© Copyright by Vishu Gupta, 2003

INTRUSION-TOLERANT STATE TRANSFER FOR GROUP COMMUNICATION SYSTEMS

BY

VISHU GUPTA

B.Tech, Indian Institute of Technology, Delhi, 2001

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

To my parents.

Acknowledgments

I would like to express my sincere gratitude towards my advisor, Prof. William H. Sanders for all the support and encouragement during the course of this research, and for all that I have learned from him.

The most important reason that PERFORM group was a great place to be was the people in the group. I would like to thank James Lyons, Hari Ramasamy and Kaustubh Joshi for their help, and numerous interesting and useful discussions I had with them. Sankalp Singh and Fabrice Stevens have been excellent officemates. I am thankful to all the past and present members of the group: Adnan Agbaria, Jenny Applequist, Tod Courtney, Michel Cukier, Dave Daly, Salem Derisavi, Sudha Krishnamurthy, Vinh Lam, Ryan Lefever, and Mouna Seri, for making my stay here a pleasurable experience. Particulary, I would like to thank Mouna for all the help received from her in the implementation of the work presented in this thesis, and Jenny, for the help she has always been, especially in making this thesis readable.

I am thankful to my roommate, Vineet Gupta, for all the help and co-operation during my two years here. I would also like to thank all my friends for helping me out whenever needed, and making life a fun experience.

My appreciation for all the support I have received from my family, and love for them cannot be described in words. I would like to thank them for always being there for me.

The research described in this thesis was funded by DARPA grant F30602-00-C-0172. I am grateful to Dr. Jaynarayan Lala and DARPA for providing direction to the ITUA project.

Table of Contents

List of Figures vi

List of Abbreviations viii

Chapter 1 Introduction 1
- 1.1 Motivation 1
- 1.2 Previous GCS Research 2
  - 1.2.1 Ensemble 3
  - 1.2.2 Practical Byzantine Fault Tolerance 3
  - 1.2.3 SecureRing 3
  - 1.2.4 Rampart 4
- 1.3 Previous State Transfer Research 4
  - 1.3.1 Eternal 5
  - 1.3.2 Maestro 5
- 1.4 Research Contributions and Thesis Organization 6

Chapter 2 Intrusion-Tolerant Architecture 8
- 2.1 The ITUA Intrusion Model 9
- 2.2 ITUA Architecture Overview 10
  - 2.2.1 Managers 10
  - 2.2.2 Intrusion-Tolerant Group Communication System 12
  - 2.2.3 Intrusion-Tolerant Gateway 14

Chapter 3 The Group Member Layer 18
- 3.1 Interface Details 18
- 3.2 Receiving Processor 20
- 3.3 Sending Processor 21
- 3.4 State Transfer Processor 22

Chapter 4 The State Transfer Protocol 24
- 4.1 System Model and Assumptions 25
- 4.2 High-level Protocol Description 26
- 4.3 Detailed Protocol Description 28
- 4.4 Correctness 38

Chapter 5 Performance Measurement 43
5.1 Experimental Setup 43
5.2 Experimental Results 44
5.2.1 Performance in the Absence of Corruption 44
5.2.2 Performance When the Leader is Corrupt 46
5.2.3 Performance When a Non-leader Member is Corrupt 48

Chapter 6 Conclusions and Future Work 51
6.1 Conclusions 51
6.2 Future Work 52

References 53

List of Figures

2.1 ITUA Architecture 11
2.2 IT-Gateway Architecture in ITUA 15

3.1 Interactions with the Receiving Processor 21
3.2 Interactions with the Sending Processor 22
3.3 Interactions with the State Transfer Processor 23

4.1 The Two-Phase State Transfer Protocol 27
4.2 The Protocol_State Transition Diagram 30
4.3 The *recv_new_view* Function 32
4.4 The *recv_ask_state* Function 34
4.5 The *recv_state* Function 36
4.6 The *recv_state_vote* Function 37
4.7 The *voting_timer* Timeout Handler 38
4.8 The *finish* Function 39

