From Crash Tolerance to Byzantine Tolerance: Fail Signalling Dependable Distributed Systems

PHD Thesis

by

Dimane Mpoeleng

School of Computing Science
University of Newcastle upon Tyne

February, 2005

Abstract

Many fault-tolerant group communication middleware systems have been implemented assuming crash failure semantics. While this assumption is not unreasonable, it becomes hard to justify when applications are required to meet high reliability requirements and are built using commercial off the shelf (COTS) components. This thesis implements new techniques to deal with Byzantine faults in a distributed group communication system.

This thesis proposes a technique by which a process is duplicated into two replicas such that the process is turned into a self-checking pair with the two replicas communicating synchronously over a reliable network, but two different replicas from different processes can be connected asynchronously. The proposed approach is based on the replicas obeying state machine replication (SMR). SMR is utilised to assure signal-on-failure (fail-signal) semantics. One or both of the two replicas always issues a signal to other entities whenever there is a failure between and within the entities. This way, dependable activities such as group member failure detection, liveliness and security are removed from the upper layer of group communication service down to the two-replica pairs. With most of failure detection and security activities confined between the two replicas, semantics of a group communication are simplified and the number of phases and rounds of group communication protocols is reduced.

The thesis demonstrates the fail-signalling concept by converting a group communication system member, through duplication of each group member, into a self checking pair. Security is augmented to the replicas' fail signalling capabilities to tolerate even more serious Byzantine faults. Performance results of

the traditional group communication system are compared with results of a group system with duplicated fail signalling group members. The thesis has proven that the fail signalling group communication has the advantage of detecting failures faster without suspicions and that resulted in better group communication semantics, better dealing with member failures and faster formation of new group views.

Acknowledgements

This thesis grew out of a series of conversations with and contributions by many people at the University of Newcastle upon Tyne. I would have liked to individually thank all of them but several people have made outstanding contribution to the realisation of this work. My most heartfelt gratitude goes to my supervisor Dr. Neil Speirs who has endured supervising this work for years. I would also like to thank Dr. Paul Ezhilchelvan for his outstanding professional support and interest in this thesis.

This thesis would not have survived as an island. It is based on previous works by, among others, Prof. Santosh Shrivastava, Dr. Graham Morgan, Doug Palmer, and Dr. F. V. Brasileiro., some of whom have provided technical assistance whenever sought.

I am thankful to staff and students of School of Computing Science at the University of Newcastle upon Tyne for making my four year study such a joy. Finally I am grateful for the financial support I enjoyed that was generously and fully offered by the University of Botswana Computer Science Department.

TABLE OF CONTENTS

Chapter 1 - Introduction

1.1 Overview 1
1.2 Thesis objectives 3
1.3 Implementation of a fail-signalling system 4
1.4 Findings and benefits of a fail-signalling service 5
1.5 Thesis structure 6

Chapter 2 - Background and related work

2.1 Overview 9
2.2 Fundamental problems in distributed systems 10
2.2.1 Detecting a crashed process in an asynchronous system 10
2.2.2 Global state 12
2.2.3 Partial failures, group consensus and membership 15
2.2.4 Closing remarks 16
2.3 Typical dependable system models 17
2.3.1 Process model 17
2.3.2 Synchronous and asynchronous network 18
2.3.3 Group communication system model 20
2.4 Failures and failure detection 21
2.4.1 Terminology of failures 21
2.4.2 Failure detectors 22
2.5 Byzantine tolerance and security 25
2.6 Group communication systems 26
2.6.1 Passive and active replication techniques 27
2.6.2 Consensus and group membership protocol 27
2.6.3 Atomic broadcast and virtual synchrony 28
2.6.4 Order protocol 29

2.6.5 Liveliness 29
2.6.6 Group partitions and gossips 30
2.7 Fault tolerant group communication systems 31
2.8 Byzantine tolerant and secure group communication systems 32
2.8.1 Group communication approach 33
2.8.2 Failure detectors and oracles 34
2.8.3 Stochastic and computational techniques 35
2.8.4 Quorum systems 36
2.9 Distributed computing systems and failure detection 36
2.10 Summary and conclusions 41
2.10.1 Problems 41
2.10.2 Current solutions 42
2.10.3 Limitations of the current solutions 42
2.10.4 Contributions of this thesis 43

Chapter 3 - The fail-signalling system

3.1 Introduction 45
3.2 Fault model 46
3.3 The problem 46
3.3.1 Dealing with FLP impossibility 46
3.3.2 Liveliness and fail-signalling 47
3.3.3 Properties of fail-signalling process 49
3.4 Construction of Fail-Signal (FS) Processes 50
3.4.1 Assumptions and principles 50
3.4.2 A fail-signalling pair 52
3.5 Implementation of fail-signalling pair 54
3.6 Optimizing a fail-signalling pair 56
3.6.1 Improving synchronized clock algorithm 58
3.6.2 Order protocol optimization in fault free runs 59
3.7 Concluding remarks 61

Chapter 4 - A fail-signalling group communication system

4.1 **Overview** 62

4.2 **The NewTOP Group Communication Service** 63

4.3 **Extending NewTOP to FS-NewTOP** 66

Chapter 5 - Results and performance analysis

5.1 **Introduction and objectives** 71

5.2 **Operating environment – Mega cluster** 72

5.3 **Configuration and Object Distribution** 72

5.4 **Measuring group consensus latency** 74

5.4.1 The global timer (real time clock) 74

5.4.2 Events: crash, detect, and consensus (determination) 76

5.5 **Performance analysis** 77

5.5.1 Throughput versus number of group members 78

5.5.2 Throughput versus message sizes 80

5.5.3 Order latency versus number of group members 82

5.5.4 Order latency versus message sizes 84

5.5.5 Cost of consensus when one member fails (varying members) 85

5.5.6 Cost of consensus when one member fails (varying message size) 86

5.5.7 Cost of consensus when one member fails (varying timeouts) 87

5.6 **Discussion and conclusion** 89

Chapter 6 - Discussion and conclusions

6.1 **Overview** 90

6.2 **Assumptions** 90

6.3 **FS service and further work** 92

Appendix A - Porting and adapting Voltan to fail signalling java/CORBA 95
References 116
Bibliography 133

List of Figures

Figure 3.1: Architecture of Fail-Signal Wrapper Objects for middleware process p.

Figure 3.2: Fail Signaling Wrapper objects

Figure 4.1: The NewTOP service: access and structure.

Figure 4.2. The FS-NewTOP system

Figure 4.3: Deployment of the components of FS-NewTOP for a 3-Member Group.

Figure 5.1a: Group member structure for (a) traditional NewTOP (b) Fail-Signalling and secure FS-NewTOP and (c) the object/node distribution strategy for FS-NewTOP.

Figure 5.1b External timer for crash events and group communication systems

Figure 5.2: Throughput under maximum message load against varying group members

Figure 5.3: Throughput under maximum message load against varying message size

Figure 5.4: Symmetrical order latency against varying group members

Figure 5.5: Symmetrical order latency against varying message size

Figure 5.6: Membership protocol overhead when one member suddenly fails (varying number of members, fixed message size to 5 bytes)

Figure 5.7: Membership protocol overhead when one member suddenly fails (varying size of messages, fixed group membership to 10 members)

Figure 5.8: Figure 5.8: Membership protocol overhead when one member suddenly fails (varying liveliness timeouts)

Figure A.1: Voltan fail silent process structure

Figure A.2: CORBA Voltan fail-signalling structure

List of Tables

Table 2.1: Failure detector properties

Table 2.2: Classes of failure detectors

Table 2.3: Atomic broadcast properties

Table 2.4: Systems' failure detection and security

Table 2.4: Classes of failure detectors

Table 5.1: Throughput under maximum message load against increasing group members

Table 5.2: Throughput under maximum message load against increasing message size

Table 5.3: Symmetrical order latency against increasing group members

Table 5.4: Symmetrical order latency against increasing message size

Chapter 1

INTRODUCTION

1.1 Overview

Today's society is much more dependent on distributed computing than ever before in homes, academia, research, commerce and industry. Distributed applications are deployed in a wide range of environments from non critical but demanding systems such as video streaming, teleconferencing, internet server farm to more mission critical applications such as finance, airline booking systems, electronic auctions, medical equipment and air traffic control. Distributed computing has also found its way into high performance systems in nuclear power stations, human genome simulations, space missions, weather forecasting and film and entertainment industry.

To satisfy the demands of these applications, there are a number of properties that the underlying distributed systems are required to obey. Some of the desirable properties include high availability, timeliness, fault tolerance and security. A system that satisfies these properties should continue to operate and service clients at anytime even if it encounters faults, it is attacked or some of its components fail. Different systems vary on how well they can handle that in terms of what malicious attacks they can handle and up to how many system components can fail without affecting the operation of the whole system.

Most of dependable distributed systems are constructed from a number of applications or processes that cooperate to achieve one goal as a group. The

processes communicate certain type of messages obeying same communication rules. This collection of processes is a Group Communications System (GCS). GCS is used as an abstraction to address a wide range of distributed systems applications. To achieve the dependable properties just stated, a GCS is required to satisfy group communication challenges that are only unique to GCS. Some of the challenges include the order of messages received by members, membership consensus, elections, clock synchronisation and atomic broadcast. Total message order protocol ensures that non-faulty members of the group receive all messages sent to them in the same order. In addition, all members have to unambiguously agree on certain actions and values that determine correctness of the group communication operations.

Causes of these dependability challenges include: the error prone network that link all the participating group members, the partial failures that may occur due to the autonomous nature of application hosts and the vulnerability of the distributed computing as a result of its exposure to adversaries that may have malicious access to the network.

Substantial amount of research work has already been carried out to tackle such dependability and security problems. It is evident from literature that almost all solutions targeted toward GCS have been centred around applying traditional agreement protocols, using signatures for authenticity, and over reliance on liveliness mechanisms.

1.2 Thesis objectives

Many fault-tolerant group communication middleware systems have been implemented assuming crash failure semantics. Crash failure semantics state that a process participating in a group can crash without propagating errors to other processes. While this assumption is not unreasonable, it becomes hard to justify when applications are required to meet high reliability requirements and are built using commercial off the shelf (COTS) components.

Distributed dependability of group communication systems has in the past relied on protocols that are based on completely connected single processes that communicate via message passing for both application specific messages and fault tolerance messages. It is desirable to decouple fault tolerance and security from application-specific services and messages. Decoupling application services from dependability services greatly simplifies and betters group communication semantics and computations. For example, removing liveliness properties from the group layer would ease the group membership protocol profoundly in that the detection of faults and agreement on values by group distributed members will be faster and decoupled from membership service.

This thesis proposes techniques that will make sure dependable activities such as group member failure detection, liveliness and security are removed from the upper group communication layer. The proposed model requires a single process to be duplicated so that the process becomes a self checking pair with the two replicas communicating synchronously over a reliable network, but two different replicas from different processes can be connected asynchronously. With most of failure detection confined within the two replicas, semantics of a group communication are wrong suspicious on process failure is eliminated. The

proposed approach is based on the two same-process-replicas obeying state machine replication (SMR). SMR is utilised to assure signal-on-failure (fail-signal) semantics. One or both of the two replicas always issues a signal to other processes whenever there is a failure between them. This way, an originally crash tolerant group communication can be extended into a Byzantine tolerant system by use of fail-signalling protocol.

1.3 Implementation of a fail-signalling system

The proposed fail-signalling (FS) approach requires that group members that are duplicated to act as one, as mentioned, conform to the state machine replication (SMR) model. SMR is used to assure signal-on-failure (fail-signal) semantics at a level where existing crash-tolerant services can be seamlessly deployed. The resulting system can provide total ordering that has no liveliness requirement for termination.

This work demonstrates the effectiveness of the suggested approach by porting a GCS system designed and developed at University of Newcastle upon Tyne called Newcastle Total Order Protocol (NewTOP). The crash-tolerant NewTOP CORBA group communication service was ported into a Byzantine tolerant Fail-signalling NewTOP one referred to as FS-NewTOP. Security is augmented to the replicas' fail-signalling capabilities so that the group system can tolerate even more serious Byzantine faults. The Fail-signalling techniques developed in this thesis can be applied to other distributed computing systems whose components are state machines.

1.4 Findings and benefits of a fail-signalling service

The thesis has proven that the fail-signalling group communication faired better in areas that it was expected to such as better group communication semantics, dealing with member failures and formation of new group membership faster. This work improved on what the distributed community have achieved by decoupling security infrastructure from the upper middleware layer. Equally important an effective fail-signalling layer was added to the system. The replicas of the process were each placed on different nodes connected by network assumed to be synchronous, while the link between two different group members may be asynchronous.

The fail-signalling structure enabled us to pull down the failure detection mechanism from the upper group layer and confine it within duplicates that either produces identical results or a failure signal to some group members all the time. If the duplicated member sends a duplicate application message or fail signal message, one of the duplicate messages is suppressed on receipt.

These techniques eliminate suspicions and partition by ensuring that a failure detector receives a failure signal from a failing process and therefore no need for suspicions [Mpoeleng03]. This use of either failure detection or group communication techniques for fault tolerance could even be more efficient if the processes were guaranteed to fail only by crashing and being able to distribute a failure signal to the nearest fellow group process(es) which immediately initiates new membership view protocol. FS protocol eliminates failure suspicions that are wrong thereby disallows partitions of a group communication system while maintaining the integrity of a normally operating group system.

The benefits of fail-signalling distributed systems, particularly group communication systems, are summarised below.

- Decoupling of fault detection and security from application layer
- Faster failure detection leading to faster group membership protocol
- Elimination of group partitioning
- Clean and assured crash without propagating errors to other parties

At the end of this work, it has been proven that the construction of a fail-signalling service that brings about the above mentioned benefits is possible. In the results chapter, findings of performance of the traditional NewTOP against the new FS-NewTOP are presented. As predicted, the FS-NewTOP performance is much better when the system loses a group member because the fail-signalling protocol produces a sure failure to the system hence eliminating the need for group member to undergo a consensus protocol but rather forces the FS group to run a new membership view immediately.

1.5 Thesis structure

The thesis is organised as following:

Chapter 2: Background and related work. This chapter covers the background of distributed computing dependability. Topics and subtopics include: fundamental challenges in distributed systems which are the subject of current research. The chapter then presents typical system model for both synchronous and asynchronous networks. Byzantine tolerance and security are

also covered. The most command group communications systems are discussed in detail before a conclusion is drawn with emphasis on the limitations of the current group communication system.

Chapter 3: The Fail-signalling system. This chapter covers the main contribution of the thesis. The chapter explains how the fail-signalling service was constructed and how the target application (or group member) is duplicated to act as one self-checking process. The chapter demonstrates how the fail-signalling service can be optimised in fault free environments. This chapter's topics include: The problem to solve, dealing with FLP impossibility, liveliness and fail-signalling, properties of fail-signalling process, construction of fail-signal (*fs*) processes, assumptions and principles, a fail-signalling pair, implementation of fail-signalling pair, optimizing a fail-signalling pair, improving synchronised clock algorithm, order protocol optimisation is fault free runs, and concluding remarks

Chapter 4: A Fail-signalling group communication system. This chapter demonstrates how a fail-signalling service can be incorporated into an existing distributed application. The target group system that is enhanced with the FS service is the group communication systems developed at the University of Newcastle upon Tyne called NewTOP. This chapter demonstrates how NewTOP was minimally changed to be connected to the fail-signalling service so that members of the NewTOP group communication are each duplicated to act as robust self checking entity. The chapter starts with an overview of the chapter, the description of NewTOP group communication service, extending NewTOP to FS-NewTOP and Implementation details.

Chapter 5: Results and performance analysis. The chapter compares performance of NewTOP versus FS-NewTOP on various performance properties. The chapter includes: Operating environment - Mega cluster,

configuration and object distribution, performance analysis, throughput versus number of group members, throughput versus message sizes, order latency versus number of group members, order latency versus message sizes, cost of group membership protocol when one member fails (varying members), cost of group membership protocol when one member fails (varying message size), and discussion and conclusion of results.

Chapter 6: (Discussion and conclusions). The chapter concludes the overall thesis and implications of its findings. It summarises the performance implications of fail-signalling service in terms of the f formula and f is the maximum number of group members that can fail without affecting the system. This chapter briefly discusses the contributions of this thesis and suggests future work.

Appendix A: Porting and adapting Voltan to fail signalling java/CORBA System. This appendix describes in detail how the fail silent system was adapted and ported to a fail-signalling Java/CORBA version. The new CORBA based Voltan seamlessly connected with the CORBA based NewTOP and ensured minimal changes to NewTOP.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Overview

This chapter reviews and analyses the inherent fault tolerance and security challenges in distributed systems and examines existing solutions that address them. A number of distributed computing systems are cited. The chapter further highlights the limitations of solutions offered by such systems and finally justifies this thesis.

First, problems that are inherent in distributed system are identified and a more detailed description of the problems is presented. Then a group communication system paradigm is discussed that is usually used to solve such problems. The chapter continues by giving details of Byzantine faults before discussing distributed system that tolerate benign faults and those that tolerate more serious faults such as Byzantine and malicious faults. Examples of practical group communication solutions are introduced and their failure detection mechanisms are discussed. The chapter concludes by highlighting the weaknesses of the current approaches and suggests better solutions.

This chapter covers the following topics: Fundamental problems in distributed systems; Distributed system model; Failures and failure detection; Byzantine faults and security; Group communication systems; Fault tolerant group communication systems; Byzantine tolerant and secure group communication

systems; Example group communication systems with failure detection; and finally Summary and conclusions

2.2 Fundamental problems in distributed systems

There are several fundamental problems that are central to building today's fault tolerant distributed systems. When designing a distributed system applications and their protocols, designers should consider these key problems ([Cristian91a], [Nancy96]):

- identifying a crashed process from a slow process in an asynchronous network configuration,
- accurately taking a snap shot of the system's global state,
- dealing with arbitrary, Byzantine faults, and
- reaching a consensus among a group of processes.

These challenges are further explained below.

2.2.1 Detecting a crashed process in an asynchronous system

An asynchronous network is characterised by computer nodes connected by links whose timing latencies are not known but bounded. It is impossible to distinguish a crashed process (or a disconnected link) from a slow process connected by an asynchronous link. The problem is tackled differently in synchronous networks

and asynchronous networks. In a synchronous network, message transmissions are subjected to maximum timeouts while in asynchronous network it is not possible to impose such upper latency time bounds. In an asynchronous network there is no known message transmission timing delays or relative processor speeds, therefore a slow process cannot be distinguished from a crashed one. This key problem has been termed the "FLP Impossibility" named after the authors, Fischer, Lynch and Paterson, who proved that it is impossible to reach an agreement among a group of asynchronously connected processes when just one of the processes crashes [Fischer85]. Further proofs of FLP impossibility result are presented in [Fischer86].

However, extensive work has been carried out to circumvent the FLP impossibility result. Attempts range from theoretical solutions ([Vijay98b]) to practical solutions ([Chandra91], [Chandra96]). Vijay *et al* introduced an *infinitely often accurate* failure detector which can be implemented in an asynchronous system and assume that when a process has failed or crashed it remains in that state throughout a run. The aim of this thesis is to guarantee that condition in a different way. Chandra *et al* introduced the concepts of unreliable failure detectors that can be used to solve consensus in an asynchronous distributed system more on Chandra solutions is discussed in this chapter.

These solutions are dominantly targeted to a group of processes, that are communicating and are completely connected or there is a direct link between any two processes, by message passing and they have to reach an agreement despite a number of them failing. The efficiency of a solution is judged by the number of rounds needed to reach an agreement and how many processes can be faulty without affecting the proper agreement over failed members of the group by non-faulty members. The number of processes that can be faulty without affecting the group decisions is an upper bound and it is denoted f, where f is the

maximum number of faulty group members relative to size of the group *n*. Efficiency of group agreement protocols is determined by the number of rounds and *f*.

The relationship between n and f is determined by several factors:

- What types of faults are to be tolerated? Benign, crash, malicious
- What is the underlying communication protocol? Synchronous or Asynchronous
- How many numbers of rounds are required for consensus?

In an asynchronous network, to tolerate benign and crash faults $2f+1 < n$ processes are needed to reach consensus. To tolerate malicious faults $3f + 1 < n$ are needed [Schneider90]. For synchronous network model to tolerate benign or malicious faults $f+1 < n$ processes are required regardless of assumption of authenticated or no-authenticated channels [Fischer82, Dolev82, DeMillo82]. $f < n/3$ is required for malicious participants [Lamport78], $f < n-2$ in the case of benign, $f < 1/2$ network connectivity for not-full connected network [Dolev82].

2.2.2 Global state

Another hurdle in distributed system protocols is that there is no real time global clock and therefore difficult to accurately get the global state snapshot of a distributed system. It is even more difficult to get a global state if a majority or all correct processes have to communicate to agree upon the system state. Each node in a distributed system has its own real time clock. It is hard to keep clocks in real time steps with one another not only because the clocks drift against each

other, but also because the same messages sent between nodes to keep clocks in step are exposed to unpredictable latencies. Parties may not be aware of not just the global time, but also the system's general global properties or state.

There are algorithms that keep physically separated clocks in synchrony such as the Berkeley algorithm and Halpern's algorithm [Halpern84] that both synchronize clocks within given bounds by message passing in a fault tolerant way. The algorithms tolerate both link and node failures of any type and they can maintain clock synchronization with arbitrary networks (not just completely connected networks). The problem with these clock synchronisation algorithms is that they still require processes to communicate with the rest or majority of processes in a group and that can lead to high message load.

Real earth time helps us determine which event happened before which. Measurements need to be taken to determine which event happened before which event for overall deterministic ordering of events.

Lamport's logical clocks in [Lamport78] and [Halpern84] demonstrate how to address the global clock problem without using real physical clocks. Logical clock are based on the concept of associating an increasing integer number to an event, depending on which event caused which? This is causality temporal relationship. Schwarz detects causality relationship in a distributed system [Schwarz94]. A causal relationship is a predicate that determines which event happened before which based on which caused which to happen. If process A *sends* message *m* to process B and B *receives* it, the *send* event happened before the *receive* event because the former caused the latter. A lot of these timing theories and systems have been developed based on Lamport's logical clock.

Various researchers have addressed global snap shot in different ways, the challenge of observing a distributed global state. Chase and others have shown

how to detect global properties of a distributed system but highlighted its limitations [Chase98]. Chandy [Chandy85] and Shah [Shah84] use distributed snapshots to determine the global state of a system while Felix and Kloppenburg use weak fault assumptions to consistently detect global predicates in an asynchronous distributed systems ([Felix00], [Kloppenburg00]). Li employs distributed predicates to try to solve more practical problems such as distributed deadlock detection [Li93].

2.2.3 Byzantine faults

Byzantine faults have been one of the dominant problems in distributed systems research community and the faults have been tackled in different ways over many years as fully discussed in subsequent sections. Byzantine faults are arbitrary and unpredictable and can either be inadvertent or malicious. The Byzantine problem was first coined by Lamport in 1982 and it was modeled around several distributed processes, some of which can be faulty but the rest of the group must be able to reach a consensus.

The Byzantine problem is analogous to various army generals in command of different regiments surrounding an enemy city. The generals communicate only by passing messages from one regiment to the other via human messengers. Some generals may be traitors by sending wrong messages, however, the non-traitors must reach agreement on whether to attack or not. This paradigm came to be known as the Byzantine Generals Problems ([Lamport82], [Lamport83], and [Cristian85]).

Subsequent research efforts to solve this problem are centered on a number of distributed processes cooperating as a group. The challenge is for the non-faulty processes (non traitors) to reach an agreement despite Byzantine failures (from traitors). It has been proven that up to a third of n processes can be faulty (f)

without affecting consensus of non faulty processes – $f < n/3$. This Byzantine agreement can be achieved by a determinist protocol with complexity of $O(f)$ rounds [Pease80].

Furthermore, Castro and Liskov have carried out a number of theoretical and practical solutions to tolerate Byzantine faults ranging from normal faults to malicious faults. They developed practical solutions for state machine replication that tolerates Byzantine solution which was tested in NFS services to prove its applicability. The algorithm operates in an asynchronous network and it offers both liveliness and safety provided no more than a third of group members are simultaneously faulty. This means that clients eventually receive replies to their requests and those replies are correct [Castro99c]. In another paper [Castro99a], the authors prove these algorithms formally using I/O automaton. Other algorithms eliminate the need for costly public-key cryptography by replacing public key cryptography with message authentication codes (MAC) and the speed is doubled while maintaining the same degree of security [Castro99b]. They further enhance their algorithms to not just tolerate but to recover from Byzantine faults [Castro00].

However all these Castro and Liskov algorithms and many other research works are based on complete communication among processors which can still be costly in terms of message traffic. On the contrary, algorithms proposed in this thesis confine majority of fault tolerance and security operations within two replicas.

