing units crash, the server replicas do not end up in an unknown situation. It was also mentioned that those processing units reflect the “up-ness” of the server replicas to which they are connected. The implementation of the robust algorithm was evaluated using the same test cases as in the FT-CORBA platform. The results were a clear increase in overheads as well as clear increase in failover times. Factors that affect both of these metrics were discussed in detail, as well as shortcomings of the existing algorithm in terms of memory usage.
Chapter 7. Evaluation of the Robust Algorithm
Chapter 8

Conclusions and Future Work

This chapter will present some concluding remarks about the work in this thesis. Also, future extensions and directions will be pointed out.

8.1 Conclusions

In this thesis we have presented our experience with building two different shaped fault-tolerant CORBA infrastructures. Evaluation was conducted for both platforms and comparison between the two alternatives have been performed.

Our results obtained after the experiments with the first platform (built following the FT-CORBA standard) lead us to the following conclusions:

- FT-CORBA standard is feasible to implement by changing an existing ORB. The main shortcoming of this approach is the existence of single points of failure in the infrastructure itself. This could be remedied by ad-hoc replication. However, another weakness is that the failure detection mechanism is pretty rigid: timers are visible at a quite high level and are explicit in this approach.

- Overheads in case of warm passive replication were close to the overhead values in cold passive replication. This is an advantage, because the failover time in case of warm passive replication is less than in case of cold passive replication.
Chapter 8. Conclusions and Future Work

- The large overhead values for passive replication are caused by the one-by-one method execution policy. The true overhead would be much less if requests would arrive with a large enough interarrival time.

- Active replication has a large overhead value, but a significantly shorter failover time than passive replication. It is certainly a viable alternative when high availability is a critical property.

In a second, alternative infrastructure, we experimented with the implementation of the robust algorithm of Dutta et al. [22], and compared the results with the ones obtained when using the FT-CORBA infrastructure. We concluded the following:

An alternative infrastructure removes single points of failure, but gives long failover times. The failover time can reach huge values when the new leader is recovering after the server processed a significant number of requests. Also, the overheads during normal failure free periods of request processing, are larger than in case of passive replication. The memory or other storage space problem also becomes serious here, because, by the design of the algorithm, no pruning of the result register is done. Therefore, exactly as in case of the failover time, memory space needs can grow to infinity.

On the other hand, failover times are much less than in case methods would be replayed instead of only applying the state change enforced by a method execution. Therefore, using this second alternative would bring a big advantage as compared to the FT-CORBA platform, especially if the failing over leader would have to set a baseline state and a small set of updates obtained since the last state reading. Recall that in the FT-CORBA platform, requests arrived since a last checkpoint were logged and had to be replayed on the failing over primary.

8.2 Future work

Future work can be directed towards increasing the intelligence of the infrastructure with respect to placement of replicas on hosts. For example, an improvement would be to introduce some metric for the host "ability to stay up". This measure could be used to create and activate replicas on hosts where probability of failure is low. If all hosts are overloaded, create a new replica on a new host, and possibly unload some other host. Of course, this operation can be done if the client is ready to pay the price in terms of
longer response time of the new replica being started on the “efficient” host. If not, the search for “good” hosts can be done as a background operation.

Also, the FT-CORBA infrastructure can be extended by replicating important components. The need for replication can be studied, and applied for the most vulnerable components.

A way of reducing the next time to failure would be to trace failure causes: design faults, language problems (e.g. Java), or machine shutdown. If one of the causes is given by design faults, then use several versions, and when one version fails, replace the existing other replicas with the new version of the code.

For the alternative infrastructure, it is appealing to study the timing values for scenarios where the client and the server group have different views on who is the leader.

A policy for the register pruning (see Section 7.3.4) has to be devised. This policy would establish when to do the pruning and how to coordinate between different replicas.

Of course, choosing one middleware (CORBA) to be used in the experiments was a first step in exploring trade-offs. An interesting future direction would be to experiment on top of other middlewares, since most of the ideas used in the CORBA setting apply in any middleware.
Bibliography