5.1 Performance in the Absence of Corruption 45
5.2 Performance When the Leader Does Not Reply 46
5.3 Performance When the Leader Sends a Wrong State 47
5.4 Performance When a Member Does Not Cast a Vote 48
5.5 Performance When a Member Casts a Wrong Vote 49

List of Abbreviations

GCS Group Communication System

ITUA Intrusion Tolerance by Unpredictability and Adaptation

CORBA Common Object Request Broker Architecture

IIOP Internet Inter-Orb Protocol

DII Dynamic Invocation Interface

UIUC University of Illinois at Urbana-Champaign

UCSB University of California at Santa Barbara

LAN Local Area Network

WAN Wide Area Network

Chapter 1

Introduction

1.1 Motivation

Distributed systems and networks have gained immense importance in society. Proliferation of such computer systems in every aspect of life, such as telecommunications, defense, energy and business, has led to ever-increasing reliance on them by government and industry. The increased accessibility has also made it easier for malicious entities to attempt attacks on these systems. Moreover, the practice of making large systems using off-the-shelf components, which are commonly seen to have serious security flaws, has led to increased probability that such attacks will be successful. Concern that successful attacks on such systems could have catastrophic effects on national security, combined with the economic and strategic importance of such systems, has led to widespread interest in ensuring their "survivability."

Ellison et al. [EFL+99] define *survivability* as a system's capability to fulfill its mission, in a timely manner, in the presence of attacks, failures, and accidents. Survivability helps a system to respond gracefully to attacks and continue to provide essential services even in the presence of successful attacks. The traditional approach to secure system design has been concentrated on building systems whose privilege levels cannot be compromised. However, the increasing complexity of distributed systems has made it increasingly difficult to ensure that all vulnerabilities of the system are covered. Survivability gains importance in such a scenario, in which the ability to perform in the presence of successful attacks is as important as trying to prevent any attacks.

One approach adopted for making survivable systems is fault tolerance or intrusion tolerance. Fault tolerance focuses on building systems using redundant components in such a way that *benign* component failures, such as fail-stopping or omission of some steps, can be tolerated. It includes fault detection, diagnosis, masking, confinement, compensation, and

recovery from faults. Intrusion tolerance can be defined as a system's ability to continue its essential functions even when significant portions of it have been compromised and may be in the control of an intelligent adversary. It resembles fault tolerance but is more difficult to achieve, because simplifying assumptions like that of *fail-stop* failures are not easily justifiable.

Intrusion tolerance can be regarded as a combination of traditional fault tolerance, computer security, and cryptography. Intrusion-tolerant systems adopt the concept of redundancy from fault tolerance, the concept of secure trust-free protocols from computer security, and concepts like encryption, digital signatures, and authentication from cryptography. These techniques can be used to provide intrusion tolerance at any level, such as the application, middleware, or operating system levels, or in the hardware. A common approach is to provide intrusion tolerance in middleware by providing intrusion-tolerant services to the application. The Intrusion Tolerance by Unpredictable Adaptation (ITUA) [CLP+01] project, undertaken by BBN Technologies, the University of Illinois at Urbana-Champaign, Boeing, and the University of Maryland, aims to provide intrusion tolerance by combining the techniques of replication and unpredictable response. There are also other related projects, like Enclaves [DSS01] at SRI, ITDOS [SMN+02] at NAI Labs, and MAFTIA [VNC00], which is a joint project of several European research labs.

The ITUA and ITDOS projects use the fault tolerance systems technique of replication to achieve survivability. These groups of replicated objects have several functions in common, like maintaining group-membership information, and need to communicate among themselves. Thus, there is a need for a lower level abstraction that can provide consistency among the processes in a group. Group communication systems (GCSs) are a convenient low-level abstraction for maintaining groups of processes and providing them with certain properties required by the applications based above them. Several group communication systems (GCSs) have been built in the past, providing different properties to system builders.