2.2.3 Partial failures, group consensus and membership

Partial failures are one of the major challenges of a distributed cooperating group processes. Unlike a centralised and localised system with one processor and one

memory, a distributed computation can fail partially and independently at various locations of the network. In such circumstances, measures should be taken to ensure integrity and correction of such computation [Cristian91a]. At any one time, various processes of the distributed system may have failed. The distributed system must be designed correctly so that these failures have little visibility to the observer of the system. This property is called high availability and it is usually implemented by replication of a service over multiple components and by replication of information.

Reiter *et al* describe group membership as a protocol that enables processes in a distributed system to agree on which group of processes are currently operational. Membership protocols are used to maintain availability and consistency in distributed applications [Reiter94b]. They present a membership protocol for asynchronous distributed systems that tolerates the malicious corruption of group members. Their protocol ensures that correct members control and consistently observe changes to the group membership, provided that in each instance of the group membership, fewer than one-third of the members are corrupted or fail benignly.

2.2.4 Closing remarks

The challenges discussed above are inherent in distributed computing and are active topics among researchers and professionals. Solutions range from simple redundancy measures to complex group communications with signatures. Major challenges of today's asynchronous distributed systems models and protocols are based on this fact compounded with security issues. One protocol that has been the challenge for decades is the agreement protocol. In the protocol, participating group members have to reach an agreement and terminate despite presence of

some faulty or betraying members within a known maximum number of faulty members (upper bound). Much research work has been carried out in trying to optimize the upper bound and various research works have provided both theoretical and practical techniques to address the upper bounds.

2.3 Typical dependable system models

This section discusses most commonly implemented distributed system models particularly those that are designed to tolerate faults and malicious attacks. A dependable distributed system is composed of a set of processes together characterised by strict stochastic specifications (minimum probability that the standard behavior is observed at run-time, and maximum probability that a potentially catastrophic failure that is different from the specified failure behaviour is observed) [Cristian99]. The dominant protocol model discussed is group communication system whose processes are distributed across a synchronous or asynchronous network. The processes communicate only through message passing, not shared memory. This work makes distinction between synchronous and asynchronous system models. The synchronous and asynchronous system models should not be confused with synchronous (send-and-wait) and asynchronous method calls (fire-and-forget to continue).

2.3.1 Process model

A distributed system is modelled as a collection of cooperating processes distributed over a network. Processes communicate only by message passing. Each process obeys state machine replication properties and Schneider describes

a state machine approach as a general method for implementing fault-tolerant services in distributed systems [Scheinder90], This message passing model is point-to-point, connection oriented and allows for asynchronous message passing. On top of the point to point, a multi-cast software layer is placed on top and an underlying hardware or network may support multi-cast functionality Lamport86], [Scheider90], [Schlichting83],

This work deals with distributed entities or processes connected by a point-to-point network. A process is a program that executes instructions sequentially and communicates with other processes only by message passing even if the two communicating processes share the same processor, memory or clock. A message is passed from process *p* to *q* by *p* calling *p.send(q,m)*, and subsequently *q* calling *q.receive(m)*. It is assumed that there is a bounded message buffer between *p* and *q*. The processes are linked by message queue and unidirectional. A bidirectional communication can be achieved by using two or more queues.

2.3.2 Synchronous and asynchronous network

The *asynchronous* model of distributed systems has no bounds on

- execution entity (the principles of distributed computing community uses the term "process") execution latencies — i.e., arbitrarily long (but finite) times may occur between execution steps

- message transmission latencies — i.e., a message may be received an arbitrarily long time after it was sent

- clock drift rates — i.e., process's local clocks may be arbitrarily unsynchronised

In other words, the asynchronous distributed system model makes no assumptions about the time intervals involved in any behaviour. Although the maximum delay and processing time are not known they are bounded.

It is also assumed that network link is a fair link which means a message from p to q is eventually delivered after an infinity number of trials. [Schlichting83] [Schneider90]. The network may fail to deliver messages, delay them, duplicate them, or deliver them out of order.

A synchronous network is when clocks speeds of participating processors are in step, and maximum network delay between two processors and processor speed is bound and known ([Cristian91b]). The *synchronous* model of distributed systems, conversely, has apriori known upper and lower bounds on these quantities

- each process has a bounded time between its execution steps

- each message is transmitted over a channel and received in a bounded time

- process's local clocks may drift either from each other or from global physical time only by a bounded rate (note that this does not necessarily bound the clocks' accuracy)

Such a system model makes it easier to facilitate reasoning about properties of a distributed system if those presumed bounds are not deterministic it follows that reasoning based on them can not be deterministic.

When processes are distributed over an asynchronous network, each process is modelled as having its own processor, memory and clock (not global clock). Messages are sent between processes over a link that has no known timing delay bounds. Messages can be indefinitely delayed or lost, however a link is assumed to be a fair link – eventually a message reaches the destination if it is sent infinite number of times.

Dolev and others have explored solving Byzantine agreement in both synchronous [Dolev93] and asynchronous environments [Bazzi01]. For synchronous systems, Dolev uses tight bounds on the load of synchronous Byzantine quorum systems for various failure assumptions.

2.3.3 Group communication system model

There are three types of fault tolerant distributed systems, those that rely on failure detectors, those that rely on group communication service, and those that use the hybrid of both failure detectors and group communication systems. Failure detectors can employ a more aggressive timeout compared to that of group communication [Birman93], [Melliar-Smith91], [Morgan00a]. The processes must conform to certain group communication system as total order of message, virtual synchronous, fault tolerance. See Sections 2.6 and 2.7 below for details of group communication systems.

2.4 Failures and failure detection

2.4.1 Terminology of failures

A fault, an error and a failure are defined. A fault is an internal defect of a system. A fault may or may not cause a deviation of the system from normal internal operation. This internal deviation of a system is called an error. If an error causes the system to misbehave against the specification of the system and the misbehaviour is observed from the environment it is called a system failure.

A failure is any type of behaviour of a system that deviates from the systems specification. In the context of this thesis, a failure occurs when a distributed systems component produces not only a wrong result but an expected result that is produced too early, too late or never. For example, node failures include crash, omission and Byzantine. A channel can fail by crashing, message loss, or production of erroneous and arbitrary messages.

There are four different ways by which a process can fail. These include *fail crash*, *fail silent*, f*ail stop*, and *fail-signalling*.

Fail-crash process dies without propagating the faults to the rest of the system. A crashed process may or may not be detected.

fail-silent process crashes cleanly without affecting the system and other processes may or may not detect it. A fail-silent node is a self-checking node that either functions correctly or stops functioning after an internal failure is detected. Such a node can be constructed from a number of conventional processors. In a software-implemented fail-silent node, the non-faulty processors of the node need to execute message order to "keep in step" and comparison protocols to check each other. [Brasileiro96]

On the other hand, Schlichting and Schneider describe a *fail-stop* process that, when crashed, writes its crashed state to a secondary storage so that the other processes know about the crash. Fail stop process are identified by three fundamental properties: 1) Halt-on-Failure Property – the process will halt before performing an erroneous state transition visible to other processes, 2) Failure Status Property. Any non-faulty process can detect the halting of any other process, 3) Stable Storage Property Part of the processes memory is "stable", i.e. unaffected by failure readable by other processes[Schlichting83].

A *fail-signalling* process sends a signal to the rest of the group when it is in a failing state. Construction of fail-signalling processes is the key aim of this thesis. Our fail-signalling process is very similar to fail stop process; the main difference is that the fail-signalling process does not assume permanent storage but a fail signal. A fail-signalling process can also be seen as extending the properties of a fail-silent process.

2.4.2 Failure detectors

Accurate failure detection in asynchronous distributed systems is notoriously difficult. In such system, a process may appear failed because it is slow, or because the network connection is slow or partitioned. This is termed the FLP impossibility result. Because of this, several impossibility results solutions have been suggested and implemented with varying degrees of accuracy.

According to Chanda and Toeug, failure detectors are timers and timeout values. There is a wide range of failure detectors that vary with the degree of *accuracy* and *completeness* properties and they range from perfect error detection ones that are

hard to implement to weak ones that can be practically implemented ([Chandra91], [Chandra96], [Chandra98]).

A failure detector is usually part of the participating process. It gathers information for the process about the other processes that it "*suspects*" to have failed which may or may not be true. All processes that do not suspect each other compare one another's list of processes suspected by their failure detectors to have failed and a group consensus is reached. This may lead to partitioning by different cliques.

Table 2.1: Failure detector properties

Completeness	
Strong	eventually every process that crashes is permanently suspected by every correct process
Weak	eventually every process that crashes is permanently suspected by some correct process.
Accuracy	
Strong accuracy	no correct process is ever suspected
Weak accuracy	some correct process is never suspected
Eventual strong	There exists a time after which no correct process is suspected
Eventual weak	there is a time after which some correct process is never suspected

Chandra's and Toueg's efforts were to provide a solution to Fischer *et al* FLP impossibility results by investigating and developing solution for the weakest possible, and therefore implementable, failure detector $(\Diamond w)$ that can be used to reach consensus in a group communication system. See Table 2.2 below for classes of failure detectors.

Failure detection is based on two principal properties: *completeness* and *accuracy*. Complete can either be strong or weak; accuracy can be strong, weak, eventually strong, or eventually weak.

The most perfect failure detector (and hard to implement) is the one that has strong accuracy and strong completeness referred to as Perfect P. On the other side of the spectrum, the weakest failure detector is the one that has weak completeness and eventually weak accuracy and it is denoted as ◊W. Chandra and Toueg proved that this is the weakest possible failure detector to reach a consensus and be implementable at the same time. The table below shows all possible classes of failure detectors based on a combination of completeness and accuracy properties listed above.

Table 2.2: Classes of failure detectors

Completeness	**Accuracy**			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	Perfect P	Strong S	◊P	◊S
Weak	Quasi Q	Strong W	◊Q	◊W

How the research community has handled failure detection problem varies from theoretical to practical ([Fischer85], [Beauquier97], [Charron-Bost00], [Deepak96], [Matsui00], [Vijay98], [Fetzer96], [Cristian99]). Fischer *et al* laid down the foundation of the problem to be solved (FLP impossibility results) and most of the rest of the authors provide solutions to it.

Group membership provides consistent information about the status of processes in the system, failure detectors provide inconsistent information. [Schiper02] confirm that failure detection and group membership are two important components of fault-tolerant distributed systems not only in fail free runs but in which processes crash. The authors investigated scenarios in which the deployment of either failure detectors or group membership is suitable.

2.5 Byzantine tolerance and security

Bellare highlights the importance of Byzantine tolerance and security [Bellare93]. Message authentication and key distribution are central cryptographic problems in distributed computing but up until early 1990s they had lacked even a meaningful definition. One consequence is that incorrect and inefficient protocols have proliferated. Authors presented mutual authentication and authenticated key exchange and their proofs that they are practical and efficient.

Handling Byzantine faults or arbitrary failures is related to handling security problems such as malicious attacks, therefore dealing with authenticated fault tolerance is the same as dealing with security in distributed systems.

Most of these security protocols assume the existence of a Public Key Cryptosystem:

- Each process/processor has a unique private key that cannot be stolen to digitally sign messages.
- Each process/processor can obtain the public key of other processes/processors (as and when required) to verify signed messages.

2.6 Group communication systems

This section introduces the basic characteristics and properties of a group communication system and it covers roles, properties of a group communication system, passive and active replication techniques, consensus and group membership protocols, atomic broad cast and virtual synchrony, liveliness, and group partitions.

Typical group communication protocols are:

Join Protocol: A new process wants to join the group.

Remove Protocol: A faulty/corrupt process must be removed from the group.

Agreement: If *p* and q are two correct processes, then both the processes will have the same membership view.

Validity: If a correct process *p* installs a view V_i, then V_i includes *p*.

Integrity: If a view includes a process *p*, but the subsequent view excludes *p*, then *p* was suspected by at least one correct member.

Liveliness: If there is a correct process *p* that is a member of a view and is not suspected by *f* correct members of the view then any process of *f* will receive "*I am alive*" message from p within time bound *t*.

Total Order of Views: If correct processes *p* and *q* both install views V_i and V_i+1, then *p* installs V_i+1 after V_i if and only if *q* installs V_i+1 after V_i.

2.6.1 Passive and active replication techniques

Passive replication of processes is based on one process acting as a leader and updating other processes after every operation has been performed. When the leader dies, one of the processes takes over. Active replication is when all processes provide service in step and they all have to be state machines [Schneider90]. In active replication, a failing leader process does not need to send its own state to other processes because they are up-to-date with its operations anyway. This work is focused on an actively replicated process group, although the same concepts of this thesis can be applied to passively replicated processes ([Amir95], Ezhilchelvan89], [Ezhilchelvan95])

2.6.2 Consensus and group membership protocol

Consensus and group membership are closely related. Consensus is when all participating processes agree upon a value, and group membership protocol is when all processes agree upon a list of processes that are believed to be active and have not crashed. The members exchange messages about any suspicions determined by their respective failure detectors even if the fail detectors are imperfect. ([Birman99], [Bracha95], [Dolev87], [Dolev93], [Dwork88], Ezhilchelvan01], [Fischer83], [Melliar-Smith91], [Pease80], [Ramasamy02], [Reiter94c], [Reiter94d]).

Most of group protocols employ the 3-phase commit strategy. The first phase is the Consensus phase where the protocols try to reach consensus on the suspicions generated by the respective fault detectors. The second phase is the

Agreement phase where the protocols reach agreement on the next view to be installed. The final phase is the Commit phase where the view is updated and the system is verified for consistency.

This thesis eliminates suspicions from the first phase thereby making consensus fast and efficient.

2.6.3 Atomic broadcast and virtual synchrony

Atomicity is the "all or nothing" property. For example, either all correct processes receive the same messages in the same order or none of them do. This gives a group of processes an illusion that they are all synchronised thus the term virtual synchrony

According to Cristian *et al* [Cristian85], atomic broadcast must satisfy three properties: unanimity, order and termination.

Table 2.3: Atomic broadcast properties

property	Description
unanimity	Every message whose broadcast is initiated by a member is either delivered to all correct receivers or to none of them
order	All delivered message from all senders are delivered in the same order at all receivers
termination	Every message broadcast by a correct sender to all correct receivers is delivered

We need to extend these properties further such that a message sent by a process is delivered to other processes belonging to a group view exactly the same view as it was when the massage was sent. That is, there should be no view change between sending the message and other processes receiving, this is known as

virtual synchrony. (Birman87], [Dwork88], [Hadzilzcos93], [Melliar-Smith94], [Moser94], [Schiper93]).

2.6.4 Order protocol

One of the fundamental principles of a group communication system is messages received by correct processes should be delivered (or committed) in exactly the same order in all correct processes. ([Dolev93], [Garcia-Molina91], [Pandey01], [Morgan00a]). This is important since for most distributed applications it is important that messages are delivered to group processes in the same order to ensure integrity. Order protocols can be strengthened by improving the accuracy of failure detectors. The ordering property of group communication ensures that all correct processes receive messages in the same order.

There are two ways that messages can be ordered: total order and causal order. In *total order* all correct processes receive and see all messages in the same order. This solution is expensive to ensure total ordering. *Causal order* on the other hand uses which event caused which to order messages. If message *m* causes message *n*, then *m* is ordered before *n*.

2.6.5 Liveliness

Liveliness property is met if message delays over the asynchronous network are perceived to remain stable for a suitably long duration. Processes send periodic "I

am alive" messages to other processes (or coordinator process) to avoid being suspected to have failed. Failure detectors make decisions based on how the liveliness property has been met by communicating processes.

Liveliness is expensive in that it floods the network with "I am alive" messages which in turn have to be managed by normally expensive membership and order protocols. The thesis addresses this problem by confining liveliness messages and failure detection operations messages between two synchronously connected duplicates of a process, not the whole communicating process group. If one of or both of the duplicates violates liveliness property, the duplicates commit suicide (replica self-destruction) and or murder (destruction of the other twin replica) before issuing a signal to the group.

2.6.6 Group partitions and gossips

Group partitioning is when the processes divide themselves into at least two groups. Members of each group "think" they are the only ones alive. This leads to a complex ordering of messages to satisfy atomic broadcast properties in Table 2.3 when groups reconcile. As Dolev points out, [Dolev95] group partitions are attributed to suspicions by false failure detectors. It would be desirable to find a way of eliminating false suspicions completely thereby rule out partitions, which is what this thesis is achieving – elimination of wrong suspicions by guaranteeing fail signal thereby avoid group partitions.

2.7 Fault tolerant group communication systems

In this section, before we look at group communication systems that tolerate more serious faults such as Byzantine and malicious faults, we discuss distributed systems, in particular group communication system that tolerate benign faults. A process participating in a group displays benign faults when it fails by crashing without propagating faults to the rest of the group.

Here is a problem - to tolerate failures, redundant components need to broadcast messages among themselves to keep their states in step with one another, but this broadcasting activity is also susceptible to failures and poor latency. A process, *p*, participating in a broadcast can fail just after receiving a message, or after sending a message to a subset of the group. This is partly what failure detection protocols are tackling.

The question is at what cost (*f*) a group of processes can reach a proper agreement when some members of the group are faulty where *f* is the maximum number of processes that can fail without affecting the agreement protocol of a group communication system. The effectiveness of a group communication system protocol is judged by the maximum number of members that can be faulty without affecting the group communication system agreement protocol. This degree of resilience which is determined by the maximum number of group members that can go wrong is called a bound. Furthermore, the efficiency of a group communication system protocol is determined by how many phases and rounds it needs for group members to reach agreement in the presence of faults.

In a study where non-faulty processes communicate honestly, Pease [Pease80] addressed a problem concerning a set of isolated processors, some unknown

subset of which may be faulty, that communicate only by means of two-party messages. However, non-faulty processors always communicate honestly, while faulty processors may lie. It is shown that the problem is solvable for, and only for, $n \geq 3f + 1$, where f is the number of faulty processors and n is the total number of group members [Hadzilzcos93].

Finally there is problem with combing failure detection infrastructure with group communication system, or treating group communication system itself as a fault tolerance tool. The roles of these two systems are usually blurred together and this makes it hard to concentrate fairly on the respective types of problems. This thesis decouples failure detection and security from the upper group communication protocols layer.

2.8 Byzantine tolerant and secure group communication systems

This section discusses approaches used by researchers to deal with more serious group communication system failure. These are well known to be Byzantine faults and malicious attacks. In the following section, real life developed Byzantine tolerant and security systems are presented.

For a large class of Internet-based dependable applications (e.g., e-auctions, B2B applications etc.,), an asynchronous middleware system must be robust and responsive in the following sense: (i) it must tolerate faults more serious than crashes (robustness), and (ii) its performance should be free of liveness requirements that need to be met by making appropriate choice of values either

speculatively (as in timeouts) or randomly (responsiveness). Systems or architectures, such as ([Kihlstrom98], [Malkhi98a], [Castro99c]), which tolerate (authenticated) Byzantine faults do exist. (An authenticated Byzantine fault causes a component to fail in arbitrary manner that is however restricted by the effectiveness of message signature and authentication mechanisms such as the RSA scheme.) These systems make use of Byzantine fault tolerant protocols developed almost 'from scratch'. Baldoni *et al* derives a Byzantine protocol from a crash-tolerant one [Baldoni00]. Such protocols require at least one extra communication round than their crash-tolerant counterparts (if the latter exist) and at least $3f+1$ nodes to guarantee total order if f is the maximum number of nodes that can become faulty. They however deal with the FLP impossibility by one of the two ways attributed above to the non-partitionable approach.

There are techniques that tackle Byzantine faults and malicious faults ranging from using group communication protocol itself, probabilistic or stochastic approach, authenticated Byzantines, to cryptography. These are discussed below.

2.8.1 Group communication approach

[Reiter94d] *et al* presents a security architecture that uses a group communication protocol that extends the process group into a security abstraction. The architecture securely and fault tolerantly support cryptographic key distribution. Authors introduce novel replication techniques that they apply only when necessary. They constructed these services both to be easily defensible against attack and to permit key distribution despite the transient unavailability of a large number of servers.

Castro and Liskov's system was the first to recover Byzantine-faulty replicas proactively and it uses group communication symmetric rather than public-key cryptography for authentication [Castro00] because public-key cryptography has been the main computational bottleneck for distributed systems in the past. They used an asynchronous state-machine replication system that tolerates and recovers from Byzantine faults, which can be caused by malicious attacks or software errors and the performance bound of the system is also $3f + 1$. Authors have also proved their theories in more practical environment and the performance of the system was promising ([Castro99a], [Castro99b], [Castro99c]).

Another technique is to use group consensus model to check efficiency of two different communication protocol group systems. [Charron-Bost00] compare two approaches; asynchronous group augmented with fail detectors and asynchronous group communication. They examined whether one of these two models is more efficient and concentrated on the uniform consensus problem. The result was that they found synchronous performed better than asynchronous model augmented with a perfect failure detector.

2.8.2 Failure detectors and oracles

An introduction of fail detector has been covered already in *Section* 2.4. One of the characteristics of failure detectors is that they supply their host processes with failure suspicions of other group processes which may be inaccurate. However failure detectors can be used to detect Byzantine faults and malicious attacks by using a security infrastructure. Oracles are more special failure detectors that are based on the notion and assumption that they never fail. Pedone supports the

aims of this thesis by suggesting that it would be desirable to make sure oracles only report genuine failures but not suspicions [Pedone02].

2.8.3 Stochastic and computational techniques

In another similar computation approach, Goldereich [Goldreich00] uses a "computational model" that assumes the existence of trapdoor permutations. Trap door permutations are security structures whose functions are hard or impossible to inverse or undo to decipher security protected messages. The authors provide full proofs for the following results: secure protocols for any two-party (and in fact multi-party) computation allowing abort, as well as for any multi-party computations with honest majority. Their "...aim is to present protocols which are secure with respect to a natural (although not the strongest possible) model (i.e., the so-called static model)."

There is also a leader approach to tolerating Byzantine and malicious faults. [Gupta00] presents a scalable leader election protocol for large process groups with a weak membership requirement. The underlying network is assumed to be unreliable but characterised by probabilistic failure rates of processes and message deliveries. The system ensures good probabilistic guarantees on correct termination, and of generating a constant number of messages. Felber *et al* [Felber01] also developed other distributed systems whose group agreements are solved probabilistically.

2.8.4 Quorum systems

Quorum Systems are well-known tools for ensuring the consistency and availability of replicated data despite the benign failure of data repositories Malkhi97a]. Replicated services accessed via quorums enable each access to be performed at only a subset (quorum) of the servers and achieve consistency across accesses by requiring any two quorums to intersect [Malkhi00]. Authors considered the arbitrary (Byzantine) failure of data repositories and present the first study of quorum system requirements and construct a system that ensures data availability and consistency despite these failures. [Malkhi98b] came up with a real life application that utilised quorum model called Phalanx. Phalanx is a software system for building a persistent, survivable data repository that supports shared data abstractions. It can survive arbitrarily malicious corruption of clients and some servers.

2.9 Distributed computing systems and failure detection

The section introduces various systems that have been composed using a combination of theories discussed above. Weaknesses of such systems are highlighted in terms of failure detection and failure reporting both of which are central to this thesis. The computation systems discussed here use different approaches to achieve different performance bounds and they are operating under different environments and fault assumptions. The section particularly reviews the system's failure detection mechanisms and the extent at which it affects the performance, complexity, and semantics of a group communication

system. Systems discussed here are Antigone, Spread, Totem, Rampart, Spread, Eternal, Horus, SecureRing, and NewTOP. The section starts with general discussion of the systems them summarise the discussions in a table.

Detecting if a processes has failed in a distributed system is one the most crucial activities in building feasible distributed systems. Therefore all systems that are distributed and mostly based on group communication employ one way or the one of, or a combination of, failure detection and tolerance approach discussed in Section 2.4 above. This section scrutinises a number of failure detection mechanisms that are used by the current systems and reveals their limitations which will pave way for the thesis of this work in Chapters 3 and 4.

The most common technique used to detect a crashed process is heartbeat. A heartbeat is a periodic message sent by a process that wishes to inform other processes that it is alive and should not be excluded from membership of the group.

Antigone [McDaniel99] provides services that enable the flexible choice of which security policy to deploy for the intended group application. It offers failure detection interfaces for detection of group member failures. In the early version of Antigone the process willing to be always included in the membership had to continually transmit periodic heartbeat messages confirming their presence. This scheme does not scale well with increasing number of group members. However, MacDaniel improved it and came up with light weight failure detection using hashing and heartbeats. This improved the performance by an order of 10.

Rampart [Reiter94d] is a secure group communication that provides services that work properly even when under actively malicious and Byzantine failures. It relies on secure channels between pairs that ensure security between themselves. This

approach is very similar to the approach of this thesis except that Rampart pairs are not treated as a single process.

Lolus [Mittra97] also requires that member of the group periodically send "*I am alive*" messages to secure their membership. Each member has to refresh its membership periodically and any member that does not do that is excluded from the group. The main disadvantage of this is the member may still be alive or just slow.

Renesse and others [Renesse96] developed a service that uses Gossips to detect failures. It greatly improves on keeping the probability that a member is wrongly accused by other group members to a constant with an increasing number of members. The algorithm is also resilient to both message loss (not timing failures) and process crash.