1.2 Previous GCS Research

Several attempts have been made in the design of fault-tolerant and intrusion-tolerant group communication systems. The ISIS system [Bir93, BvR94] from Cornell University was an early attempt at a fault-tolerant GCS. It led to other efforts for developing fault-tolerant group communication systems, like Horus [vRBM96] and Ensemble [Hay01] at Cornell University, Totem [MMSA+96] and SecureRing [KMMS98] at UCSB, and Transis [DM96] at the Hebrew University. We now describe some GCSs.

1.2.1 Ensemble

Ensemble [Hay98, Hay01] is a flexible group communication system that consists of a set of microprotocols. Each microprotocol provides a set of properties that may be required for a particular application. An appropriate subset of the microprotocols can be selected to form a protocol stack to satisfy a particular application's requirements. Since Ensemble is a layered stack, it is easy to add new layers and provide new properties. Ensemble has many security features [RBD01], but it can tolerate only crash failures. A C implementation of the important microprotocols of Ensemble, called *C-Ensemble*, has been created by Mark Hayden, who was one of the original developers of Ensemble. New intrusion-tolerant microprotocols for providing group membership and reliable, ordered message delivery were added in the C-Ensemble framework as part of the ITUA project. These properties are discussed in more detail in Section 2.2.2.

1.2.2 Practical Byzantine Fault Tolerance

PBFT [CL99b] implements a state machine replication protocol that correctly survives Byzantine faults in asynchronous networks. Requests from clients are jointly serviced by a state-machine-replicated server group, and replies are sent to the client individually by each server. Clients wait for a certain number of identical replies from the server group; the exact number of such replies depends on the Byzantine fault tolerance of the server group. The model can tolerate Byzantine behavior by no more than one-third of the server group processes. Public-key cryptography is used to authenticate messages. A modification of the protocol that uses message authentication codes in the fault-free case to reduce cryptographic overhead has been described in [CL99a]. The protocol provably does not rely on synchrony assumptions for safety, and makes only modest synchrony assumptions for liveness.

1.2.3 SecureRing

The SecureRing group communication system [KMMS98] provides reliable, ordered message delivery and group membership services in the presence of Byzantine faults. The membership protocol organizes the group members into a logical ring. Message multicast in the ring is controlled using a token. An unreliable Byzantine fault detector is used to report faulty processes to the membership protocol, which then reconfigures the system by forming a new ring consisting of apparently correct processors. The message delivery protocol ensures message delivery in a consistent total order to all members of the group. SecureRing's developers claim that their use of message digests in a signed token to allow a single digital

signature to cover multiple messages makes their protocol more efficient than protocols in which all messages need to be signed.

1.2.4 Rampart

Rampart [Rei95, Rei94a, Rei94b] toolkit has protocols for providing group membership services [Rei94b] and reliable, atomic multicast services [Rei94a] in the presence of Byzantine faults in a process group, provided that no more than one-third of the group members are faulty. The protocols use public key cryptography to authenticate messages. Processes communicate exclusively by sending and receiving messages over a completely connected, point-to-point network. The toolkit has been used to implement intrusion-tolerant applications, such as a sealed bid auction service [RF96] and a cryptographic key management service [RFLW96].

Use of group communication systems to provide intrusion tolerance creates an additional concern regarding maintenance of consistent state across *replicas*. Thus, transfer of state from existing replicas to replicas joining the group is needed. However, in a scenario in which successful attacks can be expected but it is nonetheless critical to perform, the mechanism for maintaining state consistency should be able to deal with the possibility of attack during the state transfer process or the possibility that state transfer will be obstructed by an intelligent adversary. This motivates the design of a protocol that can help remove implicit trust among the members of a group during state transfer.

1.3 Previous State Transfer Research

When a new replica joins the group, or, more generally, when several group components merge together, the state of the current replicas has to be updated into the new replica(s). The group communication system underneath decides on the direction of state transfer (which members need to receive the state from the other members). There have been a few attempts to implement state transfer between members of a group. Traditionally, protocols for state transfer have been designed for fault-tolerant systems. They can handle benign failures during state transfer, but do not expect an intelligent adversary with some control over some of the replicas to act systematically to disrupt the state transfer.

In [Bir96], Birman discusses the *push* and *pull* approaches for state transfer. In the *push* approach, one of the old group members (usually the coordinator) sends state to the new member. If it fails, the new coordinator starts the protocol from scratch, since it does

not know which portion of the state has already been transferred. If the members have shared state split among themselves, all of them send their copies to the joining member without being asked. In the *pull* approach, the joining member solicits state from the existing members. All the members save a copy of the state (which is the same at all the members) and only one of them (usually the coordinator) sends its state to the new member. The new member, on receiving the state, informs all the members that the state transfer was successful and that they can delete their saved states and proceed. Some efforts for design of a state transfer protocol have been discussed below.

1.3.1 Eternal

The Eternal system [LPSP+00] implements state transfer between members. Every replicated object in Eternal consists of three different kinds of states: application state, ORB state, and infrastructure state. Eternal uses state transfer primarily for recovery of replicas. Every replica supports a *checkpointable* interface: *getstate* and *setstate*. Any replica that wants to get the state can get it from an updated replica at any checkpoint using *getstate* and then install that state using *setstate*. Checkpoints help in restarting state transfer if the state transfer process is interrupted because of some fault. They also have passively replicated object groups, where the state of passive replicas are periodically updated from the primary replica. Eternal ensures only that the state updates at all the replicas occur in the same order with respect to the other messages.

1.3.2 Maestro

The Maestro tools [Vay98] provide an interface to the Ensemble GCS. Maestro implements a *pull*-type protocol for state transfer and also allows state merges when two groups join to form one group. In the case of state merges, the group that adopts the other group's state has to ask for state, which is the first step of any *pull*-type state transfer protocol.

Maestro defines so-called *state transfer views*, which are transient views in which some of the group members are in the process of doing state transfer. The members that have state are called *servers* and those that do not have state are called *clients*. A new member joins the group as a *client* and sends a request to the group asking for state. A transient view is installed in which the new member is listed as the *state-transferring server*. The coordinator sends a state back to the new member. Each instance of state transfer is treated as a transaction and is assigned a unique state transfer ID, which is used to restart the state transfer if the current state transfer process is interrupted. Once the new member receives

the state, it sends out a *xferDone* message to inform all the other members that the transfer is complete. All members of the group upon receiving that message, list the new member as a *normal server* in the group.

1.4 Research Contributions and Thesis Organization

The research presented in this thesis makes the following contributions:

Interface to a Group Communication System: Distributed system architectures for intrusion-tolerant systems generally do not provide correct abstraction levels from the perspective of an application developer. This usually results in underspecification of application-relevant properties and over-specification of low-level implementation details. Secure systems research generally involves research into components doing a certain task or providing a certain specific property. In the absence of well-defined interfaces, the system designers do not have a high-level composite view of the system properties, and the task of plugging various components to form a large intrusion-tolerant system becomes overly cumbersome. Also, having standard interfaces to some secure components that provide the correct level of abstraction helps system designers to test, experiment with and build systems matching their specific design priorities or system requirements. This motivates us to design and implement a generic interface to a group communication system. The interface we have designed and implemented is independent of the GCS used underneath it. It provides a standard set of functions, which are used in the ITUA architecture to interface with any GCS that provides the desired set of properties.

An Intrusion-Tolerant State Transfer Protocol: As discussed in the previous section, the current state transfer research focuses on fault-tolerant protocols. Fault-tolerant protocols ensure only that the state is transferred even in the presence of failures (for example, some replica fails or misses a step in the protocol, or a message is dropped). Such protocols rely on implicit trust in the replicas in a group, and are based on the assumption that any misbehavior of a replica is due merely to some fault in the system, and not to the influence of an intelligent adversary. Intrusion-tolerant group communication systems, such as the prototype developed as part of the ITUA architecture, provide intrusion-tolerant group communication. However, if one uses a fault-tolerant state transfer protocol with such a GCS to design a secure system, the result is that the whole system is no better than the weakest link, which is the fault-tolerant state transfer protocol. We have designed and implemented

an intrusion-tolerant state transfer protocol that does not rely on implicit trust in the replica providing the state.