Rampart group system provides the first Group Membership Protocol (GMP) that tolerates malicious intrusions using the technique of state machine replication to achieve high availability. The Rampart GMP uses a 3-phase commit protocol and provides strong consistency guarantees. In achieving this, Rampart system excludes detection of corrupt members i.e. it relies on an external fault detector. Public Key Cryptosystem is employed using Cryptolib package.

SecureRing employs the logical token ring architecture to solve the problem of group membership. The primary motivation of SecureRing group communication system is to reduce cost of digital signatures. System is partially synchronous i.e. local non-synchronised clocks are used for processing and message transmission timeouts. Each processor multicasts messages to all other processors. Communication between processors is unreliable.

Aqua is a dependable system that adapts to changing dependability requirements during execution of the application, enable application access to the internal dependability mechanism. The validation of the systems is determined by fault injection using Loki fault injector. Aqua supports: Multiple fault types (e.g. crash, value, and time), diagnosis and recovery from faults in a nearly application transparent manner (including killing and restarting objects), execution time translation of high-level dependability desired to particular system configurations (including replication type, voting scheme, number and distribution of replicas), multiple levels of adaptation and recovery, perhaps involving the application in a high-level way which allows separation of concerns by decoupling application specific operations from quality of service layer [Krishnamurthy02, Cukier98].

Delta-4 ([Powell88, Barrett90]) ensures that the nodes of the distributed system are implemented with extensive self-checking so that simple error processing techniques based on the fail-silent assumption can be employed, or b) providing error-processing techniques that do not require restrictive assumptions as the failure modes of individual nodes. Delta-4 is based on fail-uncontrolled nodes in that they do not possess any local error-detection mechanisms and can thus produce quite random or even malicious behaviour. In particular, a fail uncontrolled node may: (a) omit or delay sending (some) messages, (b) send extra messages, (c) send messages with erroneous content, or (d) refuse to receive messages.

"With fail-uncontrolled nodes, the replication of software components and the associated error-processing protocols must serve not only for recovery from errors but also for their detection. In order to mask t simultaneous errors, at least 2.t+1 replicates are necessary for there to be a majority of replicates executing on error-free nodes". In contrasts, replicas for this thesis system are fail-controlled have internal self checking system. The table below summarises the groups

systems just discussed. It highlights the main strength and weaknesses of the systems.

Table 2.4: Systems' failure detection and security

	Protocols	Security	Failure detection	Strengths	Weaknesses
Antigone	token ring, group session re-keying	Byzantine tolerance and malicious attacks	Secure detection of failures. 10X faster.	Security policy chosen flexibly, session re-keying	uses heartbeats, could be costly with increasing members.
Aqua	Flexible dependability requirements	Byzantine tolerance and malicious attacks	Standard, suspicions	Dependability can be applied to different layers of the system. Application, middleware, operating system	Wrong suspicions, partitions
Spread	Ring, WAN, LAN	Public cryptography group keys	Standard, suspicions	Scalable with WAN	Wrong suspicions, partitions
Totem /Transis	Token ring protocol on Transis	Basic security features	Allows gossips, suspicions, recovery, time outs	Handles Extended virtual synchrony partitions are allowed to remerge	Suspicions and negotiations, partitions, single token can get lost or corrupted.
Rampart	Use of mobile agents.	Secure point to point messages	Byzantine tolerance,	Deals with temporal specifications in real time	Stringent time outs may push failure detectors to the limit
Horus/ Esemble	HCPI – hours common protocol interface	Secure	Imperfect failure detectors	Scalable and layered, flexible service choice.	Uses gossips.
SecureRing	Ring	Digital signatures	Time out failures	Efficient signatures	wrong suspicions,

NewTOP	Group communic ation system protocols	Basic security	Time out failures and suspicions, crash tolerant	Allows group merging, multiple group membership	suspicions, gossips, membership protocol expensive

2.10 Summary and conclusions

2.10.1 Problems

The chapter has discussed key distributed computing challenges that the computing research community is addressing from theoretical and practical point of views. The challenges include the FLP impossibility result that stems from the fact that it is hard to distinguish a failed process from a slow one over an asynchronous network. Related to this problem is the difficulty of getting the true snap shot of the state of a distributed system. Other related hurdles are partial failures of a distributed system and the resulting non-trivial task of distributed processes reaching a consensus on the same value – agreement protocol. An example that requires group members to reach a consensus is when the distributed group members have to agree upon which member, among the group members, is dead or alive. This compilation of a list of active group members is called a *membership protocol.* The chapter also covered more serious threats to distributed computing such as Byzantine failures.

2.10.2 Current solutions

It is important that efficient group failure detection is deployed by a group communication system. Various techniques used by different group communication systems were discussed. In a situation where security of the application is of rave concern, a distributed system has to go further into protecting the system by deploying a security feature. Solutions to the above problems are imperfect failure detectors, stochastic and computational techniques, quorum systems and Authenticated Byzantine agreement.

2.10.3 Limitations of the current solutions

The current solutions are still faced with problems. Most systems assume the processes fail only by crashing. This assumption is now unrealistic considering not only the complexity of today's distributed systems but also the will to harm distributed computing systems by adversaries.

The systems are still faced with the problem of group partitioning. Solutions that deal with merging groups end up being complex and hard to specify. This is mainly due to the imperfect failure detectors that are augmented to their group communication processes. The failure detectors may lie about the life or death of other members which may force all group members to engage in an agreement protocol. The reported error could also be late which may harm the agreement performances. It is these agreement protocols that are source of poor system performance. Agreement protocols are not just used for group membership, but also application specific messages.

All the group communication systems cited above assume that processes fail in a benign manner: either crash or occasionally omit to produce a response (omission failures). However, field and experimental data collected indicates that there is a non-negligible risk of data corruption due to software failures [Chandra98, Sullivan91]. Consequences software corruptions are difficult to predict, as the error detection latency may be arbitrarily long. Such corruptions can cause an application process, or the underlying operating system to crash, 'hang' or omit producing a response (all of which are benign faulty behaviour), and also cause erroneous values to be output. A paper analyzing software defect reports collected between 1986 and 1989 for the IBM MVS operating system showed that 15% to 25% of faults (referred to as 'overlay faults' in the paper) caused corruption of data [Sullivan91]. Secondly, the impact of these faults was much more serious than the remaining faults. Thirdly, only 39% of the overlay faults were detected as addressing violation. A fault injection experiment to determine 'how fail stop are faulty programs', indicated that 7% of failures led to corruption of data [Chandra98]. Chandra and Chen proved that not all failing system obey fault stop fault model. In other words they make noisy failures. They recommend the use of transactions to increase dependability of their systems. Transactions reduced 7% of noisy failures to 3%.

2.10.4 Contributions of this thesis

This work improves on what the distributed community have achieved by decoupling security infrastructure from the upper middleware layer and even crucially augmenting the system with an effective fail-signalling layer. Each member of a distributed system is duplicated into a self-checking failure signalling

process. The replicas of the process are each placed on different nodes and the communication link between the duplicates is assumed synchronous, that is transmissions delays are bounded and known, while the link between two different group members may be asynchronous. This structure enables us to pull down the failure detection mechanism from the upper group layer then confine it within two duplicates that either produces a correct result or a failure signal *all* the time. If the member sends a duplicate application or failure message, one of the duplicate messages is suppressed on receipt.

This was constructed by developing a fail-signalling infrastructure so that it is seamlessly connected to a group communication system. A security feature was introduced into the fail-signalling layer. The security features proved not to be that fast at programming level. To curb the digital signature libraries performance problem another version of the fail-signalling systems was also developed by replacing the digital signatures with Message Authentication Code (MAC) hashing algorithms.

A more detailed account on the contribution of this work follows in the next Chapters.

Chapter 3

THE FAIL-SIGNALLING SERVICE

3.1 Introduction

This thesis presents an approach that extends a crash-tolerant middleware system into an authenticated Byzantine tolerant one with small modifications to the original system. State machine replication is used to assure signal-on-failure (fail-signal) semantics at a level where existing crash-tolerant services can be seamlessly deployed. The resulting system can provide total ordering that has no *liveness* requirement for termination.

In the next Chapter 4, we demonstrate the effectiveness of our fail-signalling approach by porting a crash-tolerant group communication service, (NewTOP) system, into a Byzantine-tolerant Fail Signalling group communication service (FS-NewTOP). This is achieved by augmenting the fail-signalling infrastructure discussed in this chapter into a group communication system.

This chapter first presents the problem of constructing a Byzantine tolerant group communication system in the context of fail-signalling. It then moves on to describing how a fail-signalling process is constructed before going into details of the implementation of such a FS process. The chapter finishes by demonstrating how the constructed fail-signalling process can be optimized in fault free runs so that the performance of the FS service remains steady especially in operating environments that have very low failure rates.

3.2 Fault model

Availability: clients and servers can crash, network cannot physically partition or experience high delays. Outside attackers: Cannot learn information exchanged by the servers, or server and clients Impersonate participants, inject/modify/replay data. Inside attackers (compromised servers or clients): Replicas always behave correctly or they send a fail signal. The system protects against eavesdropping, prevents impersonation, injection, modification, replay attacks.

3.3 The problem

We address the problem of building a Byzantine fault tolerant group-communication middleware system for asynchronous communication networks. Such a system offers services that simplify the development of distributed applications over an asynchronous network (e.g., the Internet) which supports operational processes to exchange messages but guarantees no bound on message delays. The services include: reliable multicast, causal order and total order (or atomic multicast). Total order is essential for replicating application servers that are state machine and providing fault-tolerant services. Atomic broadcast is also harder to achieve and the difficulties are epitomised in the FLP impossibility result [Fischer85].

3.3.1 Dealing with FLP impossibility

Crash-tolerant middleware systems deal with the FLP impossibility in one of three ways. In *partitionable* systems (e.g., [Cristian99, Dolev96, Ezhilchelvan95]), middleware processes that do not suspect each other remove from the group

those processes which they suspect to have crashed. Since the bound on message delays is not known precisely, suspicions can be false; this can lead to connected, operational processes being split into sub-groups (logical partitions) even when no process has crashed. Group-splitting thus reduces the fault-tolerance potentials of a group; merging the partitioned sub-groups and ensuring state reconciliation is a hard problem and an automated solution typically requires considerable message and time overhead [Lotem97].

3.3.2 Liveliness and fail-signalling

This problem does not exist in a non-partitionable system such as [Felber98a]. However, termination of a deterministic total order protocol is guaranteed, even in failure-free runs, only when message delays over the asynchronous network are perceived to remain stable for a suitably long duration. [Chandra96] precisely states this requirement in its weakest form as ◊w. Chandra's and Toueg's provided a solution to Fischer et al FLP impossibility results by investigating and developing solution for the weakest possible, failure detector (◊w) that can be used to reach consensus in a group communication system. Table 2.2 in Section 2.4 presents classes of other classes of failure detectors. If, on the other hand, non-deterministic or randomized protocols are used, termination requires that the random choices made converge (i.e. have tendency to suggest a definite agreed upon solution) and this is guaranteed, only in probabilistic terms, to be a certainty with the passage of time [Ezhilchelvan01]. Such requirements for termination are often called the liveness requirements. The requirements make it hard to predict performance and to provide meaningful performance guarantees to applications: total-ordering latency is influenced by how early the liveness requirement is met during the protocol execution which, in turn, depends on the choice of values assigned to the protocol parameters. For example, in a ◊w-based system, when

timeouts chosen for suspecting failures become small compared to actual message delays, it postpones the realization of ◊w and increases the latency even in failure-free runs; setting long timeouts, on the other hand, slows down failure detection and affects the performance when failures do occur.

For a large class of Internet-based dependable applications (e.g., e-auctions, B2B applications etc.,), an asynchronous middleware system must be robust and responsive in the following sense: (i) it must tolerate faults more serious than crashes (*robustness*), and (ii) its performance should be free of *liveness* requirements that need to be met by making appropriate choice of values either speculatively (as in timeouts) or randomly (*responsiveness*). Systems or architectures, such as [Kihlstrom98, Malkhi98a, Castro99], which tolerate (authenticated) Byzantine faults do exist. These systems make use of Byzantine fault tolerant protocols developed almost 'from scratch'. Baldoni *et. al.,* derives a Byzantine protocol from a crash-tolerant one [Baldoni00]. Such protocols require at least one extra communication round than their crash-tolerant counterparts (if the latter exist) and at least $3f+1$ nodes to guarantee total order if f is the maximum number of nodes that can become faulty. They however deal with the FLP impossibility by one of the two ways attributed above, which are liveness and randomization algorithms, to the non-partitionable approach. Chapter 2 describes this systems in detail.

In contrast, a fail-signalling process signals a failure when ever there is one. This eliminates the need to constantly check (polling) the node if it is alive since that node will signal a failure once it becomes faulty.

We achieve the objective of robustness by considering authenticated Byzantine faults and responsiveness by seeking an alternative way to deal with the FLP

impossibility. We observe that (a) the FLP impossibility applies only to crash model (unannounced stopping) and not to announced crashes, and (b) a fail-stop process [Schlichting83, Schneider84] which has been built with internal redundancy to tolerate authenticated Byzantine faults, can be guaranteed to announce its crash to its environment. These observations form the rationale for our structured approach: we first build every middleware process as a pair of self-checking processes on distinct nodes, and then construct a middleware system out of such middleware processes, termed as the *fail-signal* processes, whose failure modes are as follows.

3.3.3 Properties of fail-signalling process

A fail-signal process fails only by outputting fail-signals that are unique to that process. More precisely, a faulty fail-signal process

(fs1) outputs its fail-signal whenever it cannot produce a correct response, and
(fs2) may also output its fail-signal at arbitrary timing instances.

Two important remarks are in order:

Remark 1: By *fs1*, whenever a response is expected of a fail-signal (FS hereafter) process, it is produced; it is always correct if it is not a fail-signal. Note however that the outputting of a fail-signal does not necessarily mean that a response from the signalling process was expected nor that the process has crashed see *(fs2)* above. Thus, a faulty FS process, say p_i, behaves like a correct process whose responses pass through an adversary who is restricted only to substituting an arbitrary subset of p_j's responses with p_j's fail-signals or to randomly emitting p_j's fail-signals. A fail-stop process [Schlichting83] – the inspiration for our FS

process – offers stronger failure-guarantees: its fail-signal is a sure indication of its crash and its pre-crash states are preserved. Because of this, a 3-fold redundancy is needed for fail-stop construction, whereas a 2-fold redundancy will suffice for constructing an FS process. (Details in Section 3.3.)

Remark 2: Since a signalling FS process is necessarily faulty, a process that receives a fail-signal can correctly regard the source to be faulty; i.e., failure detection does not involve choosing appropriate timeouts which cannot always be done correctly over an asynchronous network. Thus, with the FS middleware processes, the FLP impossibility result ceases to hold and it is possible to build a *deterministic* total order protocol that neither tends to split groups in the absence of failures nor requires the existence of $\Diamond w$ (or a similar liveness requirement [Castro99]).

3.4 Construction of Fail-Signal (FS) Processes

3.4.1 Assumptions and principles

To transform a middleware process p into an FS p, p must be a deterministic state machine in the sense that the execution of an operation by p in a given state and with a given set of arguments must always produce the same result (requirement ***Remark 1***). Middleware processes that implement deterministic algorithms and protocols satisfy *Remark 1*. We construct FS p by hosting a replica pair, denoted as $\{p, p'\}$, on distinct nodes as shown in Figure 3.1. (Throughout this chapter, the replica pair of an object or process x is denoted as $\{x, x'\}$.) We make the following assumptions.

Assumption ***A1***: The nodes are assumed to be correct (i.e., non-faulty) when they are paired at start-up time. If one of these two nodes fails, the other is assumed to work correctly until the system detects this fault.

Assumption ***A2***: The nodes are connected by a reliable, synchronous communication link (LAN) that delivers messages within a known bound δ.

Assumption ***A3:*** Suppose that both nodes are non-faulty and an input is submitted to both of them at the same time t for processing. Say, p (respectively p') processes that input and generates a result at time $t+\Delta t$ (respectively at $t+\Delta t'$). We assume that $\max\{\Delta t, \Delta t'\} \leq \kappa * \min\{\Delta t, \Delta t'\}$, for some known, positive number κ.

Assumption ***A4:*** Similarly, suppose that both nodes are non-faulty and p and p' schedule a *send_result()* operation at the same time s to forward their result to the other replica. Say, p (respectively p') completes the send operation at time $s+\Delta s$ (respectively at $s+\Delta s'$). We assume that $\max\{\Delta s, \Delta s'\} \leq \delta * \min\{\Delta t, \Delta t'\}$ for some known, positive number δ.

Assumption ***A5:*** A3 and A4 require that the maximum difference between the delays for processing and scheduling of middleware messages, be bounded and known. Finally, a process of a correct node can sign the messages it sends and the signed message cannot be generated nor undetectably altered by a process in another node.

3.4.2 A fail-signalling pair

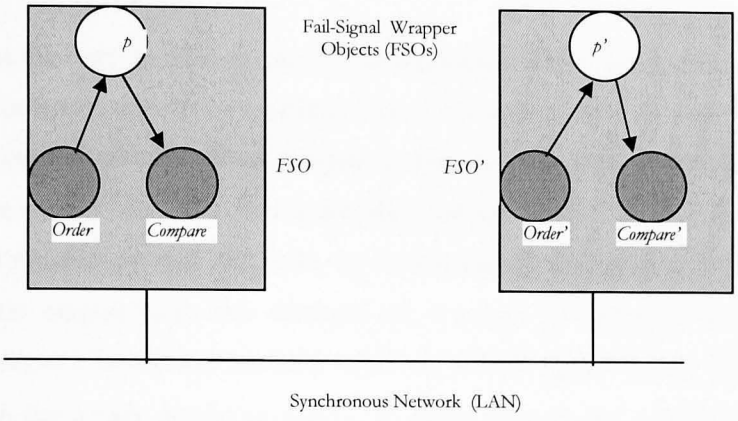

Figure 3.1: Architecture of Fail-Signal Wrapper Objects for middleware process p.

The pair {*p, p'*} is made to act as a self-checking pair by means of process pairs {*Order, Order'*} and {*Compare, Compare'*} which are threads (like *p* or *p'*) within a fail-signal wrapper Object pair {*FSO, FSO'*} (see Figure 3.1). A message destined for FS p must be received by both the wrapper objects *FSO* and *FSO'*. The Order process pair ensures that the inputs are submitted to *p* or *p'* in an identical order. The Compare processes check if *p* and *p'* generate identical outputs. If so, the output is transmitted to the destination(s), together with verifiable evidence (see below) that output checking has been carried out. Note that if a destination is an FS process, then each Compare process transmits the output to both the replicas of the destination FS process.

When *Compare* (of *FSO)* receives an output generated by *p*, it signs it and forwards a copy to *Compare'*. If it receives a signed message of identical contents from *Compare'* within a certain timeout, it signs the received message and outputs the double-signed message which will be regarded as an output of FS *p*. Similarly, *Compare'* will output a double-signed message where the first-signature is by

Compare. An output from FS p is *valid* only if it bears the authentic signatures of both *Compare* and *Compare'*.

At the start-up time or pre-processing phase, when both nodes are correct, each Compare process is supplied with a fail-signal message signed by the other Compare process. When *Compare* decides that an output produced by *p* could not be successfully compared within the timeout, it signs the fail-signal supplied to it at the start-up time and emits the double-signed fail-signal to the destination(s) of that output; from this moment on, it ceases to exchange messages with the remote *Compare'* and instead it sends the double-signed fail-signal to destination(s) of any locally produced output; it also replies to the sender of any incoming message with the double-signed fail-signal. That is, *Compare* of a correct node, after detecting a failure in the other node, sends a (double-signed) fail-signal to all entities that are expecting a response from the FS p.

Observe that when both nodes are correct, two valid outputs are generated – both having identical contents and been signed by both *Compare* and *Compare'* but in different order. When faulty *p'* generates no, late, or incorrect output, *Compare* starts emitting double-signed fail-signal to nearest destination replicas that expect a response from the FS p; further, *Compare* stops its interacting with *Compare'*, leaving the latter unable to produce any valid output. Thus, an FS p can fail only by emitting a fail-signal that can be uniquely attributed to it. It is possible that a node fault can cause the local Compare process to emit fail-signals arbitrarily. This leads to *fs2* described earlier.

3.5 Implementation of fail-signalling pair

The construction of fail-signal processes is based upon University of Newcastle's earlier work on the construction of fail-silence processes [Brasileiro96, Black98]. The key idea in the construction of a fail-silent process is similar to that of fail-signal processes except that no fail-signals are emitted. A fail-silent process (or object) is made up of a self-checking process (or object) pair. The pair receives the same set of requests in the same order, compute the requests, and then compare each other's results. If the results differ, the replicas stop functioning by committing suicide and refrain from propagating any output to the environment. The fail-silent process has been implemented in both C++ and Java. For this thesis, the implementation of fail-silent processes has been enhanced to include fail-signalling aspect and to run in a CORBA environment so that it can be integrated with NewTOP group communication system introduced in Section 3.1. The detailed fail-signal implementation are as follows:

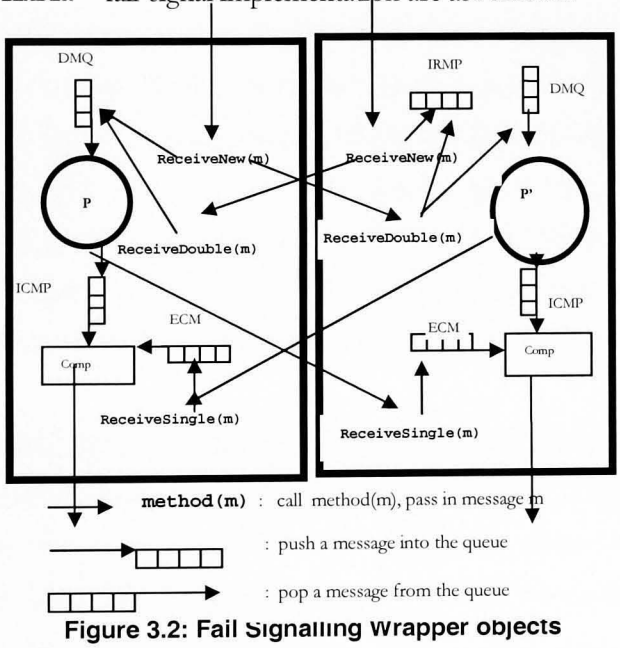

Figure 3.2: Fail Signalling Wrapper objects

Figure 3.2 illustrates the internal structures and the inter-workings of the Fail-Signal wrapper Objects (FSOs). An FSO consists of two threads: the Compare process and the replica of target process that needs to be made into an FS process. FSO is the leader and FSO' is the follower. If an input is from another FS process, it is checked for authentic, double signature; this is implemented in the `receiveNew(m)` method which, at the leader FSO, places the received input into the local Delivered Message Queue (DMQ) and then sends a copy to the follower by calling the follower's `receiveDouble(m)` method.

When the follower receives a message from the leader, it places it into the local DMQ; a copy of the message is also deposited in the Internal Received Message (IRMP) Pool. When the follower receives a valid message via `receiveNew()` method, it performs a different task to that executed by the leader. As it gets the message, it checks if the message is in the IRM Pool and if so, the pair is deleted.

Otherwise the follower stores it in the IRM Pool with an associated timeout t1. If the message is not received from the leader within t1, the follower dispatches the message to the leader by calling the `receiveDouble()` of the leader. The message within IRM Pool is given a new timeout t2. If this second timeout expires and the message has not been received from the leader, the follower assumes the leader has failed and starts emitting fail-signal to appropriate destinations. In the implementation the t1 is set to 0, and t_2 is set to 2_d.

The target thread selects a message from DMQ, processes the message, and may produce an output message. A copy of an output message is signed once and transmitted to the other target replica by calling the `receiveSingle(m)` method of the latter. The unsigned message is stored in the Internal Candidate Message Pool (ICMP), setting a timeout. When a single signed message is

received, it is placed in the External Candidate Message Pool (ECMP). The Compare thread compares relevant messages in ICMP and ECMP. If the comparison indicates that both messages contain identical result, then the comparison is deemed successful, the message from the ECMP is signed again, and the doubly signed message is sent to the destination(s). If the comparison fails or if an ICMP entry times-out, the Compare thread starts emitting fail-signals to appropriate destinations.

Observe that the simple, asymmetric, leader-follower arrangement guarantees message ordering when the leader is correct. If the leader is faulty, any out-of-order processing will manifest as a failure in the output comparison, causing the follower FSO' to start emitting fail-signals.

3.6 Optimizing a fail-signalling pair

In the order protocol component, synchronized clocks are used and the main issues to consider for optimisation are the message delays, the clock skews among the two host processors, and the concept of *stable* message sequence number. Say a message is sent at T and ε is the maximum clock skew between any two clocks, while δ is the upper bound of a message delay possible, and then the message becomes stable at:

$$T + \Delta, \Delta = \delta + \varepsilon$$

Because of the increasing timestamp order, already delivered messages are discarded.