This thesis is organized as follows. Chapter 2 discusses the intrusion-tolerant architecture used as part of the ITUA project in order to provide dependability and availability to critical client-server applications using an intrusion-tolerant GCS and a CORBA interface for applications to build upon. Chapter 3 discusses our design and implementation of an interface to any group communication system on top of which the CORBA interface resides in the ITUA architecture.

The design and implementation details of the intrusion-tolerant state transfer protocol have been discussed in detail in Chapter 4. We measured the performance of the state transfer protocol. The performance measurements give an idea of the overhead introduced in making the protocol intrusion-tolerant. Also, we measured the deterioration in performance by measuring the performance for infiltrated systems. The results are described in Chapter 5.

Chapter 6 outlines the conclusions we reached after design and implementation of the interface to the GCS and the state transfer protocol and measurement of the performance overhead. Some ideas for future work are discussed.

Chapter 2

Intrusion-Tolerant Architecture

In this chapter, we discuss the ITUA architecture for an intrusion-tolerant system by specifying the properties it aims to provide, the type of attacks it attempts to tolerate, the architecture of the system, and its associated environment. These details of ITUA are also presented in [Wea01].

The system to be hardened and made intrusion-tolerant consists of a set of non-overlapping *security domains*, where a security domain implements a boundary that is difficult for the attacker to cross. A security domain may consist of a single host or a set of hosts. A single host may itself form a security domain if it does not share administrative privileges with any other host. Another example of a security domain would be a set of hosts in a LAN separated from other networks through security mechanisms such as firewalls. Heterogeneity across domains (e.g., different operating systems) would help ensure that the attacker cannot exploit the same vulnerability to penetrate into all domains.

A domain will be in one of two states: *infiltrated* or *normal*. An infiltrated domain is one in which an attacker has gained access to, and can freely control or damage, the resources of that domain. For example, a domain consisting of a single UNIX host is considered infiltrated if the attacker has gained root access to that domain. A domain is considered infiltrated even if one host in the domain is infiltrated. A normal domain is one that has not been infiltrated. When first started, a domain is in the normal state. Once a domain becomes infiltrated, it can never be considered normal again.

A host in a domain can have one or more application or system processes running on it. New processes can be started on the host, and existing processes may be killed. Hence, the set of processes running on the host is dynamic. Each process is in one of two states: *correct* or *corrupt*. A correct process behaves according to its specifications, while a corrupt process does not. A process can become corrupt as a result of its domain having become infiltrated. Once a domain is infiltrated, any processes in that domain are considered potentially corrupt.

In order to provide applications with intrusion tolerance, multiple replicas of that application are started, distributed across multiple security domains. Each domain has only one replica corresponding to a particular application, because having more than one replica for the same application in a domain does not help intrusion tolerance, as all of the replicas would become corrupt together. The replicas interact with each other and coordinate their actions to form a *replication group*. Two replication groups can join together to form a bigger group called a *connection group* to facilitate communication between the two applications. The processes in such a group are replicas of one of the replication groups. Once a process becomes corrupt, it can never become correct again. To maintain the same level of intrusion tolerance, we can remove the corrupt replica from the system, and start a new replica in a normal domain to take the place of the corrupt replica. A process in a replication or connection group can also be referred to as a member of that group. A process can be a member of only one replication group, but multiple connection groups, at the same time.

2.1 The ITUA Intrusion Model

The ITUA intrusion model describes the set of attacks that the ITUA architecture is interested in defending against. The model is a trade-off between including the set of all possible attacks (some of which certainly cannot be defended against) and including only simple attacks that the ITUA system can easily withstand. It is an attempt to include the most feasible attacks, while at the same time providing some defense against the worst possible attacks. We do not describe the intrusion model by listing all of the attacks. Instead, we identify certain abstract features of the attacks considered[1], and describe the intrusion model in terms of those abstract features.

The attacker aims to disrupt the normal functioning of the system. He can do so by corrupting the processes. The attacker can continue to corrupt processes until the attack has been repulsed or the system's tolerance to corruption has been destroyed. An attack may be partially or completely successful. A partially successful attack will, at the very least, result in a degradation of the system's quality of service. A completely successful attack will result in the system's failure. A process can become corrupt only after the domain in which the process resides has been infiltrated. The attack primitive is the infiltration of a domain, which can be done by exploiting an operating system or application vulnerability, stealing a password, or introducing a virus, worm, or Trojan horse. Infiltrating a domain takes a non-negligible amount of time, and because of the heterogeneity of domains, there is a limit

[1] An attack is said to be *considered* in the ITUA intrusion model if the model is interested in defending against that attack.

on the number of domains that can be infiltrated simultaneously. That gives the defense some time to react to the attack. We call this the *staged* attack assumption, meaning that there is a non-negligible time between successive domain infiltrations. The ITUA model will not be able to defend against a situation in which all the domains housing the processes in a replication group are simultaneously corrupted. It also does not defend against a situation in which the attacker infiltrates the domains in stages, but the infiltrated domains show no observable signs of an intrusion until all the domains have been infiltrated. This is effectively the same as the first situation.

2.2 ITUA Architecture Overview

Figure 2.1 shows the basic ITUA architecture. The CORBA application specifies the intrusion tolerance requirements to the ITUA middleware, which then spawns the distributed application objects to satisfy those requirements. The existence of the ITUA middleware is totally transparent to these distributed CORBA application objects. From their perspective, they communicate with other distributed objects using remote method invocations, just as if the middleware were plain CORBA. In actuality, the inter-object interaction is intercepted and application-level behavior is altered by the Quality Objects framework within the ITUA middleware. This framework constitutes the in-band adaptation mechanisms for intrusion tolerance. It is complemented by out-of-band mechanisms, which involve intrusion response and recovery actions to manage and configure system resources independent of the inter-object communication.

The ITUA architecture uses a decentralized infrastructure to implement such out-of-band intrusion-tolerance mechanisms. This decentralized infrastructure comprises special processes called *managers* (see Section 2.2.1) distributed across the security domains. The ITUA middleware, in addition to implementing the management functions, also has a component called the *intrusion-tolerant gateway (IT-Gateway)* (see Section 2.2.3) that makes the remote method invocations of the distributed objects intrusion-tolerant. The gateway has intrusion-tolerant replication protocols and works on top of an intrusion-tolerant *group communication system (IT-GCS)* (see Section 2.2.2) to make the object remote method invocations intrusion-tolerant.

2.2.1 Managers

For a particular domain, the host on which the replica process corresponding to the application runs also has a manager running on it. The managers in all domains form a process