One parameter to consider in determining the efficiency of an order protocol is called *actual stability delay* ($\sum_a$). This is the time elapsed between when a particular

a message m_a is received by a processor and when that message is queued in the DMQ$_i$. With assumptions that there is a negligible difference between *reference* clocks any correct clock when intervals ε, δ, and Δ are measured. $\sum_a$ will be presented:

$$\Sigma_a = \Delta + \min\{\alpha\varepsilon, \lambda_a\}$$

Where λ_a ($\lambda_a \geq 0$) is the magnitude if the message skew according to the reference clock, ε is the magnitude of the actual clock synchronization error at the time the message is first received from the network, α is the *ahead* factor. Alpha(α) is assigned **1** if the processor receiving message m_a is ahead of the other processor, or α is assigned **0** of the receiving processor is not ahead, or λ_a=0. Further more:

$$\Sigma_{\min} \leq \Sigma_a \leq \Sigma_{\max}$$

which brings us to:

$$\Sigma_{\min} = \Delta; \Sigma_{\max} = \Delta + \varepsilon; and \Delta \leq \Sigma_a \leq \Delta + \varepsilon$$

The implication is that there is a fixed overhead of at least Δ units of time. The motivational aim is to squeeze as much as possible either the Δ or ε or both to 0 in the above formula. The best case scenarios are when:

$$\lim_{\varepsilon \to 0} \Sigma_a = \Delta$$

or

$$\lim_{\Delta \to 0} \Sigma_a \leq \varepsilon$$

Or even better

$$\lim_{\Delta+\varepsilon \to 0} \Sigma_a = 0$$

Therefore the aim is to improve the order protocol while leaving the compare protocol unchanged.

3.6.1 Improving synchronized clock algorithm

Brasileiro and others noticed that they could improve Δ by using best timestamps (quick arrivals) of the received messages to dynamically determine the subsequent time constraints on the received messages. This is achieved by stabilization interval, which is quickly arriving messages to almost immediately. This is well illustrated in Figure 5 of [Brasileiro96]. The question is how prolonged can the tuning of this constraints be without increasing the failure rate as a result of more and more messages failing to meet the time interval delays?

We go back to the target of optimizing $\sum_a$ to be as close to 0 as much as possible. Let us assume that the first processor to receive message m_a from the other processor at time T_{first}. The other processor will receive the equivalent message from the first processor at $T_{first}+\delta_a$. Because this second processor will have the T=($T_{first}+\delta_a$) greater than its current clock T_{first} , it will immediately order the message to the DMQ_i. There is not much about the second processor beyond ordering the message into the DMQ_i. However the first processor will have to order the message as soon as it has determined the message to be stable. The message can be declared stable and sent at time $T_{first}+ \perp$ or if the first processor is lucky the message can be declared stable and sent to the network at time $T_{first}+ \lambda_a + \delta_a$, if $\lambda a + \delta a \leq \perp$. This means the first processor received the message from the other processor before $T_{first}+ \perp$. We therefore have:

$$\Sigma_a = \min\{\Delta, \lambda_a + \delta_a)$$

and from,

$$\Sigma_{\min} \leq \Sigma_a \leq \Sigma_{\max}$$

and

$$\Sigma_{min} = 0, \Sigma_{max} = \Delta$$

we have a more attractive stability delay of:

$$0 \leq \Sigma_a \leq \Delta$$

3.6.2 Order protocol optimization in fault free runs

Where there are no physical clocks to synchronize the nodes, the Lamport logical clocks are used for generation of message time stamps between two replicas of a fail-signalling entity. In the order protocol each processor keeps its own local time and the estimate local time of the other processor. Hence we have two logical clocks in each processor: local logical clock (LLC) and remote logical clock (RLC) both of which are defined in [Barasileiro96]. When a processor send a message it sets its RLC to either the timestamp of an incoming message T and sets its LLC to itself or T+1 which ever is greater.

An interesting aspect of the properties of the protocol: messages are relayed to the neighbour (leader or follower) bearing timestamps and RLC is smaller than local LLC and remote LLC. This RLC is the threshold for stable messages for ordering. The actual stabilisation delay will be:

$$\Sigma_a = \lambda_a + \delta_a$$

However there is a problem: only after the arrival of the RLC can the local message become stable and be ordered.

Solution: timeouts. At time $t + 2\delta$, RLC is updated to T if its value is less than T. We now have:

$$\Sigma a = \min\{2\delta, \lambda_a + \delta_a\}$$

and with our optimised

$$\Sigma_{\min} = 0, \Sigma_{\max} = 2\delta$$

we have

$$0 \leq \Sigma_a \leq 2\delta$$

If our fail-signalling process is operating in a fault free environment, and then we have an approximation of:

$$\varepsilon = \frac{1}{2}\delta$$

The above range of Σ_a can now swallow the δ and be expressed on ε thus:

$$0 \leq \Sigma_a \leq 4\varepsilon$$

This expression implies that if we operate in failure free environment, it is attractive that we concentrate in the clock skews (ε) than the network link delays (δ). This makes our fail signal implementation less costly to as part of the target system when failures are rare, while retaining failing signalling (FS) properties.

3.7 Concluding remarks

We have constructed a fail-signalling service that duplicate the target application component to turn into a self checking fail-silent and signalling entity. The idea of signal-on-failure is not new and it is one of the three failure-guarantees offered by the Fail-stop processes of [Schlichting83]. A Fail-stop process requires (at least) three (internal) replicas, while an FS process can be done with a replica pair. A significant benefit of our fail-signal based approach is that the FLP impossibility result derived for unannounced crashes ceases to apply and consequently the total ordering is guaranteed to terminate so long as the asynchronous network does not suffer permanent partitions.

The assumptions made in the construction of FS processes have implications at the application and middleware levels. Assumptions A3 and A4 (in Section 3.3) require that the maximum delay within which the replicas of an FS middleware process complete processing of an incoming message or scheduling an outgoing message, must be known and bounded.

Fail-signal construction also assumes that the two nodes of an FS process are connected by a synchronous link (assumption A2) and that no more than one node becomes faulty (assumption A1). A2 can be realized, say, by keeping each pair of nodes geographically close and making use of the fast Ethernet technology.

The next chapter shows how to construct a Byzantine-tolerant, group-communication system by extending a crash-tolerant system. The extension involves replacing crash-prone middleware processes with fail-signal or FS processes that are self checking and secure.

Chapter 4

A FAIL-SIGNALLING GROUP COMMUNICATION SYSTEM

4.1 Overview

The objective of this thesis is twofold: to 1) demonstrate that our fail-signal (FS) based approach can considerably simplify the middleware construction by decoupling authentication and group membership protocol from the middleware and 2) to evaluate the performance degradation when crash-tolerance is swapped for (authenticated) Byzantine tolerance. When the construction of a deterministic, crash-tolerant middleware system and that of FS processes conform to the same standard, integration that leads to a Byzantine-tolerant middleware system requires very little code change to the original crash-tolerant system. As a proof of concept we enhance the NewTOP group communication system (which is a CORBA compliant, crash-tolerant, and partitionable middleware system [Morgan00a, Morgan00b]) with FS middleware processes. The enhanced NewTOP, called the FS-NewTOP hereafter, tolerates authenticated Byzantine faults and does not lead to group partitioning. We then measure the ordering latencies of FS-NewTOP and the original NewTOP under those conditions that put the components of FS processes under maximum processing load.

This chapter shows how to construct a Byzantine-tolerant, group-communication system by extending a crash-tolerant system. The extension involves replacing crash-prone middleware processes with fail-signal or FS processes that are self

checking and secure. The chapter describes the existing NewTOP and how the FS-NewTOP is constructed as an extension of the existing system.

The next chapter, Chapter 5, presents the experimental set-up, measures the latency and throughput cost extracted by this extension.

4.2 The NewTOP Group Communication Service

The Newcastle Total Order Protocol (NewTOP) is a CORBA compliant, crash-tolerant, partitionable middleware system. The system implementation is centered on a CORBA node called the NewTOP Service Object (NSO). NSO is the pinnacle of the design of NewTOP in that is does total order, multicasts, inter group communication. There is one NSO for each group member. When application processes want to form a group with a common goal and to avail themselves group communication services, each process is allocated an NSO as shown in Figure 4.1. An application process A_1 acts as a 'client' to its NSO in obtaining group communication services from the latter. The communication between A_1 and its NSO, and the communication between NSO's themselves are handled by an ORB.

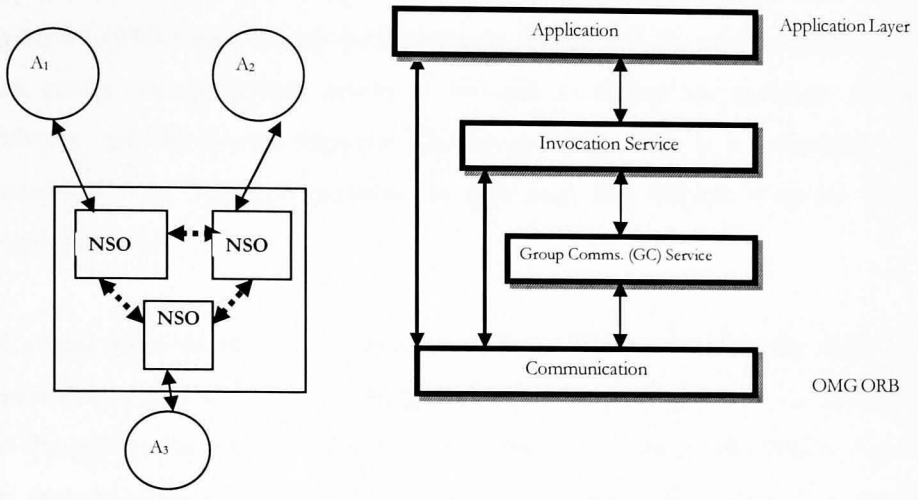

Figure 4.1: The NewTOP service: access and structure.

An NSO and its application 'client' need *not* reside on the same host, for the reasons that NSO is a CORBA object and the communication between an NSO and its client is handled by the ORB (*location independence*) [OMG00]; however, for performance reasons, they are normally hosted by the same node. Further, NewTOP requires an A_i to be member of a group in which A_i intends to multicast and permits A_i to be a member of more than one group at the same time. Being a partitionable system, it does not however support merging of partitioned sub-groups.

An NSO comprises of two subsystems: Invocation service and Group Communication (GC) service. The former allows the application to specify the type of NewTOP service needed and marshals a multicast message accordingly. The latter implements protocols to provide a variety of services: symmetric total order, asymmetric total order, reliable multicast, simple (unreliable) multicast and (partitionable) group membership.

When A_i multicasts a message to the group, the message is marshaled into a generic CORBA type any by the Invocation service and the relevant protocol of the group communication service is invoked to deliver the message. At the delivery end, the reverse happens. The Invocation service at a destination end unmarshals the delivered message (of type any) and delivers it to the client application A_j.

If a host node of, say, A_1 in Figure 4.1 develops Byzantine faults, the faults can manifest at two levels. First, at the *application level*, the message which A_1 multicasts to the group may contain erroneous information. To tolerate this failure, A_2 and A_3 must be replicas of A_1; given that the latter are correct, the failure of A_1 can be masked through a majority vote. Secondly, the fault may manifest at the *middleware level*. The NSO associated with A_1, when hosted in the same node as A_1, may corrupt, probably undetectably, A_1's multicast message. NewTOP, designed to be only crash-tolerant, cannot tolerate such failures and provide correct middleware services. It is the middleware-level failures of non-benign nature that we wish to tolerate by extending NewTOP into a Byzantine tolerant one. One of the most common assumptions distributed systems experts make is, a process that is participating in a system can either produce a correct result when functioning correctly or stop producing the result at all when it fails (fail silently). Although this assumption may be adequate for non-mission-critical systems, experiments have shown that it not reasonable to make fail-silence assumptions for more life-critical or monetary-critical systems that require high integrity of computed results [Scott01].

4.3 Extending NewTOP to FS-NewTOP

Figure 4.2 depicts the structure of the FS-NewTOP system, extended from (crash-tolerant) NewTOP by using an extra node, a synchronous link to connect the node pair, and the Fail Signal Wrapper Objects whose target is the NewTOP group communication (GC) service object. The wrapping of GC is made transparent to GC. To achieve this transparency, CORBA interceptors are used similar to those in [Narasimham99b]. A call to NewTOP GC, either from the Invocation layer or from a remote NewTOP GC, is intercepted on the fly and is submitted to both GC and GC' in an identical order with the FSO acting as the leader. Similarly, a double-signed response returned by FSO and FSO' to the Invocation layer is intercepted, signatures stripped and duplicates suppressed. This interceptor based technique used here is very similar to the one used in the Eternal system [Narasimhan00]. Since the GC service is implemented as a single-threaded, deterministic application, GC and GC' run as deterministic state machines regardless of other software (e.g. CORBA) running on the host nodes.

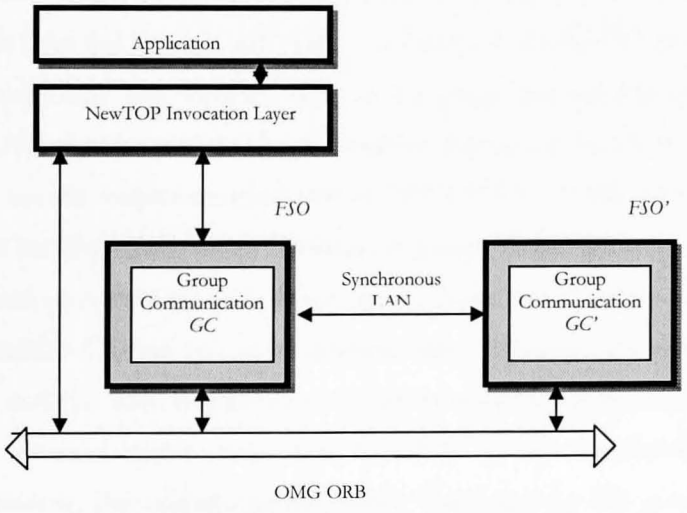

Figure 4.2. The FS-NewTOP system

Observe that the wrapper transparency means that *GC* and *GC'* regard themselves as the *only* GC service below the Invocation layer as in the original NewTOP. Futher, that the *GC'* is hosted on a different node to the Invocation layer will not matter since the communication between the two is via the ORB (which hides location) and through the wrapper object *FSO'* which is a CORBA object. Thus, with the CORBA-compliant fail-signal wrappers and the ORB technology, the extension to FS-NewTOP was seamless. Indeed, the applications can specify, as a NewTOP service option, whether Byzantine tolerance is needed or crash tolerance is sufficient. In the latter case, *FSO* will not choose to order the input and compare the output; *FSO'* will remain unused. Adding this functionality will be a future work. Below, we state the modifications to the original NewTOP necessary for the extension.

The NewTOP group membership object in GC system makes use of a failure suspector module which periodically 'pings' remote NSO GCs and generate suspicions based on a timeout mechanism. In the FS-NewTOP, a suspector module does not have to send 'pings'; instead, it converts the fail-signals received into 'suspicions' and supplies them to the group membership object. As stated earlier, fail-signals uniquely identify, and are indications of a fault at, the signalling entity; so, the suspicions generated in FS-NewTOP, unlike those in NewTOP, cannot be false. This avoids splitting of groups when there are no failures and preserves all correct FS-GCs in one group. Note also that all input messages are submitted to *GC* and *GC'* in an identical order. Therefore the suspector modules of *GC* and *GC'* send suspicions to the group membership objects of *GC* and *GC'* in an identical order. Since the NewTOP group membership protocol is deterministic, the outputs (group views) computed by the group membership objects of *GC* and *GC'* will be identical.

Referring to Figure 4.2, a non-benign failure at the Invocation layer that results in an application message being lost or corrupted can be treated as an application-level non-benign failure, mentioned earlier. Total order protocols are unconcerned with the correctness of the application-level contents of the messages they order. So, even if NewTOP-GC were to implement a $\Diamond w$ based (crash-tolerant) protocol, by the fail-signalling properties of FS-GCs, the requirements of $\Diamond w$ will be met so long as FS middleware processes are not permanently disconnected and a majority of them remain correct. The reader is referred to [Ezhilchelvan02] which transforms a $\Diamond w$ based, crash-tolerant total-order protocol for a (mixed) system of f FS processes and $(f+1)$ Byzantine-prone processes.

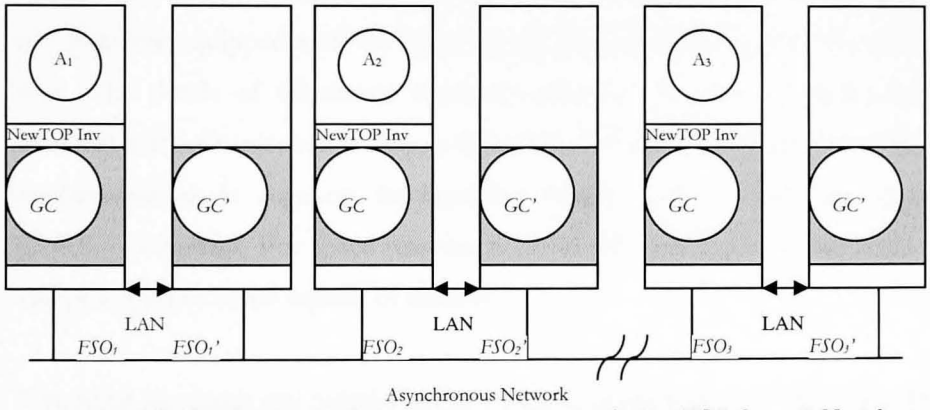

Figure 4.3: Deployment of the components of FS-NewTOP for a 3-Member Group.

Figure 4.3 shows the deployment of FS-NewTOP for a 3-member application group $\{A_1, A_2, A_3\}$. For a given i, $1 \leq i \leq 3$, FSO_i and FSO'_i are placed in nodes connected by a synchronous LAN; they are also connected to every $\{FSO_i$,

$FSO'_j\}$, $i \neq j$, by the (reliable) asynchronous network. At most one node fault (of authenticated Byzantine nature) can be tolerated by the system shown in figure 4. If the node of an FSO'_i is faulty, A_i cannot effectively communicate with the rest of the group due to FSO_i having stopped the middleware operations or fail-signals being emitted arbitrarily (by FSO'_j). If the node of an FSO_i is faulty, the following failure mode is also possible: A_i can transmit messages of application-specific erroneous contents. If A_i's are replicas of each other, a client of this replica group must multicast its request to the entire group and must majority-vote the results received from the replicas. Thus, with the FS-NewTOP, $4f+2$ nodes are needed to mask f Byzantine faults. The other cost of FS-NewTOP over NewTOP is the performance

The details of implementing fail-signal processes are very similar to our earlier implementation of fail-silence processes. In the latter, a Compare process simply stops functioning when matching of output messages does not succeed. They are not therefore equipped with the single-signed fail-signal messages at the start-up time. The details of fail-silence implementation can be seen in [Brasileiro96, Black98] and fault-injection testing in [Stott01]. This thesis worked on fail-silence implementation to augment fail-signalling feature and to make the system CORBA compliant. For space reasons, we have left the details to Appendix A, except to outline some aspects of interest.

The input messages are ordered using a simple, asymmetric protocol that does not require nodes' clocks to be synchronised. One of the wrapper objects, say *FSO*, is fixed as the Leader and the other, *FSO'*, as the Follower. The Order process of *FSO'*, *Order'*, accepts the order decided by *Order* and checks whether every message it receives is being ordered by the leader. This means that a given input is submitted to p and then to p', and the time difference can be at most δ. A

Compare process computes, for every locally produced output, the time elapsed since the corresponding input was submitted for processing (as π) and the time taken to sign and forward the output to its remote counterpart (as τ).

In summary, in the traditional NewTOP, despite to minimise incorrect suspicions by processes, it is possible for processes to wrongly suspect other processes to have crashed. In FS-NewTOP this wrong suspicions, or even suspicion protocol, are eliminated thereby making group semantics much simpler.

Chapter 5

RESULTS AND PERFORMANCE ANALYSIS

5.1 Introduction and objectives

This chapter presents results obtained from both the traditional NewTOP and the new fail-signalling and secure FS-NewTOP group communication system. The two systems are compared under two main conditions: 1) when the number of group members is increased up to 15 and the message is fixed to a very small size, and 2): when the size of the message is varied from 5 bytes to 10K bytes and the number of group member is fixed at 10.

NewTOP and FS-NewTOP configurations were tested for performance on five system properties: throughput for very small message size[1] for members ranging from 2 up to 15 members, throughput for message sizes in the 5 bytes to 10k range for a fixed 10 group membership, symmetrical order latency for very small message size for members ranging from 2 up to 15 members, symmetrical order latency for a fixed group size of 10 with messages from 0 to 10k bytes, and time taken for a group to reach an agreement for a new membership when one member fails by suddenly crashing.

As predicted our results indicate that a fail-signalling and secure group communication is possible to construct and it performs better than traditional group communications in membership protocols and eases the semantics (see

[1] Equivalent to the *micro-benchmark* technique that measures performances based on the *null* operation call.

section 5.7 and 5.8). However the traditional group communication performs better in other performance categories including throughput and order latency.

5.2 Operating environment – Mega cluster

The results were collected from 16 nodes connected by 1-Mega bits/second high speed homogenous LAN. Each node is a dual Intel Pentium III (coppermine 860.946 MHz), has 256 KB cache and 512 MB RAM. The machines are networked in a 10.x.x.x configuration that is standalone and not part of campus wide network to ensure exclusive access to the test bed. Nodes were running Linux 2.4 and Java 1.4.2 was used to compile and run the test systems.

5.3 Configuration and Object Distribution

There are five system configurations each tested for the five different properties including: 1) throughput for very small message size[2] for members ranging from 2 up to 15 members, 2) throughput for message sizes in the 5 bytes to 10k range for a fixed 10 group membership, 3) symmetrical order latency for very small message size for members ranging from 2 up to 15 members, 4) symmetrical order latency for a fixed group size of 10 with messages from 0 to 10k bytes, and 5) time taken for a group to reach an agreement for a new membership when one member fails by suddenly crashing.

The five different system configurations to be tested against the above five configurations are: 1) NewTOP without fail-signalling infrastructure, 2) FS-NewTOP without message signatures, 3) FS-NewTOP with messages signed

[2] Equivalent to the *micro-benchmark* technique that measures performances based on the *null* operation call.

with 512 bit RSA key size, 4) FS-NewTOP with messages signed with 1024 bit RSA key size 5) FS-NewTOP with HMAC-MD5.

In case of traditional NewTOP, each group object (NSO) is placed on its own exclusive node, and for FS-NewTOP each of the two replica objects (both equating to 1 fail-signalling group member object - FSO) was placed on its own node. See Figure 5.1 below of a three-member group communication system for both traditional NewTOP and the new fail-signalling FS-NewTOP.

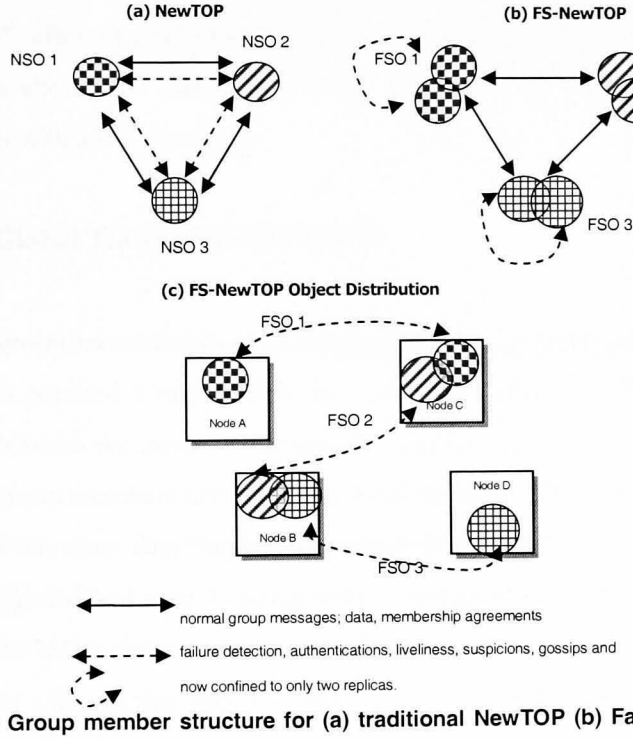

Figure 5.1a: Group member structure for (a) traditional NewTOP (b) Fail signalling and secure FS-NewTOP and (c) the object/node distribution strategy for FS-NewTOP.

The two replicas of the same FS-NewTOP member are placed on different nodes to reduce the risk of a single point of failure. Communication between the replicas of the same FSO (say FSO 1) is assumed to be synchronous therefore the duplicates have to be on the same high speed and reliable LAN connection.