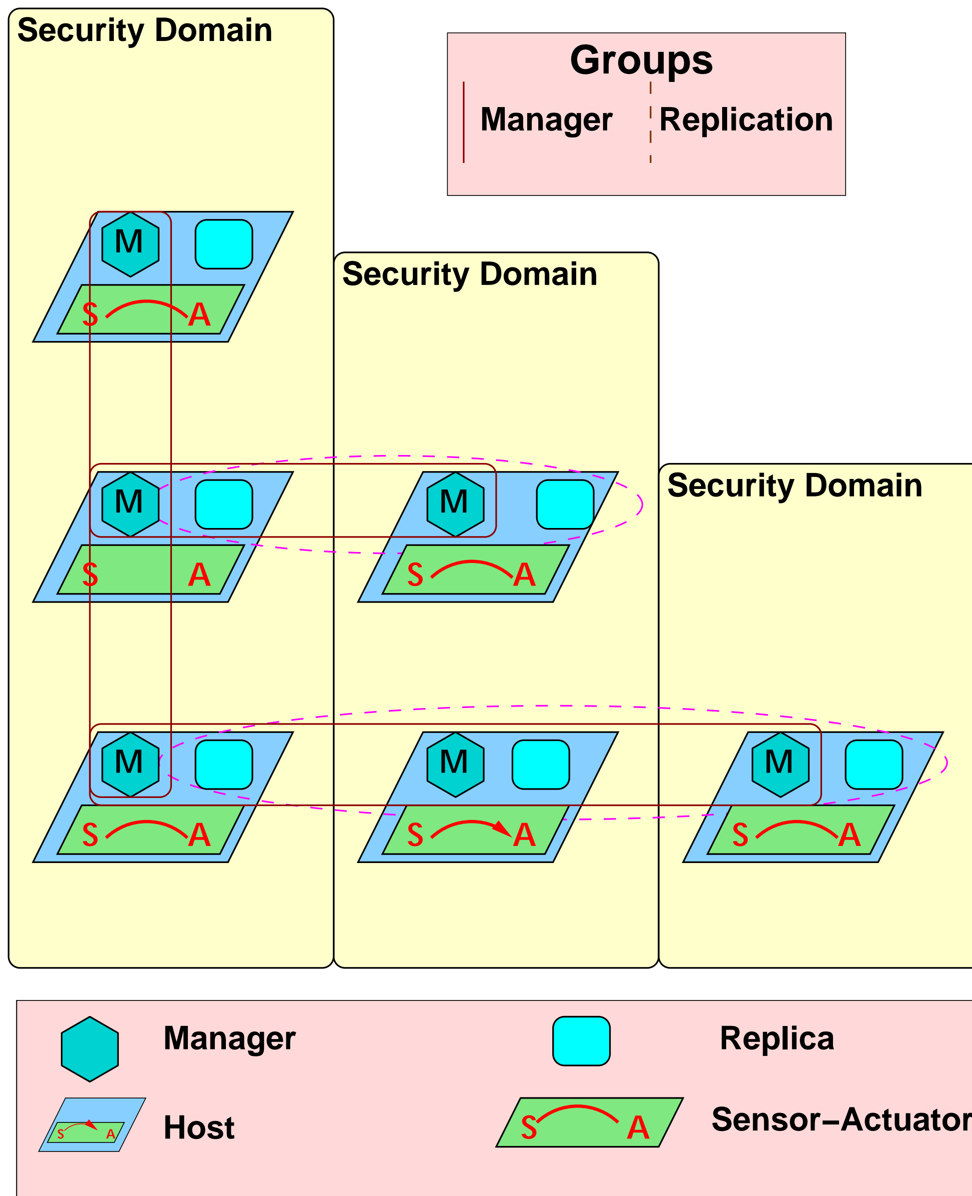

Figure 2.1: ITUA Architecture

group called the *manager group*. The manager performs two main functions: security and replication management. In its security role, the manager uses local sensor-actuator loops to collect information about potential intrusions and anomalous events, and to perform quick *knee-jerk* reactions to such events. The scope of these knee-jerk reactions is usually local, i.e., they involve resources on the manager's domain. However, in some cases, the manager for a particular domain also interacts with other domain managers to report anomalous events it observed in its domain and to make decisions that can affect hosts across all the domains, such as whether to change the security posture of a replication group. In its replication management role, the manager makes decisions regarding the starting and killing of replicas. It will also communicate with other managers to reach agreement about whether new replicas should be started to increase the intrusion tolerance of a replication group, and what security posture a replication group should change to after an observed anomaly.

2.2.2 Intrusion-Tolerant Group Communication System

The manager group, replication group, and connection group are all basically process groups consisting of processes communicating with each other. The intrusion-tolerant group communication system (IT-GCS) used in the ITUA architecture is based on the following assumptions.

System Model and Assumptions

The IT-GCS is based on a distributed system that consists of multiple hosts running several processes communicating over an unreliable network. The system is *timed asynchronous* [CF99]. Specifically, it is asynchronous in the sense that it does not require the existence of upper bounds on message transmission and scheduling delays. However, processes have access to local hardware clocks (which need not be synchronized). Time-outs are defined for message transmission and scheduling delays. When an experienced delay is greater than the associated time-out delay, a *performance failure* is said to have occurred. This *timed* asynchronous system assumption circumvents the impossibility of consensus in an asynchronous environment [KMMS97].