5.4 Measuring group consensus latency

The purpose of task is to measure how much time it takes for NewTOP group members to reach a consensus on a new group view when one group member crashes. Performance is measured in milliseconds for how long a group of three up to ten members takes to agree upon a new stable group view if one of the members crashes. This is when a group membership performance protocol "terminates" after one-member crash. The results indicate that performance range from about 300 milliseconds with a group size of three up to 1500 milliseconds with a ten-size group.

5.4.1 The Global Timer (real time clock)

To collect group membership protocol performance, an independent global real time clock is required. Unfortunately, we cannot encapsulate the real timer within NewTOP because the group communication system uses logical clocks. Even if NewTOP group members used real time local clocks at various nodes, this would require an even more daunting task of synchronizing all the clocks to go in step with a true global real time. Consequently, a more natural way to go about this problem is to have a timer as an external observer of membership performance.
The timer is a server that logs and timestamps any event sent to it using local system clock. The three main events required in the measurements include; when one member crashes, when the crash is detected by at least one non-faulty member, and when the remaining members reach a new group view agreement. The timer is executed on a different node within the same LAN the group is running. The one-way (not return) communication latency between the sender of the event and the timer may be considered negligible and it was measured at be

0.05 of a millisecond. This low latency is achieved by using an unreliable but fast UDP socket protocol as the communication channel between the group members and the timer. The timer server reads real time from the system clock of the host node using System.currentTimeMillis() java method call. See figure 5.1b.

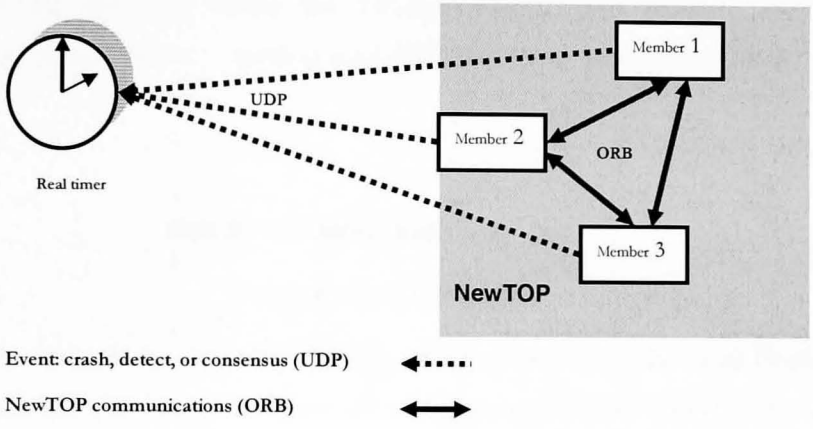

Figure 5.1b External timer for crash events and group communication systems

For example, if member 3 crashes, a crash event is sent to the timer using UDP, both member 1 and 2 may both send a crash detection event to the timer but only the earliest is recorded. Finally, both member 2 and member 3 must agree that member 3 has crashed, this is when they both send the consensus signal and the last one is the one recorded indicating all-member agreement.

5.4.2 Events: crash, detect, and consensus (determination).

A group member can send one of three types of events being crash, detection, or consensus (termination). When a group member crashes, it raises an exception which is caught and within the {} catch block within which a crash event is sent to the timer by calling the `triggerTimer(int Event)` method. The `triggerTimer()` method is a UDP client to the global timer server.

```
try
{
    . . .
}
catch (CrashException ce)
{
    triggerClock(CRASH);
}
```

The timer keeps record of the time at which it receives the crash event t(crash), then the time at which at least one of the group members detected the crash fault t(detect), then the timer records the time at which a consensus t(cons<ensus) has been reached with a new view excluding the failed member. The detection and consensus events can be sent by any un-crashed member of the group, but the timer will only pick the earliest crash event and the last consensus event.

We record both the time between crash and detection TΛ(detect), and the time between crash and consensus TΛ(consensus). From this two we calculate the actual consensus overhead between the detection and consensus (compare between crash and consensus.

$$T_{\Lambda}(detect) = t(detect) - t(crash) \qquad f(1)$$

$$T_{\Lambda}(consensus) = t\ (consensus) - t(crash) \qquad f(2)$$

$$T_{\Lambda}\ (actual\ consensus) = T_{\Lambda}(consensus) - T_{\Lambda}(detect) \qquad f(3)$$

$T_A(detect)$ is particularly important for future comparisons with fail-signaling NewTOP in Task B. The hypothesis is FS-NewTOP should yield a more superior $T_A(detect)$ because in FS-NewTOP members are duplicated to act as one and are strongly coupled, therefore the other duplicate must be quick to pick a crash fault of the other. Similarly, $T_A(consensus)$ is also expected be better in FS-NewTOP since in FS-NewTOP suspicions, which have some overhead, are eliminated and replaced with true failure signals. Section 5.5.5 presents results collected using this method.

5.5 Performance analysis

We used both the RSA and HMAC for security of FS-NewTOP. FS-NewTOP performance has generally been affected by the overhead of RSA signatures on relatively larger messages. To partially solve this, java native security RSA modules were replaced with Message Authentication Code (MAC) security for the twin-replicas of FS-NewTOP. MAC is a bit string that is a function of both data (either plaintext or ciphertext) and a secret key, and that is attached to the data in order to allow data authentication. The function used to generate the message authentication code must be a one-way function.

"A MAC is a function which takes the secret key k (shared between the parties) and the message m to return a tag $MAC_k(m)$. The adversary sees a sequence $(m_1; a_1); (m_2; a_2); \ldots ; (m_q; a_q)$ of pairs of messages and their corresponding tags (that is, $a_i = MAC_k(m_i)$) transmitted between the parties. The adversary breaks the MAC if she can find a message m, not included among $m_1; \ldots; m_q$, together with its corresponding valid authentication tag $a = MAC_k(m)$. The success probability of

the adversary is the probability that she breaks the MAC. (Notice that an adversary who finds the key certainly breaks the scheme, but the scheme can also be broken by somehow combining a few messages and corresponding checksums into a new message and its valid checksum." [Bellare96]

In this thesis, we used HMAC algorithm combined with MD5 algorithm.

5.5.1 Throughput versus number of group members

The first experiment measures the throughput of the five system configurations under maximum message load. The results were obtained by fixing the size of messages to 5 bytes, and then each member sending as many messages as possible to the group. We obtain a count of messages per second. This scenario is carried out on each of the five systems: 1) NewTOP, 2) FS-NewTOP without digital signatures, 3) FS-NewTOP with RSA size 512 bits, 4) FS-NewTOP with RSA size 1024 bits, and 5) FS-NewTOP with MAC-MD5 hashing. Figure 5.2 and Table 5.1 show the performance results for all the five group system configurations against the increasing number of group members from 2 up to 15.

Table 5.1: Throughput under maximum message load against increasing group members

No. of group members	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NewTOP	77.2	103.1	123.5	132.6	141.3	143.7	148.0	149.4	150.3	144.9	139.3	134.9	132.0	124.0
FS-NewTOP + No Sig	60.1	73.9	91.4	108.0	113.0	116.4	114.9	116.0	112.0	97.9	97.9	96.0	92.0	91.6
FS-NewTOP+ RSA 512	57.2	64.7	72.3	86.2	90.8	93.0	96.0	93.6	93.6	84.0	70.9	64.6	63.4	60.0
FS-NewTOP + RSA 1024	52.7	56.0	60.9	68.4	71.5	76.0	80.6	80.6	76.9	64.5	43.3	37.0	30.7	28.0
FS-NewTOP + HMAC-MD5	53.4	67.7	85.4	98.7	103.4	109.3	104.5	106.7	100.2	89.1	88.3	87.5	84.1	80.3

**all times in milliseconds*

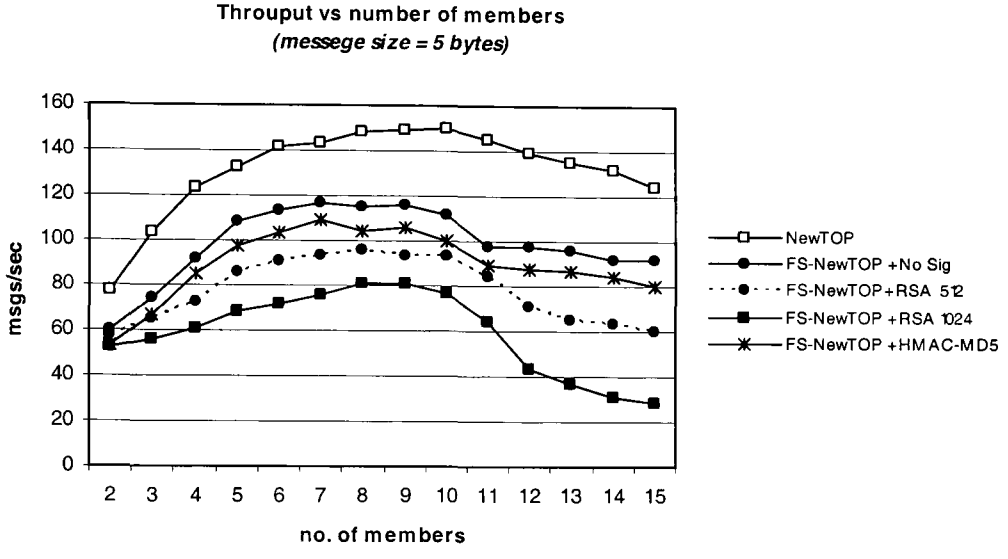

Figure 5.2: Throughput under maximum message load against varying group members

All five systems increase their performance until the number of group members reaches 10 at which they sharply lose their throughput. This unexpected pattern of performance is because NewTOP itself maintains a pool of 10 threads per group member to do the housekeeping jobs (such ordering, delivering, and sending messages) and when the number of members reaches a threshold of 10 the performance begins to degrade.

As expected, each of the four configurations of FS-NewTOP performs worse than the pure NewTOP because all the four configurations replicate each of the NewTOP members. The replicas also need the additional fail signal protocol activities such as queue management for more queues; encrypting and decrypting signatures; and handling duplicate messages. The fail-signalling FS-NewTOP without digital signatures performs between 32% (at 11 members) and 22% (at 2

members) less efficiently compared to the pure NewTOP. The FS-NewTOP with 512 bit wide RSA digital signals delivers 51% and 26% fewer messages compared with NewTOP at 15 members and 2 members respectively. The FS-NewTOP with 1024 bit RSA performed even worse than its two FS counterparts by a significant cost of signatures at 2 and 15 members compared to the original pure NewTOP.

The overhead of the FS-NewTOP 1024 is significant for all group sizes. This is because of the overhead introduced by the FS protocol itself, and the cost of digital signatures on each relatively larger message. The dwindling performance of the 1024 bit is even prominent when we change the size of the message size as explained in the next subsection. However, HMAC based FS-NewTOP performed better than both of the RSA FS-NewTOP.

5.5.2 Throughput versus message sizes

We then measure throughput with the number of group members fixed at 10 but changing message sizes from 0 to 10k bytes. The same experimental environment as described earlier is maintained. All the five systems significantly lose performance as the message size reaches 4k bytes. FS-NewTOP with signatures performs particularly poorly for large messages.

Table 5.2: Throughput under maximum message load against increasing message size

Message size (bytes)	5 bytes	1k	2k	3k	4k	5k	6k	7k	8k	9k	10k
NewTOP	146.5	123.0	99.6	85.3	78.9	74.7	68.0	64.0	60.0	57.5	51.7
FS-NewTOP + No Sig	108.3	88.5	72.0	63.3	50.7	40.5	34.3	28.5	22.2	20.6	18.2
FS-NewTOP + RSA 512	101	62.6	37.7	26.3	17.1	13.5	9.2	6.4	5.7	5	3.6
FS-NewTOP + RSA 1024	85.2	49.3	20.9	9.9	4.0	1.7	1.7	2.3	1.7	1.2	1.1
HMAC-MD5	103.7	82.0	66.8	53.3	44.6	33.1	25.6	20.9	16.3	13.6	12.3

**all times in milliseconds*

The worst performance of FS-NewTOP without signatures is when the size was 10k. It is 63% worse than the pure NewTOP when the message was 10k, and it was 26% less efficient than NewTOP when the message size was 5 bytes.

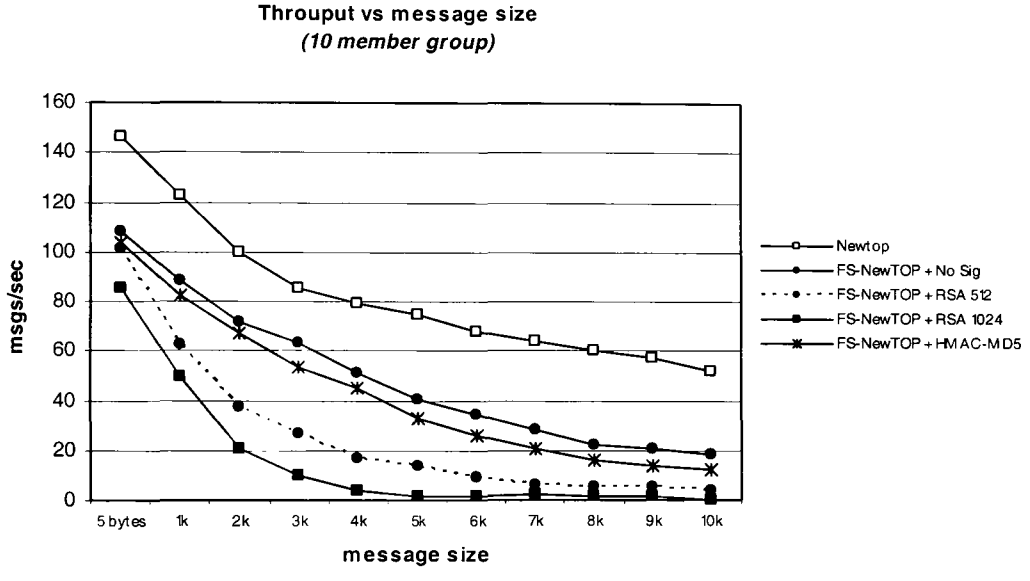

Figure 5.3: Throughput under maximum message load against varying message size

The FS-NewTOP with the 512 bit signature is not good enough as the message size larger than 4k bytes. The worst performance price is paid by the FS-NewTOP 1024 bit that almost completely stops as the system struggles with high cryptographic overhead on messages over 5k bytes. Once more, the HMAC system performed better than both of the RSA FS-NewTOP.

5.5.3 Order latency versus number of group members

Order latency is the time in ms it takes to symmetrically order messages of size 5 bytes received from non-faulty group membership where the group varies between 2 and 15. When messages have been ordered, they are ready to be delivered to the application. The results were obtained from consistently sending messages from each group member M_i at fixed intervals. The interval between the messages is fixed to δ (250 ms) for all different number of group members. 250 ms was found to be the most optimal time interval such that messages are not kept in the queues for too long and on the other hand the interval ensures that messages propagated into the system are not too widely separated. The message size is also fixed to a small size of only 5 bytes for members from 2 to 15.

Table 5.3: Symmetrical order latency against increasing group members

Nu. Of group members	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NewTOP	121	245	978	1772	2376	2639	3137	3350	3431	3782	3870	4120	4330	4470
FS-NewTOP + No Sig	300	774	1963	2750	3394	3862	4226	4552	4842	5248	5657	5763	5904	5970
FS-NewTOP + RSA 512	599	1451	2595	3439	4096	4558	5109	5360	5792	6017	6245	6562	6762	7151
FS-NewTOP + RSA 1024	642	1739	2979	3878	4610	5111	5460	6016	6285	6531	6974	7151	7525	7877
HMAC – MD5	316	835	2097	2992	3590	4135	4523	4757	5082	5578	5854	6159	6136	6475

**all times in milliseconds*

The four FS-NewTOP versions show a consistent performance correlation with the pure NewTOP as the number of members increase.

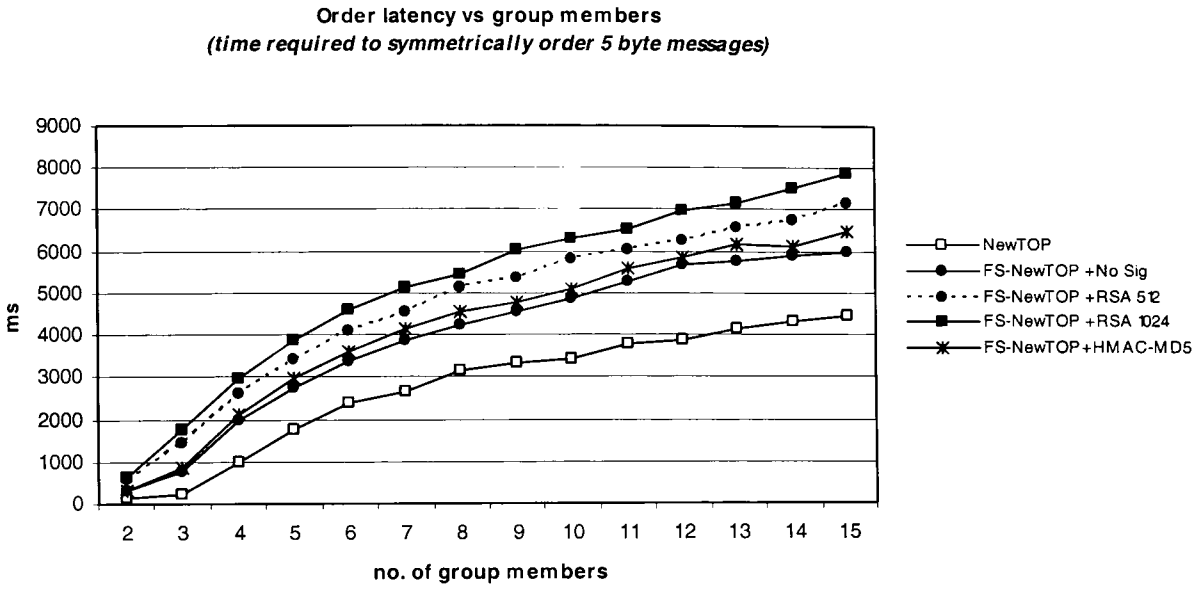

Figure 5.4: Symmetrical order latency against varying group members

The consistency of the performance for all the versions yields a correlation of coefficient close to 1. However, the FS-NewTOP does not perform as well as the pure NewTOP as the table shows and the figures indicate that the FS-NewTOP without signatures is 44% and 33% worse than NewTOP at 2 and 15 members respectively, while FS-NewTOP 512 FS-NewTOP 1024 perform less efficiently compared to NewTOP at 2 and 15 members respectively.

5.5.4 Order latency versus message sizes

The same order latency experiment is conducted with group membership size fixed at 10 but message size increases from 5 bytes to 10k bytes.

Table 5.4: Symmetrical order latency against increasing message size

	5 bytes	1k	2k	3k	4k	5k	6k	7k	8k	9k	10k
NewTOP	3188	3535	4671	5369	6697	7595	8268	8729	9409	9865	10108
FS-NewTOP + No Sig	3851	4543	5991	7642	8919	9730	10790	11379	12387	13360	13743
FS-NewTOP + RSA 512	5749	6468	8133	9600	10791	11855	13432	14613	15412	15889	16200
FS-NewTOP + RSA 1024	6307	7188	9239	11030	12590	14224	15647	16749	17549	18390	18573
HMAC – MD5	4234	4781	6562	8037	9652	10532	11917	12499	13429	13917	15206

**all times in milliseconds*

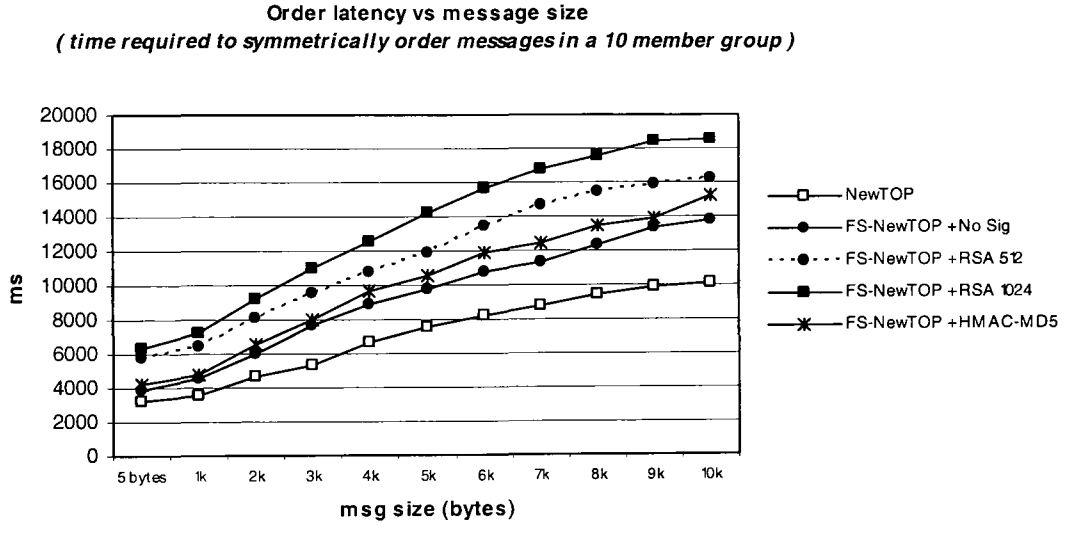

Figure 5.5: Symmetrical order latency against varying message size

5.5.5 Cost of consensus when one member fails (varying members).

This experiment measures the time in milliseconds, which is taken for the remaining non-faulty group members to reach an agreement of the new consistent membership view at each member, *vi*, when one member suddenly fails by stopping. Messages were sent from each member, and one group member was crashed during this process. Time is then recorded from the moment one member crashed until all members agreed upon new membership. The results below indicate a performance advantage for NewTOP over the traditional FS-NewTOP. We start with three members (not two) so that we are left with a minimum of two members when one member crashes. Note that because a signaled failure from an FS-NewTOP is a guaranteed failure, the receiving members do not have to negotiate with other members in order to establish if indeed the member M_f has actually failed.

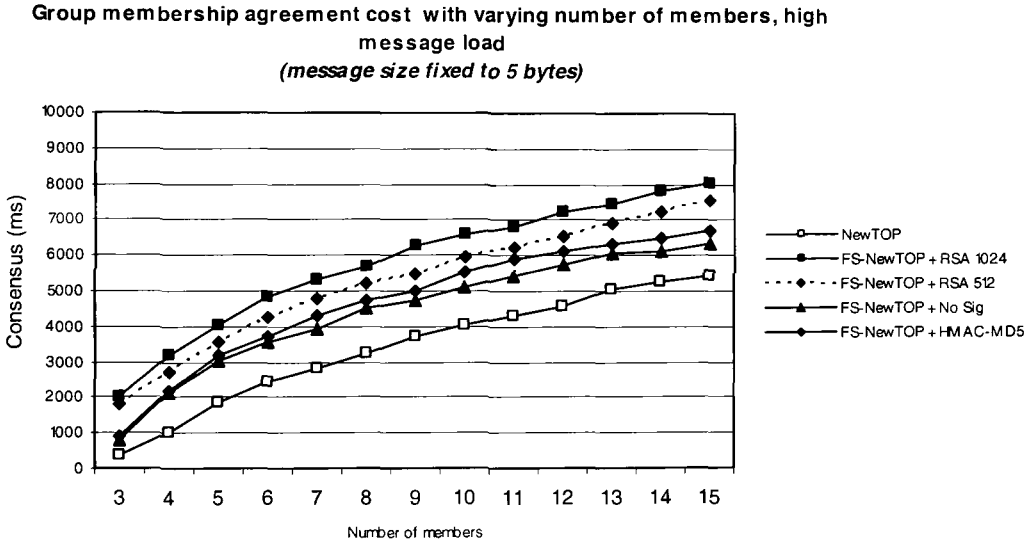

Figure 5.6: Membership protocol overhead when one member suddenly fails (varying number of members, fixed message size to 5 bytes)

When a non faulty member receives a fail signal message, it immediately updates its view without group view consensus broadcasts with other members. That is why in this experiment the pure NewTOP performs worse in group membership protocol than all the four FS-NewTOP versions for the first time.

5.5.6 Cost of consensus when one member fails (varying message size)

We measured the time in milliseconds it takes for the remaining non-faulty group members to reach an agreement of the new consistent membership view at each member, vi, when one member suddenly fails by stopping. But instead of taking measurement against number of members, we varied the message size instead and fixed the number of group members to 10.

This experimental setting produces interesting results in that for small message sizes over 3k, NewTOP performances better than FS-NewTOP 1024, FS-NewTOP 514, FS-NewTOP H-MAC, and FS-NewTOP without digital signature (see figure 5.7). This result show that digital signatures gradually take a longer time to process as the message size increases.

Group member cost agreement with varying message size, high message load (*group membership fixed to 10)*

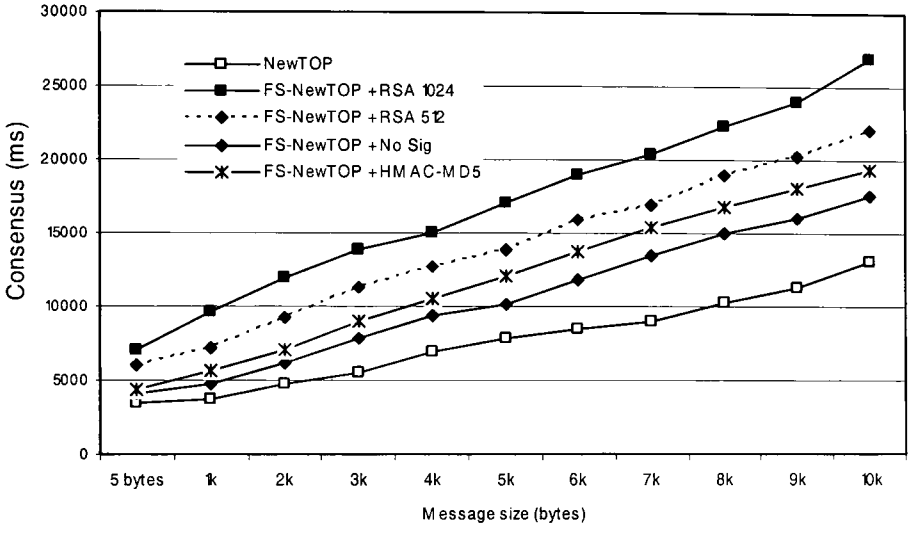

Figure 5.7: Membership protocol overhead when one member suddenly fails (varying size of messages, fixed group membership to 10 members)

The results are obtained by crashing one member (or crashing one replica of a member for FS versions) then measuring the time from the crash until the time all group members have agreed upon a new group view. Message sizes vary from 5 bytes up to 10k.

5.5.7 Cost of consensus when one member fails (varying timeouts).

The aim of this experiment was to measure the group consensus of the five systems when liveliness timeout was varied. The timeout was adjusted in NewTOP only because all FS-NewTOP disregards timeouts because there do no employ liveliness mechanisms.

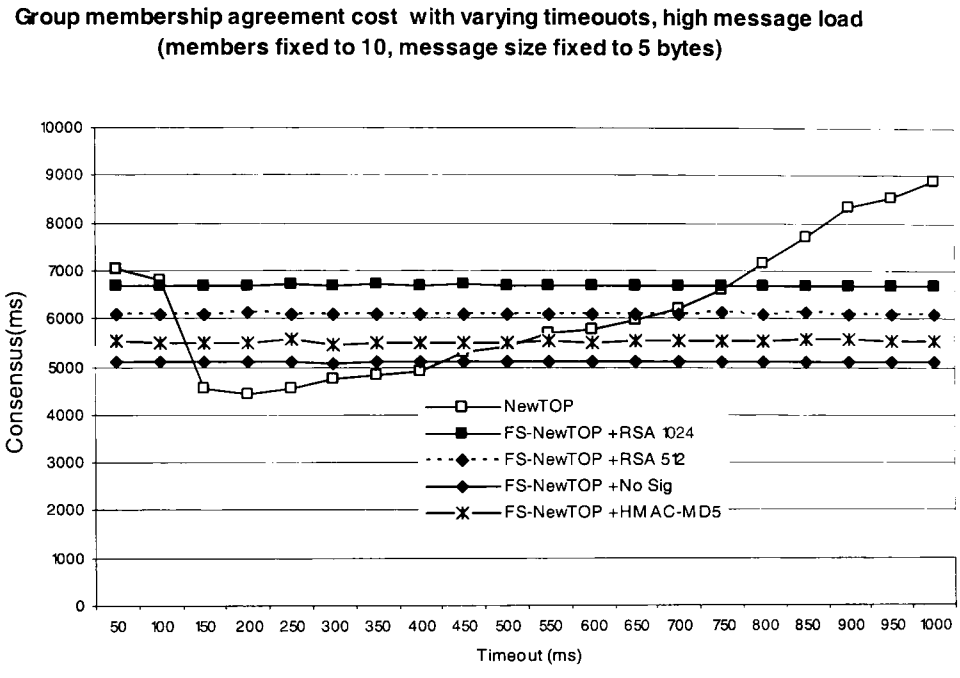

Figure 5.8: Membership protocol overhead when one member suddenly fails (varying liveliness timeouts)

At timeouts that are lower that the average NewTOP roundtrip (around 170ms) the NewTOP protocol does not perform well as it has to deal with incorrect suspicions. As the timeout nears the roundtrip time, NewTOP performs best. But as the timeout increases beyond the average roundtrip time, NewTOP performance goes down because it has to wait a long time to detect failure. For all versions of FS-NewTOP in Figure 5.8, the group membership cost is constant because fail-signalling protocol does not make use of timeouts.

5.6 Discussion and conclusion

Performance results were collected for NewTOP group communication system and four configurations for FS-NewTOP system 1) with no signatures, 2) with RSA 1024 key and four, 3) RSA 512 key and 4) with HMAC-MD5 hash algorithm. Each FS-NewTOP group member is duplicated into two entities that exchange signed messages.

Results have further shown that FS-NewTOP responds consistently regardless of the time-outs imposed (Figure. 5.8). This is one of the main advantages of FS-NewTOP.

Chapter 6

DISCUSSION AND CONCLUSIONS

6.1 Overview

A Byzantine-fault tolerant group communication system has been constructed by extending a crash-tolerant group system called NewTOP. The extension involves replacing crash-prone middleware processes with fail-signal or FS processes resulting in FS-NewTOP. The idea of signal-on-failure is not new and it is one of the three failure-guarantees offered by the Fail-stop processes of [Schlichting83]. A Fail-stop process requires (at least) three (internal) replicas, while an FS process can be done with a replica pair. A significant benefit of our fail-signal based approach is that the FLP impossibility result derived for unannounced crashes ceases to apply and consequently the total ordering is guaranteed to terminate provided the asynchronous network does not suffer permanent partitions. The fail-signal overhead was measured through a series of experiments. The increase in ordering latency and the reduction in throughput were found to be relatively small for large groups.

6.2 Assumptions

Assumptions in Chapter 3 are listed again:

*Assumption **A1***: The nodes are assumed to be correct (i.e., non-faulty) when they are paired at start-up time. If one of these two nodes fails, the other is assumed to work correctly until the system detects this fault.

Assumption ***A2.*** The nodes are connected by a reliable, synchronous communication link (LAN) that delivers messages within a known bound δ.

Assumption ***A3:*** Suppose that both nodes are non-faulty and an input is submitted to both of them at the same time t for processing. Say, p (respectively p') processes that input and generates a result at time $t+\_\!\!\perp t$ (respectively at $t+\_\!\!\perp t'$). We assume that $\max\{\_\!\!\perp t, \_\!\!\perp t'\} \leq \kappa * \min\{\_\!\!\perp t, \_\!\!\perp t'\}$, for some known, positive number κ.

Assumption ***A4:*** Similarly, suppose that both nodes are non-faulty and p and p' schedule a *send_result()* operation at the same time s to forward their result to the other replica. Say, p (respectively p') completes the send operation at time $s+\angle s$ (respectively at $s+\angle s'$). We assume that $\max\{\_\!\!\perp s, \_\!\!\perp s'\} \leq \delta * \min\{\_\!\!\perp t, \_\!\!\perp t'\}$ for some known, positive number δ.

Assumption ***A5:*** A3 and A4 require that the maximum difference between the delays for processing and scheduling of middleware messages be bounded and known. Finally, a process of a correct node can sign the messages it sends and the signed message cannot be generated nor undetectably altered by a process in another node.

The assumptions made in the construction of FS processes have implications at the application and middleware levels. Assumptions A3 and A4 require that the maximum delay within which the replicas of an FS middleware process complete processing of an incoming message or scheduling an outgoing message, must be known and bounded. Otherwise, correct replicas might find each other untimely and start emitting fail-signals unnecessarily. When the load imposed by application level processing is unknown, realizing A3 and A4 will require that the

replicas be run with a high priority. We note here that an application process, such as A_i in Figure 4.3, can also be made into an FS A_i in the same way middleware processes were transformed. This will incur an additional FS overhead at the application level and subject replicas of A_i to assumptions A3 and A4. Alternatively, when A_i is Byzantine fault-prone (as in Figure 4.3), another application, say B, can be replicated on the host nodes of *FSO'*.

Fail-signal construction also assumes that the two nodes of an FS process are connected by a synchronous link (assumption A2) and that no more than one node becomes faulty (assumption A1). A2 can be realized, say, by keeping each pair of nodes geographically close and making use of the fast Ethernet technology. Realizing A1 in an intrusion prone environment requires sufficient diversity and intrusion detection measures, which will be the focus of our future work; in this paper, we have assumed that the causes of the faults can only be internal.

6.3 FS service and further work

Our middleware system requires a total of $4f+2$ nodes, with assumption A1 restricting the locations of faults. However, the traditional Byzantine-tolerant total-order protocols neither require A1 nor anything similar to it. On the other hand, they rely on protocol-specific, *liveness* conditions to prevail for termination. In its design philosophy, [Verissimo02] is similar to ours and assumes a synchronous WAN to avoid the FLP impossibility result. Much of the motivation for its design, expressed elegantly using the *Wormholes* metaphor [Verissimo03], also underpin our approach.

One cost aspect of this approach is that masking of f Byzantine faults at the application level requires at least $2f+1$ FS-processes (i.e. a total of 4f+2 processes), with each replica requiring access to a total-order service. Compare 3f + 1 needed to tolerate malicious faults [Schneider90]. Since we construct our total-order service using FS middleware processes and each FS process itself is a self-checking pair running on distinct nodes, $4f+2$ nodes are needed in our approach i.e., $(f+1)$ nodes more than the known optimal requirement for a Byzantine-tolerant middleware system. We believe that this cost will be small, given the falling hardware price.

There are limitations of this work: the fail signalling service was constructed from an object oriented programming java and CORBA, which have proved to be less efficient in terms of speed. Furthermore, there has not been stringent Byzantine fault injection into the fail signalling service to see how it behaves. Although a simulation of crashing one member of the group was carried out to see how FS service behaves when one member suddenly becomes corrupt.

The other benefit of two fail signalling protocol is to tolerate intrusion. In an intrusion-prone environment, the adversary may attempt to block, delay, or corrupt messages on the network without signalling a security alert. Therefore, the message transmission delays between correct middleware processes cannot be bounded priori with certainty, unless the network is a trusted private channel. In our design, we have assumed that when correct middleware processes send messages to each other there is no known bound on delays to be experienced. A sent message will eventually be received, and a received message is the one that has been sent. In other words, the classical asynchronous delay model is assumed for the interaction between middleware processes of the failing replicas.

Further work can be carried out on this service. One of which can expose the service to real life Byzantine faults to see how it can sustain them.

Appendix A

PORTING AND ADAPTING VOLTAN TO FAIL SIGNALLING JAVA/CORBA SERVICE

A.1 Introduction

We describe how Voltan, a fail silent service, was ported from Java to CORBA Java and this appendix describes how the porting was done. The Voltan software library for building distributed applications provides the support for (i) a process pair to act as a single Voltan self-checking `fail-silent' process; and (ii) connection management for Voltan process communication. A Voltan fail-silent process is written by the application developer as a single threaded program. The Voltan system replicates this program transparently. The active replication of applications engenders problems when dealing with non-deterministic calculations [Brasileiro96].

A.2 Voltan Structure

The following diagram shows a more microscopic relationship between two replicas. In the diagram the replicated processes are referred to as application threads. Voltan replicas mainly use internal queues and threads to facilitate replica-to-replica message exchange. Only double-signed messages are considered authentic. "Double" because each sending replica must sign an outgoing message to the other recipient pair of replicas. A sequence number uniquely identifies a message. The leader replica dictates to the follower the order of how messages are received and consumed by leader and follower applications.

This dictatorship ensures that the computation of messages is deterministic between the replicas.

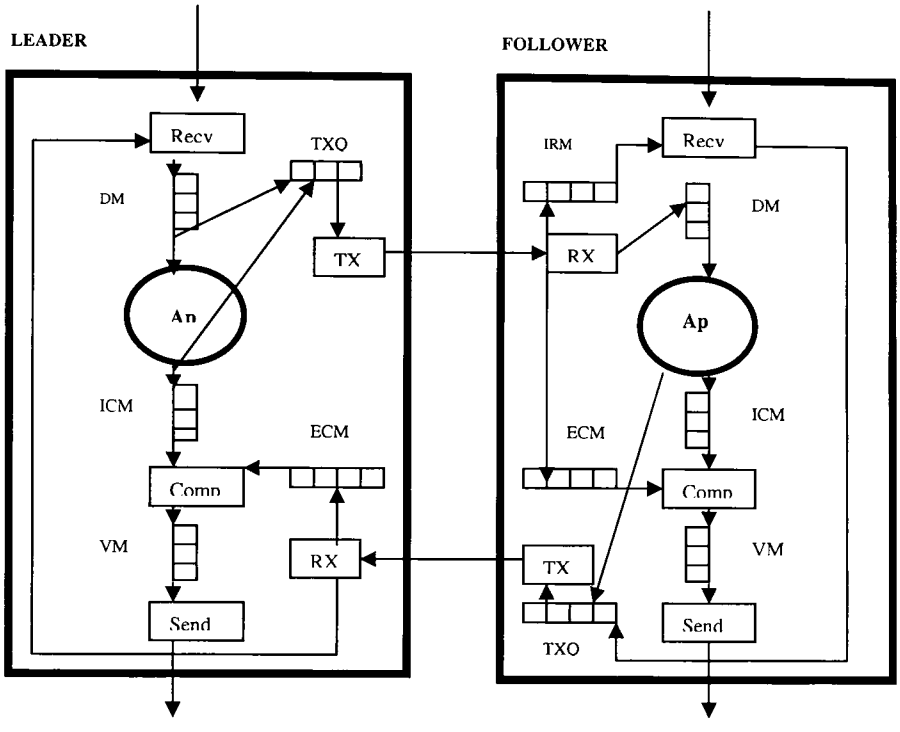

Figure A.1: Voltan fail silent process structure

A3 Similarities between the leader and the follower

Starting with what is common on the leader and the follower sides, the new double-signed message from another Voltan node is queued in the DM queue. The application stripes off the header information, including signatures, then processes the message. The newly computed message is then fleshly single signed and queued in the ICM queue to be later compared with the single-signed message from the partner (follower or leader) queued in the ECM queue. The comparator thread compares the two equivalent messages. If there are any discrepancies between the two, the leader/follower fails "silently and emits a signal" and immediately.

Otherwise if the messages compare, the message from the other partner is signed the second time and stored in the VM queue, the double-signed correct and authentic messages queue. Finally, the send thread pops the correct message from the VM queue sends it to the network destined for the recipient Voltan process.

Both the leader and the follower application forward their newly single signed messages to each other via the TXQ queue. The recipient RX thread checks the type of the incoming message so it can queue it into appropriate queue.

A.4 Differences between the leader and the follower

The difference between the leader and the follower stems from the fact that the leader has to dictate the order in which messages are to be consumed by both the leader and the follower application threads. A new double-signed message in the leader DM queue is forwarded to the follower DM via the leader TXQ queue. The leader TX thread then forwards the message to the follower via the follower RX thread. The same message is queued in the follower IRM queue for timeouts and leader-missed messages, as you will see how that is done below.

The other difference is the follower Receive thread does not queue the message in its follower DM queue because the leader has forwarded the messages to the follower DM queue. Instead when the follower receives a new double-signed message from a network, it checks if the message is in its IRM queue. If the message is there, it means the leader has received it and forwarded it there. In addition, the follower checks the time stamp of the message in the IRM queue and if it does not satisfy the allowable time delay, the follower declares a failure on the bases of a time-out. Each message has a time stamp field in its control header in case the message is not in the IRM

queue, it is possible that the leader missed the message therefore it is forwarded to the leader after some time delay of *2d* via the follower TXQ queue. Where *d* is the maximum one trip delay between the nodes. When *d* is *0* it means the follower forwards the messages to the leader immediately.

A.5 Limitations of this Voltan structure

In the old Voltan structure the communication between the leader and follower is established. Both in the C++ and Java versions of Voltan, the communication between the any two entities is mainly RPC and Sockets. There are three main communication points that require the use of RPC and Sockets: These are 1), between the TX and RX threads. 2), between the client Voltan process and the leader and the follower Receive threads, and finally between the Send thread and the recipient Voltan process. This exposes the programmer to low level network programming structure does only lead to a complex implementation of Voltan but also inflexibility associated with RPC and Sockets.

Other undesirable features of this structure that may hinder performance are threads and queues. They are about five threads in each replica: Receive, TX, Comparator, RX, Send, and Application. In addition we have five queues: DM, TXQ, ICM, ECM, and VM. The follower has IRM as an additional queue. Each of these queues is shared by at least two threads. The competing threads use synchronization for mutually exclusive access to the shared queues. This is performance costly since there is thread context switching between any two accesses to any shared queue by two competing threads.

Furthermore, the pure C++/Java implementation does not give much flexibility in deploying Voltan in today's distributed object oriented middleware systems that are dominantly based on CORBA.

A.6 CORBA Voltan model

The following section illustrates how porting Voltan to CORBA curbed the just mentioned limitations of the C++/Java version of Voltan. Initially we ported Voltan to CORBA exactly as it is in terms of the queues and threads. In the second version CORBA Voltan, we eliminated all features that were rendered redundant by the use of object based method calls. The performance of the latter was approximately 10x faster than the former version of Voltan laden with queues and threads. The performance tests were carried out with the pair replicas running on two different workstations on the same LAN.

A6.1 Introduction

CORBA is not a programming language but a distributed object environment that facilitates cross-language and cross-platform object-client/server communication. The basic units of computation are viewed as objects as opposed to processes, but at the same time CORBA object are rapped with language specific processes or threads. These processes and threads are made transparent to programmers. The interaction between the objects is facilitated by method calls within object references. Because CORBA is language neutral and is not itself a programming language, our design of CORBA Voltan may be implemented with any language of choice such as C++ or Java.

A.6.2 Object references and method calls

In CORBA, the caller (client) obtains the object reference of the server object. The client then executes the method within the server object using the dot operation as if the remote server object is in the same address space as the client. CORBA indeed hides the complexities of the underlying

networking operations such as addressing, location resolutions, marshalling and de-marshalling of parameters, and serializations. All these are not transparent in the pure C++ and pure Java Voltan. The following is an example code that demonstrates remote object calls:

```
remoteObjectRef.method();
```

The client object calls the above statement to obtain results computed by a remote object using the `method()`. The caller of the method has obtained the object reference and that is not shown here. In the same way, the following is a fragment of code from the new CORBA Voltan code that demonstrates remote object calls. The leader calls the methods in the remote follower object this way:

```
FOLLOWER.receiveMsgToCompareFromLeader(msg);
```

The above statement is the key to how CORBA Voltan has been implemented. The LEADER object, that has previously obtained the object reference of the FOLLOWER, calls the above statement. The leader is forwarding a single signed message, msg, to the follower for comparison. The FOLLOWER object does not have to be in the same workstation as the LEADER object, and yet the leader calls the method as if it (follower object) is collocated with the leader (same address space). In addition, the leader object does not have to be concerned about the whereabouts of the follower. In other words, the leader does not go through low-level network operations since the CORBA core engine, known ORB, handles all that transparently. We can think of an ORB as a language specific implementation of the specifications of CORBA.

The follower code is responsible for implementing this called method. The following code demonstrates how straightforward and comprehensible the

implementation of the above method is on the follower side. There is literally one statement required.

```
public void receiveMsgToCompareFromLeader(MessageBlock msg)
{
  // check authenticity of message
  // then push into the ECMQ
  F_ECMQ.push(msg);
}
```

If you look at Figure 3 below, the above method is exactly identical to `ReceiveSingle()`. "Single" means single-signed message, "Double" means double signed message. The follower needs to push the message into its own ECM queue for later comparison to the message from the its (follower) own ICM queue. The preceding F_ before ECMQ denotes the follower.

A.6.3 Oneway asynchronous "fire and forget" calls

CORBA allows the caller to make a call but not wait for the call to return by declaring the functions oneway in an .idl declaration file. (See Section A.9 below for details of .idl and CORBA Voltan implementation). Almost all the calls between the leader and the follower are implemented this way. It is efficient because it uses "fire and forget" technique. The success of the call is not certain to the caller because the caller continues with the next instruction immediately. For example, the follower method called by the leader in Section A.6.2 above is one-way because the leader does not have to witness the pushing of the message into the follower queue. In stead, the leader continues with other operations immediately after "firing" the call. Fortunately, it is the very reason that we are using fail-signalling to handle any such link failures in case the one-way method does not succeed.

A.6.4 Callback methods

Callback methods allow the object to behave both like the client and the server at the same time. In fact, in CORBA world, any object can be both a client and a server at the same time provided it initiates calls and receives calls respectively. When a client object gives the server its own reference such that the server can call methods in the client we say the server is making a callback to the client. This call pattern exactly suites our Voltan leader and follower requirements. Both the leader and the follower are remote CORBA objects and they know each other's object references. This means that they can call each other's methods at any time of their lifetimes as they please.

Oneway calls and callbacks make a perfect combination in this situation because the leader and follower cannot block in a deadlock or waste time while waiting for each other's methods to return. Instead, they both fire! and forget about the call. CORBA takes care of the rest.

A.6.5 CORBA Voltan structure

The figure above sums up the relationship between the leader and the follower in the new CORBA Voltan. The original structure in Figure A.1 shrank to the structure in Figure A.2. The new structure has much less queues and threads. This is mainly because object method calls are used. As earlier mentioned in Section 3.4, both the leader and follower obtain each other's object references at the initialization stage for subsequent calls to the each other. The leader, follower, applications are all CORBA objects. Remember that in the original Voltan, and application was a thread. The details of how methods within the application object are called, to consume messages from the DM queues, are not shown. This is so because it depends how the application consumes the messages.

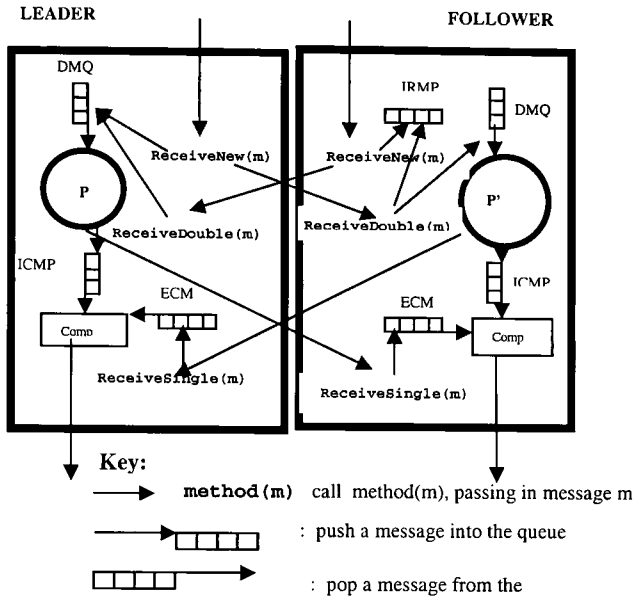

Figure A.2: CORBA Voltan fail-signalling structure

FSO is the leader and FSO' is the follower. If an input is from another FS process, it is checked for authentic, double signature; this is implemented in the `receiveNew(m)` method which, at the leader FSO, places the received input into the local Delivered Message Queue (DMQ) and then sends a copy to the follower by calling the follower's `receiveDouble(m)` method.

When the follower receives a message from the leader, it places it into the local DMQ; a copy of the message is also deposited in the Internal Received Message (IRMP) Pool. When the follower receives a valid message via `receiveNew()` method, it performs a different task to that executed by the leader. As it gets the message, it checks if the message is in the IRM Pool and if so, the pair is deleted. Otherwise the follower stores it in the IRM Pool with an associated timeout t1. If the message is not received from the leader within t1, the follower dispatches the message to the leader by calling the receiveDouble()of the leader. The message within IRM Pool is given a new timeout t2. If this second timeout expires and the message has not been

received from the leader, the follower assumes the leader has failed and starts emitting fail-signal to appropriate destinations. In the implementation the t_1 is set to 0, and t_2 is set to 2_d.

The target thread selects a message from DMQ, processes the message, and may produce an output message. A copy of an output message is signed once and transmitted to the other target replica by calling the `receiveSingle`(m) method of the latter. The unsigned message is stored in the Internal Candidate Message Pool (ICMP), setting a timeout. When a single signed message is received, it is placed in the External Candidate Message Pool (ECMP). The Compare thread compares relevant messages in ICMP and ECMP. If the comparison indicates that both messages contain identical result, then the comparison is deemed successful, the message from the ECMP is signed again, and the doubly signed message is sent to the destination(s). If the comparison fails or if an ICMP entry times-out, the Compare thread starts emitting fail-signals to appropriate destinations.

A3.6 Deprecated and modified features

The following are threads and queues from the original Voltan that are now deprecated in the new CORBA Voltan. The features phased out apply to both the leader and the follower. (Compare Figure A.1 and Figure A.2)

`Receive` (*thread*)

The receive-from-network thread has been removed. Because the sender of a message now has to simply call the `receiveNew`(m) method to send the message.

`TX` (*thread*)

The transmit-to-partner thread has been phased out. It has been replaced witj calling the "receive" methods in each replica object directly.

TXQ (*queue*)

The queue for the TX thread has also been removed.

RX (*thread*)

The receive-from-partner thread has been phased out too. Similarly to the TX thread, the RX threads have been replaced by the called methods pushing messages to the local queues. See example code at the end of Section A.6.2.

Send (*thread*)

The send thread has been removed because the newly compared message to is sent back to the other Voltan node by the comparator calling the equivalent receiveNew(m) on that recipient object.

VM (*queue*)

Likewise, the VM queue is no more needed.

A7. CORBA Voltan Implementation

This section gives a more detailed method-by-method description of the new CORBA Voltan. There are slightly more methods mentioned in the document than those shown in Figure 3 above. Further, the names of the methods in Figure 3 may be slightly different from the ones presented here, however they are exactly equivalent. For example, the following methods

from figure A.2 (short ones to save space) and this text (longer ones) are equivalent. Single denotes single signed message while double denote double signed message:

```
receiveNew(m) is equivalent to:

        receiveExternalMsg(m)

receiveDouble(m) is equivalent to:

        receiveExternalMsgFromFollower(m)
        receiveExternalMsgFromLeader(m)

receiveSingle(m) is equivalent to:

        receiveMsgToCompareFromFollower(m)
        receiveMsgToCompareFromLeader(m)
```

A.7.1 Introduction

Like earlier mentioned, CORBA is a language neutral object oriented environment that allows programmers to specify attributes and behavior of objects. The key CORBA structure that allows such specification is called an .idl file. Although the syntax of the file looks very similar to C++ or Java syntax, it has got nothing to do with these languages. The compiler of the .idl file is the one that dictates the programming language that is to be used to implement the objects. The details of idl compilation results and specific language mappings are beyond the scope of this document. The reader is referred to the Bibliography at the end of this document.

Most of the methods that are in the leader object are called by the follower object and *vice versa.*

We only show the interface of the leader and the follower. For a full list of the Voltan .idl file, see Appendix A. For the source code of the whole CORBA Voltan, contact the author.

A.7.2 Leader idl specification and implementation

```
interface  LeaderReplica
{
 void receiveConnectionFromFollower(in string objRef);
 oneway void receiveExternalMsg(in MessageBlock msg);
 oneway void receiveExternalMsgFromFollower(in MessageBlock msg);
 oneway void receiveMsgToCompareFromFollower(in MessageBlock
msg);
 oneway void receiveHeartBeatFromFollower();
 oneway void failLeaderSilently();
}; // LeaderReplica
```

A.7.2.1 void receiveConnectionFromFollower(objRef)

The follower calls this method at the initialization stage. It is assumed that the leader is already registered with the CORBA ORB and is waiting indefinitely for an "invitation" from the follower for partnership. To obtain the reference of the leader object, the follower uses one of the several conventional methods to obtain a reference of an object that is already registered. The follower must pass in its own object reference when calling this method. The leader then obtains the reference of the follower and stores it in the local private variable called say FOLLOWER. The beauty of this is the leader will never have to ask who the follower is or where it is for all subsequent method calls and *vice versa*. The leader will always use the FOLLOWER.method() syntax for all future calls to the follower. The leader has to immediately inform the follower that it (leader) has accepted the invitation as the partner by calling the `receiveConnectionAckFromLeader(in long status)` of the follower (See Section A.2.3.1)

Implementation:

```
public void receiveConnectionFromFollower(String objRef)
{
 // obtain the follower reference for good

  this.FOLLOWER=
  FollowerReplicaHelper.narrow(orb_.string_to_object(objRe));
```

```
  FOLLOWER.receiveConnectionAckFromLeader(vGlobals.CONNECTED);

  // ready the comparator thread.

  Thread L_Comparator =  new Comparator(
                      L_ICMQ, L_ECMQ);
}
```

A.7.2.2 oneway void receiveExternalMsg(msg)

The previous method does not have the preceding oneway. That is because the follower must wait for the connection method to return because it is still negotiating with the leader. In contrast, this method has one-way because the leader caller does not have to wait for the execution of the method on the follower side. This is more efficient and ensures asynchronous communication between the caller and the called.

This method receives double-signed messages from the network then queues them in the DM queue to be later consumed by the application.

Implementation:

```
public void receiveExternalMsg(MessageBlock msg)
{
  if (checkAuthenticity(msg)
  {
    L_DMQ.push(msg);
  }
}
```

A.7.2.3 oneway void receiveExternalMsgFromFollower(msg)

The follower forwards a double-signed message to the leader, by calling this method, in case the leader has missed the message. First, the message is checked if there is any matching message in the leader DM queue. If the message is already in the leader DM queue, the message is discarded. That is why we use a slightly different method for pushing the message.

Implementation:

```
public void receiveExternalMsgFromFollower(MessageBlock msg)
{
  if (checkAuthenticity(msg)
  {

   L_DMQ.matchOrPush(msg);
  }
  else
  {
    FOLLOWER.killFollowerSilently();
    this.killFollowerSilently ();
  }
}
```

A.7.2.4 oneway void receiveMsgToCompareFromFollower(msg)

This method is called by the follower application to forward a fleshly single-signed message to leader to be later compared by the leader comparator.

Implementation:

```
public void receiveMsgToCompareFromFollower(MessageBlock msg)
{ //like above, check authenticity of msg then act
  // accordingly
  L_ECMQ.push(msg);
}
```

A.7.2.5 oneway void receiveHeartBeatFromFollower()

In case there is no computation activity between the leader and the follower, the follower calls this method to remind the leader that it (follower) is still alive.

Implimentation:

```
public void receiveHeartBeatFromFollower()
{
 this.heartBeat = System.currentTimeMillis();
}
```

A.7.2.6 oneway void failLeaderSilently()

This method is called whenever there is any discrepancy in the value or timeout failure has occurred. It kills the current partner when called. Both the leader and follower can call this method. Before the object is killed, an application specific feedback has to be given either in the form of an exception or an appropriate call to the system that is relying on Voltan for fail-silence.

Implimentation:

```
public void failLeaderSilently()
{
  System.out.println("FEEDBACK DEPENDS ON APPLICATION");

  System.out.println("DEACTIVATE CORBA OBJECTS");

  System.exit(-1);
}
```

A.7.3 Follower idl specification and implementation

```
  interface FollowerReplica
  {

   oneway void receiveConnectionAckFromLeader(in long status);
   oneway void receiveExternalMsg(in MessageBlock msg);
   oneway void receiveExternalMsgFromLeader(in MessageBlock msg);
   oneway void receiveMsgToCompareFromLeader(in MessageBlock
msg);
   oneway void receiveHeartBeatFromLeader();
   oneway void failFollowerSilently();

  }; // FollowerReplica
```

A.7.3.1 oneway void receiveConnectionAckFromLeader(status)

The leader calls this method at initialization stage inside the leader `receiveConnectionFromFollower()`. This method is **oneway** because the leader does not have to wait for the method to return.

Implementation:

```
public void receiveConnectionAckFromLeader(int status)
{
   if (status == vGlobals.CONNECTED)
   {
     Thread F_Comparator =  new F_Comparator(
     F_ICMQ, F_ECMQ)
   }
   else
   {
     LEADER.failLeaderSilently();
     this.failFollowerSilently();
   }
}
```

A.7.3.2 oneway void receiveExternalMsg(msg)

This method receives double-signed messages from the network then forwards it to the follower. The follower is not supposed to queue this message into its own DM queue because the leader is the one that dictates order of messages in both the leader and the follower DM queues. See Figures A.1 and A.2 for the equivalent `receiveNew()` method on the follower side.

The method first checks if the message is in its IRM queue. See Section A.6.5 above

Implementation:

```
public void receiveExternalMsg(MessageBlock msg)
{
 F_IRMQ.insert(msg,timeout);
 LEADER.receiveExternalMsgFromFollower(msg);
}
```

A.7.3.3 oneway void receiveExternalMsgFromLeader(msg)

This method is called by the leader to forward a double-signed message to the follower for queuing in the follower DMQ (See Figure 2). In addition,

the message is pushed into the follower IRM queue. See Section 2.3 for details of how the IRM queue works.

Implementation:

```
public void receiveExternalMsgFromLeader(MessageBlock msg)
{
  F_DMQ.push(msg);
  F_IRMQ.push(msg);
}
```

A.7.3.4 oneway void receiveMsgToCompareFromLeader(msg)

This method is called by the leader application to forward a fleshly single-signed message to the follower ECM queue to be later compared to the equivalent follower single-signed message from the follower ICM queue.

Implementation:

```
public void
receiveMsgToCompareFromFollower(MessageBlock msg)
{
     F_ECMQ.push(msg);
}
```

A.7.3.5 oneway void receiveHeartBeatFromLeader()

In case there is no computation activity between the leader and the follower, the leader calls this method to remind the follower that it (leader) is still alive.

Implimentation:

```
public void receiveHeartBeatFromLeader()
{
  this.heartBeat = System.currentTimeMillis();
}
```

A.7.3.6 oneway void failFollowerSilently()

This method is called whenever there is a discrepancy in value or timeout failures. It kills the current partner when called. Both the leader and follower can call this method. Before the object is killed, an application specific feedback has to be given either in the form of an exception or an appropriate method call to the system that is relying on Voltan.

Implimentation:

```
public void failFollowerSilently()
{
  System.out.println("ERROR MESSAGE TO APPLICATION");
  System.out.println("DEACTIVATE CORBA OBJECTS");
  System.exit(-1);
}
```

A.8 Concluding remarks

The new CORBA Voltan, based on method calls, has proved to be a viable alternative to a pure C++ or Java Voltan initially based on threads and queues. The combination of oneway calls and callback methods has made the implementation of the new CORBA Voltan simple and efficient. The cost or overhead of the CORBA Voltan pairs is only approximately 7 milliseconds per message. That is if a computation takes t time to compute a message if the application un-replicated, it will take the same application $t + 7$ milliseconds in Voltan/replicated version. The tests were carried out with each replica of the pair running on two different Windows 2000 Pentium workstations on the same LAN. Moreover, the CORBA Voltan opens a window for us into the many CORBA based systems, such as NewTOP, that assumes fail crash for their computational components. Further more, making distributed systems components fail-signal should alleviate the systems of their failure detection mechanisms. By so doing, the distributed system protocol latency "footprint" or overhead can be profoundly reduced.

A.9 Fail signalling IDL

```
// voltan CORBA/Java
//
// (c) 03/03/02 16:10:15  (usa date)
//
// University of Newcastle upon Tyne
// Computing Science Department
//
// module: Voltan.idl
// author: Dimane Mpoeleng

module Voltan
{


  struct Control
  {
    string sourceObjectName;
    string destinationObjectName;
    string targetObjectName;
    unsigned long long sequenceNumber;
    unsigned long long timestamp;
    long messageChecksum;
    long processorSignature1;
    long processorSignature2;
    long messageLength;
    long messageType;
    long numberOfSignatures;
  };

  struct MessageBlock
  {
    Control msgControl;
    string message;
  };

  interface Replica
  {
    string name;
  };


  interface FollowerReplica;


  typedef sequence<Replica> replicaList;


  interface  LeaderReplica : Replica
  {

   // Receive msgs from follower and network

   void receiveConnectionFromFollower(in string objRef);

   oneway void receiveExternalMsg(in MessageBlock msg);
   oneway void receiveExternalMsgFromFollower(in MessageBlock msg);
   oneway void receiveMsgToCompareFromFollower(in MessageBlock msg);
   oneway void receiveHeartBeatFromFollower();

   // Fail Leader Silently

   oneway void failLeaderSilently();

  }; // LeaderReplica
```

```
  interface FollowerReplica : Replica
  {

   // Receive msgs from leader and network

   oneway void receiveConnectionAckFromLeader(in long status);

   oneway void receiveExternalMsg(in MessageBlock msg);
   oneway void receiveExternalMsgFromLeader(in MessageBlock msg);
   oneway void receiveMsgToCompareFromLeader(in MessageBlock msg);
   oneway void receiveHeartBeatFromLeader();

   // Fail Leader Silently.

   oneway void failFollowerSilently();

  }; // FollowerReplica


  interface ReplicaFactory
  {
   Replica createReplicaObject(in string objectRef, in string role);
  }

}; // Voltan module

#endif
```

References

[Amir95] Y. Amir, "Replication using Group Communication over a Partitioned Network." *PhD thesis, Institute of Computer Science*, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[Baldoni00] R. Baldoni, J.-M. Helary, and M. Raynal. "From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach". *In Proceedings of the International Conference on Dependable Systems and Networks*, pp. 273-282, New York, NY USA, June, 2000.

[Barrett90] P.A. Barrett, P.G. Bond, A.M. Hilborne, L. Rodrigues, D.T. Seaton, N.A. Speirs and P. Verissimo, "The Delta-4 Extra Performance Architecture (XPA)", *Digest of Papers*, FTCS-20, Newcastle upon Tyne, pp.481-488, June 1990.

[Bazzi01] R. A. Bazzi. "Access cost for asynchronous Byzantine quorum systems." *Distributed Computing Journal volume 14, Issue 1*, pp. 41–48, January 2001.

[Beauquier97] J Beauquier and S Kekkonen-Moneta. "Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors." *International Journal of System Science*, 28(11), pp.1177–1187, 1997.

[Bellare93] M. Bellare, and P. Rogaway. "Entity authentication and key distribution." *In Crypto 93*, USA, IEEE Computer Society Press, pp. 232–249, 1993.

[Bellare96] M. Bellare, R. Canettiy and H. Krawczykz, "Keying Hash Functions for Message Authentication", *Advances in Cryptology, Crypto 96 Proceedings*, Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.

[Birman93] K. P. Birman. "The Process Group Approach To Reliable Distributed Computing", *Communications of the ACM*, 36(12), pp. 37 - 52, 1993.

[Birman94] K. P. Birman, and R.V. Renesse. "Reliable Distributed Computing with the Isis Toolkit." *IEEE Computer Society Press*, USA, 1994.

[Birman96] K. P. Birman, *Building Secure and Reliable Network Applications*, Manning, 1996

[Birman99] K. P. Birman. "A review of experiences with reliable multicast." *Software, Practice and Experience*, 29(9) p741–774, Sept 1999.

[Black98] D. Black, C. Low and S. K. Shrivastava, "The Voltan Application Programming Environment for Fail--Silent Processes", *Distributed Systems Engineering*, vol. 5,pp. 66-77, June 1998.

[Bracha95] G. Bracha and S. Toueg. "Asynchronous Consensus and Broadcast Protocols." *Journal of the ACM*, 32(4), 1995.

[Brasileiro96] F. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. Speirs and S. Tao, "Implementing fail-silent nodes for distributed systems", *IEEE Transactions on Computers*, 45(11), pp. 1226-1238, November 1996.

[Castro99a] M. Castro and B. Liskov. "A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm". *Technical Memo MIT/LCS/TM-590*, MIT Laboratory for Computer Science, 1999.

[Castro99b] M. Castro and B. Liskov. "Authenticated Byzantine fault tolerance without public-key cryptography." *Technical Report /LCS/TM-595*, MIT, 1999.

[Castro99c] M. Castro and N. Liskov. "Practical Byzantine fault tolerance." In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), New Orleans, USA*, pp. 173–186, February 1999.

[Castro00] M. Castro and B. Liskov. "Proactive recovery in a Byzantine fault-tolerant system." In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00), San Diego, USA*, pp. 273–287, October 2000.

[Chandra91] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Asynchronous Systems", *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 325-340, ACM, August 1991.

[Chandra96] T. D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, 43(2), pp. 225 - 267, March 1996.

[Chandra98] S. Chandra and P. Chen, "How fail-stop are faulty programs?", Proc. of Fault tolerant Computing Symp., *FTCS-28*, June 1998, pp. 240-249.

[Chandy85] K. M. Chandy and L. Lamport. "Distributed snapshots: determining global states of distributed systems." *ACM Transations on Computing Systems*, 3(1) pp.63–75, 1985.

[Charron-Bost00] B. Charron-Bost, Rachid Guerraoui, and Andr´e Schiper. "Synchronous system and perfect failure detector: Solvability and efficiency issues." *In International Conference on Dependable Systems and Networks (IEEE Computer Society)*, 2000.

[Chase98] C. M. Chase and K. G. Vijay. "Detection of global predicates: Techniques and their limitations." *Distributed Computing*, 11(4) pp.191–201, 1998.

[Cristian85] F. Cristian, H. Aghili, H. Strong, and D. Dolev. "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement." In *International Conference on Fault Tolerant Computing*, 1985.

[Cristian91a] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of the ACM*, 34(2), February 1991.

[Cristian91b] F. Cristian, "Reaching Agreement On Processor Group Membership In Synchronous Distributed Systems", *Distributed Computing*, 4(4), pp. 175 - 187, 1991.

[Cristian99] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model", In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10 (6), pp. 642-57, June 1999.

[Cukier98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J., "AquA: An Adaptive Architecture that provides Dependable Distributed Objects", *Proceedings 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, USA, October 1998, pp.245-253.

[Cukier01] M. Cukier, J. Lyons, P. Pandey, H. V. Ramasamy, W. H. Sanders, P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, M. Atighetchi, and J. Gossett, "Intrusion Tolerance Approaches in ITUA," FastAbstract in *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pp. B64-B65, 2001

[Deepak96] T. D. Chandra and S. Toueg. "Unreliable failure detectors for reliable distributed systems". *Journal of the ACM*, 43(2) pp.225–267, March 1996.

[DeMillo82] R. DeMillo, N. Lynch and M. Merritt. "Cryptographic Protocols" Proc. 14th ACM Symposium on the Theory of Computing, 1982.

[Dolev82] D. Dolev and H. R. Strong. "Polynomial Algorithms for Multiple Processor Agreement". *Proc. 14th ACM Symposium on the Theory of Computing*, pp.401-407, 1982.

[Dolev87] D. Dolev, C. Dwork, and L. Stockmeyer "On the Minimal Synchronization Needed For Distributed Consensus", *Journal of the ACM*, 34(1), pp. 77 - 97, January 1987.

[Dolev93] D. Dolev, S. Kramer, and D. Malki, "Early Delivery Totally Ordered Multicast in Asynchronous Environment", *Digest of Papers, FTCS-23*, Toulouse, p544-553, 1993.

[Dolev95] D. Dolev., D. Malki, and R. Strong. "A framework for partitionable membership service." *Technical Report 95-4*, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.

[Dolev00] S. Dolev. "Self-Stabilization." *MIT Press*, 2000.

[Dwork88] C. Dwork, N. Lynch, L. Stockmeyer, "Consensus in The Presence of Partial Synchrony", *Journal of the ACM*, 35(2), pp. 288 - 323, April 1988.

[Ezhilchelvan95] P. Ezhilchelvan, R. A. Macedo, S. K. Shrivastava "NewTOP: a fault-tolerant group communication protocol", *15th IEEE Intl. Conf. on Distributed Computing Systems*, Vancouver, pp. 296-306, May 1995.

[Ezhilchelvan01] P. D. Ezhilchelvan, A Mostefaoui, and M Raynal, "Randomized Multivalued Consensus", *In Proceedings of the Fourth*

International IEEE Symposium on Object oriented Real-time Computing (ISORC), Magdeburg, Germany, pp. 195-201, May 2001.

[Ezhilchelvan02] P D Ezhilchelvan, "A Middleware Architecture for Intrusion Tolerant Service Replication", *In Proceedings of the International Workshop on Intrusion Tolerant Systems*, pp. C61 – C67, In association with IEEE International Conference on Dependable Systems and Networks (DSN02), Washington, DC, June 23 - 26, 2002.

[Felber98a] P. Felber, R. Guerraoui and A. Schiper, "The implementation of a CORBA object group service", *Theory and Practice of Object Systems*, 4(2), 1998, pp. 93-105.

[Felber01] P. Felber and F. Pedone. "Probabilistic atomic broadcast". Technical report, Bell Labs, Lucent, Dec. 2001.

[Felix00] F. C. Gärtner and S. Kloppenburg. "Consistent detection of global predicates under a weak fault assumption". In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, pp. 94–103, N¨urnberg, Germany, October 2000. IEEE Computer Society Press.

[Fetzer96] Christof Fetzer , Flaviu Cristian, "Fail-awareness in timed asynchronous systems", *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing,* p.314-321, Philadelphia, Pennsylvania, United States, May 23-26, 1996.

[Fischer82] M. J. Fischer and N.A. Lynch. "A Lower Bound on the Time to Assure Interactive Consistency". *Information Processing Letters 14*, pp. 183-186, 1982.

[Fischer83] M. Fischer, "The Consensus Problem in Unreliable Distributed Systems", *Proceedings of the International Conference on Foundations of Computing Theory*, Sweden, 1983.

[Fischer85] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, April 1985.

[Fischer86] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1, pp. 26-39, 1986.

[Fuchs97] E. Fuchs, "Validating the Fail-Silence Assumption of the MARS Architecture", *Sixth IFIP Conf. on Dependable Computing for Critical Applications* (DCCA-6), Germany, March 1997.

[Garcia-Molina91] H. Garcia-Molina, "Ordered and Reliable Multicast Communication", *ACM Transactions on Computing Systems*, Vol. 9, No. 3, pp. 242-271, August 1991

[Goldreich00] O. Goldreich. "Secure multi-party computation", *Working draft, version 1.2*, March 2000.

[Gupta00] Gupta, I., Renesse, R. V., and Birman, K. P. "A probabilistically correct leader election protocol for large groups." *TR, Department of Computer Science*, University of Cornell, 2000.

[Hadzilzcos93] J. Y. Hadzilzcos and S. Toueg, "Fault-Tolerant Broadcasts and Related Problems", *ACM Press Frontier*, Addison-Wesley, Chapter 5, pp. 97-145, 2nd edition, 1993.

[Halpern84] J.Y. Halpern, B. Simons, H.R. Strong and D. Dolev, "Fault-tolerant Clock Synchronization", *Proc. Third ACM Symp.* on PODC, Vancouver, Canada, pp.89-102, August 1984.

[Krishnamurthy02] S. Krishnamurthy, "An Adaptive Quality of Service Aware Middleware for Replicated Services", *Ph.D. Thesis,* University of Illinois, 2002

[Krishnamurthy03] S. Krishnamurthy, W. H. Sanders, and M. Cukier. "An Adaptive Quality of Service Aware Middleware for Replicated Services", *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, pp. 1112-1125, November 2003.

[Kloppenburg00] S. Kloppenburg and F. C. Gärtner, "Consistent detection of global predicates in asynchronous systems with crash failures", *TUD-BS-2000-01,* Darmstadt, Germany, 2000.

[Lamport78] L. Lamport, "Time Clocks and The Ordering Of Events in a Distributed System", *Communications of The ACM*, 21(7), pp. 558 - 565, 1978.

[Lamport82] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, 4 (3), pp.382-401, 1982.

[Lamport83] L. Lamport. "The weak Byzantine generals problem." *Journal of the ACM*, 30(3) pp.668-676, July 1983.

[Li93] P. Li and B. McMillin. "Fault-tolerant distributed deadlock detection/resolution." In *Proceedings of the 17th Annual International Computer Software and Applications Conference (COMPSAC'93)*, pp. 224–230, November 1993.

[Lotem97] E. Y. Lotem, I. Keidar and D. Dolev. "Dynamic Voting for Consistent Primary Components", *In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 63-71, 1997.

[Malkhi97a] D. Malkhi and M. Reiter. "Byzantine Quorum Systems." In *ACM Symposium on Theory of Computing*, 1997.

[Malkhi98a] D. Malkhi and M. Reiter, "Byzantine Quorum Systems", *Distributed Computing*, 11(4), pp.203-213, 1998.

[Malkhi98b] D. Malkhi and M. Reiter. "Secure and scalable replication in Phalanx." *In Proc. 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana, USA*, Oct 1998.

[Malkhi00] D. Malkhi, M. Reiter, and A.Wool. "The load and availability of Byzantine quorum systems." *SIAM Journal on Computing 29(6)*, pp. 1889–1906, 2000.

[Matsui00] H. Matsui, M. Inoue, T. Masuzawa, and H. Fujiwara. "Fault-tolerant and self-stabilizing protocols using an unreliable failure detector." *IEICE Transactions*, E83-D(10) pp.1831–1840, October 2000.

[McDaniel99] McDaniel, P. D., Prakash, A., and Honeyman, P. "Antigone: A Flexible Framework for Secure Group Communication." *In Proceedings of the 8th USENIX Security Symposium*, Berkely USA, Usenix society, August 1999.

[Melliar-Smith91] P. M. Melliar-Smith, "Membership Algorithms for Asynchronous Distributed Systems", *Proc. Of 12th Intl. Conf. On Distributed Comp.* Systems, p 480-488, 1991.

[Melliar-Smith94] P. M. Mellier-Smith, L. E. Moser, Y. Amir, and D. A. Agarwal , "Extended Virtual Synchrony", *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, Poland, pp. 56 - 65, 1994.

[Mittra97] S. Mittra. "Iolus: A framework for scalable secure multicasting." *In SIGCOMM*, New York, USA, ACM press. September 1997.

[Morgan00a] G. Morgan and S. K. Shrivastava, "Implementing Flexible Object Group Invocation in Networked Systems", *in the Proceedings of the International Conference on Dependable Systems and Networks*, New York, June 2000.

[Morgan00b] G. Morgan and P. D. Ezhilchelvan, "Policies for using Replica Groups and their Effectiveness over the Internet", *The second International Workshop on Networked Group Communications*, Stanford, November, pp. 119-130, 2000.

[Moser94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal, "Extended virtual synchrony," in *Proceedings of the IEEE 14th International Conference on Distributed Computing Systems*, pp. 56–65, IEEE Computer Society Press, Los Alamitos, CA, June 1994.

[Moser96] L. E. Moser, P.M. Melliar-Smith D. Agarwal, R. Budhia and C.Lingley-Papdopoulous, "Totem: a Fault-tolerant multicast group communication system",*CACM*, 39 (4), April 1996, pp. 54-63.

[Nancy96] N. Lynch. "Distributed Algorithms." *Morgan Kaufmann Publishers*, 1996.

[Narasimhan99b] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA", *IEEE Computer,* pp. 62-68, July 1999.

[Narasimhan00] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "The Eternal System", In *Encyclopedia of Distributed Computing*, edited by P. Dasgupta and J. E. Urban, Kluwer Academic Publishers (2000).

[OMG00] OMG Technical Committee Document ptc/00-04-04, Object Management Group, March 2000.

[Pandey01] P. Pandey, "Reliable Delivery and Ordering Mechanisms for an Intrusion-Tolerant Group Communication System," *MS Thesis*, University of Illinois at Urbana-Champaign, 2001

[Pease80] M. Pease, R. Shostak, and L. Lamport. "Reaching agreement in the presence of faults." *Journal of the ACM*, 27(2) pp.228-234, Apr. 1980.

[Pedone02] Pedone A., Schiper A., Urban P, Cavin D, "Solving Agreement Problems with Weak Ordering Oracles", *Technical Report HPL-2002-44*, Hewlett-Packard Laboratories, March 2002.

[Ramasamy02] H. V. Ramasamy, "Group Membership Protocol for an Intrusion-Tolerant Group Communication System," *MS Thesis*, University of Illinois at Urbana-Champaign, 2002.

[Reiter94a] M. K. Reiter. "Secure agreement protocols: Reliable and atomic group multicast in rampart." In *ACM Conference on Computer and Communication Security*, pp. 68–80, New York, USA, November 1994.

[Reiter94b] M. K. Reiter, K. P. Birman, and R.V. Renesse, "A security architecture for fault-tolerant systems." *ACM Transactions on Computer Systems*, 16(3) pp.986–1009, November 1994.

[Reiter94c] M. Reiter. "Secure Agreement Protocols." In *ACM Conference on Computer and Communication Security*, 1994.

[Reiter94d] Michael K. Reiter, "A Secure Group Membership Protocol," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 176-189, 1994.

[Reiter95] M. Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems (LNCS 938)*, 1995.

[Renesse96] Renesse, R.V., Birman, K.P., and Maffeis, S. "Horus, a flexible group communication system." Communications of the ACM, 39(4) pp.76–83, April 1996.

[Schiper93] A. Schiper and A. Ricciardi, "Virtual Sunchronous Communication Based on a Weak Failure Suspector", *Digest of Papers*, FTCS-23, Toulouse, pp. 534-543, 1993.

[Schiper02] A. Schiper, "Failure Detection vs Group Membership in Fault-Tolerant Distributed Systems: Hidden Trade-Offs" *PAPM-PROBMIV* pp. 1-15 2002.

[Schlichting83] R. Schlichting and F. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", *ACM Transactions on Computer Systems*, Vol. 1(3), pp. 222-238, August 1983.

[Schneider84] F. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Transactions on Computer Systems*, Vol. 2(2), pp. 145-154, May 1984.

[Schneider90] F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM ComputingSurveys*, Vol. 22, No. 4, 1990.

[Schwarz94] R. Schwarz and F. Mattern. "Detecting causal relationships in distributed computations: in search of the holy grail." *Distributed Computing*, 7, pp. 149–174, 1994.

[Shah84] A. Shah and S. Toueg. "Distributed snapshots in spite of failures." *Technical Report TR84-624,* Cornell University, Computer Science Department, July 1984.

[Stott01] D. T. Stott, N. A. Speirs, Z. Kalbarczyk, S. Bagchi, J. Xu, and R.K. Iyer, "Comparing Fail Silence Provided by Duplication versus Internal Error Detection for DHCP Server", *IEEE 15th Int. Symposium on Parallel and Distributed Processing*, San Francisco, April 2001.

[Sullivan91] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability - a study of field failure in operating systems", Proc. of Fault tolerant Computing Symp., FTCS-21, June 1991, pp. 2-9.

[Ver´ıssimo00] P. Ver´ıssimo, A. Casimiro, and C. Fetzer. "The timely computing base: Timely actions in the presence of uncertain timeliness." In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 533–542, New York City, USA, IEEE Computer Society Press, June 2000.

[Ver´ıssimo00] P. Ver´ıssimo, N. F. Neves, and M. Correia, "The Middleware Architecture of MAFTIA: A Blueprint," *Technical Report DI/FCUL TR 00-6,* Department of Computer Science, University of Lisbon, 2000.

[Verissimo02] P. Verissimo and A. Casimiro, "The Timely Computing Base Model and Architecture", *IEEE Transaction on Computing Systems*, 51(8), pp. 916-930, 2002.

[Verissimo03] P. Verissimo, "Uncertainty and predictability: Can they be reconciled?", *Future Directions in Distributed Computing,* Springer-Verlag LNCS 2584, (To appear in) 2003.

[Vijay98a] K. G. Vijay and J. R. Mitchell. "Distributed predicate detection in a faulty environment." In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, 1998.

[Vijay98b] K. G. Vijay and J. R. Mitchell. "Implementable failure detectors in asynchronous systems." In *Proc. 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1530 in Lecture Notes in Computer Science, Springer-Verlag, Chennai, India, December 1998.

Bibliography

[Agarwal94] D. A. Agarwal, "Totem: A Reliable Ordered Delivery Protocol for Inter-connected Local Area Networks", *PhD Thesis, Department of Electrical and Computing Engineering,* University of California, Santa Barbara, 1994.

[Alpern85] B. Alpern and F. B. Schneider. "Defining liveness". *Information Processing Letters*, 21, pp. 181–185, 1985.

[Alvisi00] L. Alvisi, D. Malkhi, E. Pierce, and R. Wright. "Dynamic Byzantine quorum systems." In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.

[Amir00] Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Schlossnagle, J. Schultz, J. Stanton, and G. Tsudik, "Secure group communication in asynchronous networks with failures: Integration and experiments." *In International Conference on Distributed Computing Systems*, IEEE Computer Society Press, USA, April 2000.

[Amir92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. "Transis: A Communication Sub-System for High Availability." *In FTCS conference*, USA, IEEE Computer Society Press. http://www.cs.huji.ac.il/_transis, pp. 76–84, July 1992.

[Amir98] Y. Amir and J. Stanton, "The spread wide area group communication system," *Tech. Rep. 98-4*, Johns Hopkins University Department of Computer Science, 1998.

[Arlat90] J. M. Arlat, L. Aguera, Y. Amat, J.C. Crouzet, J.C. Fabre, E. Laprie, D. Martins, D. Powell, "Fault Injection for Dependability Validation: a Methodology and Some Applications", *IEEE Trans. on Soft. Eng.*, 16 (2), pp. 166-182, February 1990.

[Babaoglu94] O. Babaoglu, R. Davoli, L. Giachini, M. G. Baker, "Relacs: a communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems", *BROADCAST Project deliverable report* (available from Dept. of Computing Science, University of Newcastle upon Tyne, UK).

[Babauglu94] O. Babaoglu and A. Schiper, "On Group Communications in Large-Scale Distributed Systems", *Proceedings of the 6th ACM SIGOPS European Workshop*, pp. 17 - 22, Germany, 1994.

[Bazzi00] R. A. Bazzi. "Synchronous Byzantine quorum systems." *Distributed Computing Journal Volume 13, Issue 1*, pp. 45–52, January 2000.

[Bennett84] C. H. Bennett and G. Brassard. "An update on quantum cryptography." In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology: Proceedings of CRYPTO 84*, volume 196 of *Lecture Notes in Computer Science*, pp. 475-480. Springer-Verlag, 1985, 19-22 August 1984.

[Birman87] K. P. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *11th Annual Symposiumon Operating Systems Principles*, pp. 123–138, November 1987.

[Birman91] K. Birman, "Design Alternatives for Process Group Membership and Multicast", *Technical Report TR91-1257*, Department of Computing Science, Cornell University, December 1991.

[Burns87] J. E. Burns and N. A. Lynch. "The Byzantine firing squad problem." In F. P. Preparata, editor, *Advances inComputing Research, Parallel and Distributed Computing*, volume 4, JAI Press, Inc., Greenwich, Conn., pp. 147-161, 1987.

[Canneti92] R. Canneti and T. Rabin. "Optimal Asynchronous Byzantine Agreement". *Technical Report #92-15*, Computer Science Department, Hebrew University, 1992.

[Charron-Bost95] B. Charron-Bost, C. Delporte-Gallet, and Hugues Fauconnier. "Local and temporal predicates in distributed systems." *ACM Transactions on Programming Languages and Systems*, 17(1) pp.157–179, January 1995.

[Coan89] B. A. Coan, D. Dolev, C. Dwork, and L. Stockmeyer. "The distributed firing squad problem. *SIAM Journal on Computing*, 18(5) pp.990-1012, Oct. 1989.

[Cristian96a] F. Cristian, "Synchronous and Asynchronous Group Communications", *Communications of The ACM*, 39(4), pp. 88 - 97, 1996.

[Cristian96b] F. Cristian, B. Dancey, and J Dehn, "Fault-Tolerance in Air Traffic Control Systems", *ACM Transactions on Computer Systems*, Vol 14, No. 3, pp.265-286, August 1996.

[Diffie76] W. Diffie. And M. Hellman. "New directions in cryptography." *IEEE Transactions on information Theory*, IT-22:644–654, November 1976.

[Dolev83] D. Dolev and H. R. Strong. "Authenticated algorithms for Byzantine agreement." *SIAM Journal on Computing*, 12(4) pp.656-666, 1983.

[Dolev96] D. Dolev and D. Malkhi, "The Transis Approach to High Availability Cluster Communication", *Communications of the ACM*, Vol. 39, No. 4, pp. 64-74, April 1996.

[Ezhilchelvan01] P D Ezhilchelvan, A Mostefaoui, and M Raynal, "Randomized Multivalued Consensus", *In Proceedings of the Fourth International IEEE Symposium on Object oriented Real-time Computing (ISORC),* May 2001, Magdeburg, Germany, pp. 195-201.

[Ezhilchelvan86] P. Ezhilchelvan and S. K. Shrivastava, "A characterization of faults in systems." Proc. *of 5th IEEE Symp. on Reliability in Distributed Software and Database Systems*, Los Angeles, pp. 215-222, January 1986.

[Ezhilchelvan89] P. D. Ezhilchelvan, S. K. Shrivastava and A. Tully, "Constructing Replicated Systems Using Processors With Point to Point Communication Links", *The Proceedings of 16th Annual Symposium on Computer Architecture*, Jerusalem, pp.177-184, June 1989.

[Ezhilchelvan91] P. D. Ezhilchelvan and S. K. Shrivastava, "A Distributed Systems Architecture Supporting High Availability and Reliability', *Proc. of 2nd IFIP Conf. on Dependable Computing for Critical Applications*, Arizona, February 1991.

[Ezhilchelvan99] P. Ezhilchelvan and S. K. Shrivastava, "Enhancing Replica Management Services to Tolerate Group Failures", *Proceedings of the second International Symposium on Object oriented Real-time Computing (ISORC),* St Malo, France, May 1999.

[Fekete97] A. Fekete, N. Lynch, and A. Shvartsman, "Specifying and using a partionable group communication service," *In Proceedings of the 16th annual ACM Symposium on Principles of Distributed Computing*, (Santa Barbara, CA), pp. 53–62, August 1997.

[Felber98b] P. Felber, "The CORBA Object Group Service: a Service Approach to Object Groups in CORBA", *PhD thesis*, Ecole Polytechnique Federale de Lausanne, 1998.

[Feldman97] P. Feldman and S. Micali. "An optimal probabilistic protocol for synchronous Byzantine agreement." *SIAM Journal on Computing*, 26(4) pp.873-933, Aug. 1997.

[Fitzi01] M. Fitzi, N. Gisin, and U. Maurer. "Quantum solution to the Byzantine agreement problem." *Physical Review Letters*, 87(21), Nov. 2001.

[Fitzi02a] M. Fitzi, N. Gisin, U. Maurer, and O. von Rotz. "Unconditional Byzantine agreement and multi-party computation secure against dishonest minorities from scratch." In *Advances in Cryptology, EUROCRYPT 2002*, Lecture Notes in Computer Science, 2002.

[Fitzi02b] M. Fitzi, D. Gottesman, M. Hir, H. Holenstein, A. Smith, "Detectable Byzantine agreement secure against faulty majorities", ACM Press, Monterey, California, pp.118 – 126, 2002.

[Garay98] J. Garay and Y. Moses. "Fully Polynomial Byzantine Agreement for n 3t Processors in t+1 Rounds." *SIAM Journal of Computing*, 27(1), 1998.

[Goldwasser02] S. Goldwasser and Y. Lindell. "Secure computation without a broadcast channel." http://eprint.iacr.org/2002/040.ps, Apr. 2002.

[Gong95] L. Gong, P. Lincoln, and J. Rushby. "Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults," 1995.

[Gottesman01] D. Gottesman and I. Chuang. "Quantum digital signatures." Manuscript, 2001.

[Gutmann01] P. Gutmann, "Cryptlib Security Toolkit," April 2001

[Hardjono99] T. Hardjono, B. Cain, and N. Doraswamy. "A framework for group key management for multicast security." *Technical Report* draft-ietf-ipsecgkmframework-01.txt, IETF, Network security subworking Group, 1999.

[Harney97a] H. Harney, and C. Muckenhirn. "Group key management protocol architecture." *RFC 2094*, IETF, 1997.

[Harney97b] H. Harney and C. Muckenhirn. "Group key management protocol specification." *RFC 2093*, IETF, 1997.

[Hayden01] M. Hayden, "Ensemble Reference Manual," Cornell University, 2001

[Hiltunen00] M. A. Hiltunen, S. Jaiprakash, R. D. Schlichting, R. D., and C. A. Ugarte, C. A. "Finegrain configurability for secure communication." *Technical Report TR00-05*, Department of Computer Science, University of Arizona, June 2000.

[Hiltunen96] Hiltunen, M. A. and Schlichting, R. D. "Adaptive distributed and fault tolerant systems." *International Journal of Computer Systems Science and Engineering*, 11(5) pp.125–133, September 1996.

[Kihlstrom98] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," *Proceedings of the 31st IEEE Hawaii International Conference on System Sciences*, pp.317-326, 1998

[Kim00] Y. Kim, A. Perrig, and G. Tsudik. "Simple and fault-tolerant key agreement for dynamic collaborative groups." *In 7th ACM Conference on Computer and Communication Security*, New York, USA, November 2000. ACM press.

[Lala91] J. H. Lala, R.. E. Harper, and L. S. Alger, "A Design Approach for Ultrareliable Real Time Systems", *IEEE Computer*, pp.12-22, May 1991.

[Lamport77] L. Lamport. "Proving the correctness of multiprocess programs." *IEEE Transactions on Software Engineering*, 3(2) pp.125–143, March 1977.

[Lamport86] L. Lamport. "On interprocess communications." *Distributed Computing*, pp. 77–101, 1986.

[Lamport89] L. Lamport. "The Part-Time Parliament." *Technical Report 49*, DEC Systems Research Center, 1989.

[Lamport95] L. Lamport. "How to write a proof." *American Mathematical Monthly*, 102(7) pp.600–608, August/September 1995.

[Lindell02] Y. Lindell, A. Lysyanskaya, and T. Rabin. "On the composition of authenticated byzantine agreement." In *ACM Symposium on Theory of Computing (STOC '02)*, 2002.

[Little92] M. C. Little, "Object Replication in a Distributed System", *PhD Thesis*, University of Newcastle upon Tyne, 1992.

[Little99] M. C. Little, S. K. Shrivastava, "Implementing high availability CORBA applications with Java", *Proceedings of the IEEE Workshop on Internet Applications*, San Jose, California, June 1999.

[Macedo95] R. J. De A. Macedo "Fault-Tolerant Group Communication Protocols For Asynchronous Systems", *PhD Thesis*, Department of Computing Science, University of Newcastle upon Tyne, 1995.

[Malkhi96] D. Malkhi and M. Reiter. "A High-Throughput Secure Reliable Multicast Protocol". *In Computer Security Foundations Workshop*, 1996.

[Malkhi97b] D. Malkhi and M. Reiter. "Unreliable Intrusion Detection in Distributed Computations." In *Computer Security Foundations Workshop*, 1997.

[Matt01] B. Matt, B. Niebuhr, D. Sames, G. Tally, Brent Whitmore, and David Bakken, "Intrusion Tolerant CORBA Architectural Design," *Technical Report 01-007, NAI Labs*, 2001.

[Mellier-Smith97] P. M. Mellier-Smith, P. Narasimhany, L E Moserz. "Replica Consistency of CORBA Objects in Partitionable Distributed Systems", *Distributed Systems Engineering*, p 139-150, 1997

[Mishra93] S. Mishra, L. Peterson and R. D. Schlichting, "Consul: A Communications Substrate for Fault-Tolerant Distributed Programs", *Distributed Systems Engineering*, pp. 87-103, 1993.

[Morgan99] G. Morgan, S. K. Shrivastava, P. D. Ezhilchelvan and M.C. Little, "Design and implementation of a CORBA fault-tolerant object group service", *Proceedings of the conference on Distributed Applications and Interoperable Systems*, 1999.

[Mpoeleng03] D. Mpoeleng, P. D. Ezhilchelvan and N. A.Speirs. "From Crash Tolerance to Authenticated Byzantine Tolerance: a Structured Approach, the Cost and Benefits", *In Proceedings of the 2003 IEEE International Conference on Dependable Systems and Networks (DSN '03),* San Francisco, California, USA, 22-25 June 2003 pp. 227-236 IEEE Computer Society Press 2003.

[Narasimhan97] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Replica Consistency of CORBA Objects in Partitionable Distributed Systems", *Distributed Systems Engineering* (4), pp. 139-150, 1997.

[Narasimhan99a] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Providing support for survivable corba applications with the immune system," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, (Austin, TX), pp. 507–516, May 1999.

[Ptzmann96] B. Ptzmann and M. Waidner. "Information-theoretic pseudo-signatures and Byzantine agreement for $t >= n=3$." Research report, IBM Research, Nov. 1996. Submitted for Publication.

[Powell88] D. Powell, G. Bonn, D. Seaton, P. Veríssimo and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", *Digest of Papers, FTCS-18*, Tokyo, pp. 246-251, June 1988.

[Reiter96] M. K. Reiter, "Distributing trust with the rampart toolkit," *Communications of the ACM*, vol. 39, pp. 71–74, Apr. 1996.

[Reiter96] M. Reiter. "A Secure Group Membership Protocol." *IEEE Transactions on Software Engineering*, 22(1), 1996

[Renesse97] Renesse, R.V., Birman, K. P., Hayden, M., Vaysburd, A., and Karr, D. "Building adaptive systems using ensemble." *TR 97-1638*, Cornell University, July 1997.

[Rivest78] R. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM*, 21(2), pp.120-126, 1978.

[Rivest92] R. Rivest. "The MD5 Message-Digest Algorithm." *Internet RFC-1321*, 1992.

[Rodeh00a] O. Rodeh, K. P. Birman and D. Dolev. "Optimized group rekey for group communication systems." *In Symposium on Network and Distributed System Security*, USA, Internet Society. Febuary 2000.

[Rodeh00b] O Rodeh, K. P. Birman, and D. Dolev, "A study of group rekeying." *Technical Report TR2000-1791*, Cornell University Computer Science, March 2000.

[Rodeh98] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev, "Ensemble security," *Tech. Rep. TR98-1703*, Cornell University, Department of Computer, 1998.

[Rodrigues01] R. Rodrigues, M. Castro, and B. Liskov. "BASE: Using abstraction to improve fault tolerance." In *Proceedings of the18th Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[Scheider90] F. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial", *ACM Computing Surveys*, 22 (4), pp. 299-319, December 1990.

[Shrivastava90] S. K. Shrivastava, P. D. Ezhilchelvan, and N. A. Speirs,. "Fail-Controlled Processor Architectures For Distributed Systems", *Technical Report*, Department of Computing, University of Newcastle upon Tyne, 1990.

[Shrivastava91] S. K. Shrivastava, D. T. Seaton, N. Howard and N. A. Speirs, "Fail-Silent Hardware for Distributed Systems", *in* "DELTA-4: A

Generic Architecture for Dependable Distributed Systems", *(D. Powell, ed.), Chapter 11, Springer-Verlag*,1991.

[Shrivastava91] S. K. Shrivastava, P. D. Ezhilchelvan, N.A. Speirs, and D.T. Seaton, "Fail-Controlled Computer Architectures for Distributed Systems" *Tech. Report No. 333*,Computing Laboratory, University of Newcastle upon Tyne, July 1991.

[Schiper02] A. Schiper, "Failure Detection vs Group Membership in Fault-Tolerant Distributed Systems: Hidden Trade-Offs", H. Hermanns and R. Segala (Eds.). PAPM-PROBMIV 2002, LNCS 2399, pp. 1–15, Springer-Verlag Berlin Heidelberg 2002

[Speirs92] N. A. Speirs, P. D. Ezhilchelvan, S. K. Shrivastava, S. Tao and A. Tully, "TheDesign and Implementation of Voltan Fault-Tolerant Nodes for Distributed Systems",*Tech. Report, Computing Laboratory*, University of Newcastle upon Tyne, 1992.

[Speirs93] N. A.. Speirs, S. Tao, F. V. Brasileiro, P. D. Ezhilchelvan, and S. K Shrivastava, "The Design and Implementation of Voltan Fault-Tolerant Systems for Distributed Systems, Transputer Communications, 1(2), pp. 1-17, November 1993.

[Sullivan91] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability - a study of field failure in operating systems", *Proc. of Fault tolerant Computing Symp., FTCS-21*, pp.2-9, June 1991.

[Tao95] S. Tao, P. D. Ezhilchelvan and S. K. Shrivastava, "Focused Fault Injection Testing of Software Implemented Fault Tolerance Mechanisms of VOLTAN TMR Nodes." *IEE Distributed System Engineering Journal*, 2 (1), pp. 39-49, 1995.

[Theuretzbacher86] N. Theuretzbacher, "VOTRICS: Voting Triple Modular Computing System", *Digest of papers, FTCS-16*, Vienna, pp. 144-150, July 1986.

[Tully90] A. Tully and S.K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs", *Proc. 9th IEEE Symposium on Reliable Distributed Systems*, Huntsville, pp. 104-113, October 1990.

[Tully91] A. Tully, "Distributed Programming on Transputer Networks - an Object Oriented Model for Concurrent Processing", *Transputer Applications Conference*, TA91, Glasgow, August 1991.

[Venkatesan89] S. Venkatesan. "Reliable protocols for distributed termination detection." *IEEE Transactions on Reliability*, 38(1) pp.103–110, April 1989.

[Wallner98] Wallner, D., Harder, E., and Agee, R. "Key management for multicast: Issues and architectures." *Internet Draft* draft-wallner-key-arch-01.txt, IETF, Network Working Group, (Work in progress) September 1998.

[Wallner99] D. Wallner, E. Harderb and R. Agee." Key management for multicast: Issues and architectures." *Technical Report 2627*, IETF, Network security subworking Group, 1999.

[Wensley78] J. H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak and CB Weinstock., "SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control", Proc. *IEEE*, 66 (10), pp.1240-1255, October 1978.

[Wiener98] M. Wiener. "Performance Comparison of Public-Key Cryptosystems." *RSA Laboratories' CryptoBytes*, 4(1), 1998

[Wong98] C. K. Wong, M. Gouda and S. S. Lam. "Secure group communication using key graphs." *In SIGGCOM*, New York, USA, ACM press. September 1998. Science, September 1998.