The protocols implemented in the GCS are concerned with one set of processes that wish to be in a group. The reliable multicast protocol uses message digests and digital signatures based on public key cryptosystem. Each process possesses a private key, public key pair. The private key of a process is known only to the process. The process uses its private key to sign messages digitally. Each process is able to obtain the public keys of other processes to verify signed messages. The protocols also use message digest functions, such as MD5, in which an

arbitrary-length message m is mapped to a fixed-length output $d(m)$. This function takes the message envelope as the data.

It is assumed that all processes are computationally bound. This means that a corrupt process cannot find two messages m and m' such that $m \neq m'$ and $d(m) = d(m')$; it cannot produce a valid signature of a correct process, or compute the message summarized by a digest from the digest. We assume that private keys cannot be stolen from correct processes.

The group membership protocol installs a series of views, V_0, V_1, ..., each of which is a set of identifiers of processes or members present in the view. The processes in a single view V have *ranks* or integer identifiers from 0 to $|V| - 1$. The processes are denoted by p_0, p_1, ..., $p_{|V|-1}$. When a view is installed, the lowest-ranked process in the view, p_0, is the leader of the view. The leader has no additional privileges, but does have additional responsibilities[2] compared to the rest of the group. If the leader is detected to be corrupt, the second-lowest-ranked process (we call this process the *deputy*) takes over as the new leader. If the deputy is also corrupt, the third-lowest-ranked process (the *deputy's deputy*) takes over as the new leader, and so on.

The following properties are provided by the group communication system to the process groups.

Reliable Multicast

The reliable delivery property ensures that the messages sent out in the group reach their destinations even in the presence of an asynchronous, unreliable network in which messages can be lost, reordered, or delayed, and also guarantees that the multicast message is delivered correctly (without a change in its contents) across all correct processes, even in the presence of corrupt senders. [Pan01] shows the details of the implementation of the Reliable Multicast property in the C-Ensemble group communication system.

The reliable multicast property maintains the following characteristics.

- **Integrity** For any message m and process p, a correct process q delivers m (purportedly from p) at most once, and, if p is proper, only if p multicast m.

- **Agreement** If process p is correct throughout a view and delivers m in that view, then all processes that are correct throughout that view deliver m in the same view.

- **Per-sender FIFO** If p and q are correct, and q delivers m_1 and the m_2 from p, p must have multicast m_1 and m_2 in that order.

[2] The responsibilities will be explained in Chapter 4.

Total Order

The total order property ensures that all the members of the group receive all messages in the same order. Thus, if two correct processes p and q deliver two messages m_1 and m_2, then they deliver them in the same order. Message sequence numbers and generating functions are used to provide the total ordering property. Implementation of the total ordering protocol in the IT-GCS has been discussed in [Pan01].

Group Membership

The group membership property of the IT-GCS ensures that all correct processes maintain the correct information about the current membership of the group in spite of intrusions. The group membership protocol used in the ITUA architecture (for details, see [Ram02]) is responsible for maintaining group membership information, removing processes from the group, and joining new processes into the group. In providing those functions, the protocol relies on the reliable multicast and total order properties.

2.2.3 Intrusion-Tolerant Gateway

The architecture of the IT-Gateway, implemented as part of the ITUA architecture, is shown in Figure 2.2. The gateway translates between the object-level messages at the CORBA application object and the process-level messages multicast by the IT-GCS. A *Group Factory* located on each host is used to create or kill that replica, and to obtain host information. The application object generates IIOP[3] messages to communicate with other distributed application objects. These IIOP messages are intercepted by the dynamic skeleton interface (DSI) of the handler for that object. The handler will then transparently communicate the messages to the recipient process groups. The handler also takes care that only a single response reaches the calling object. Thus, from the calling object's perspective, communication with other distributed objects is the same as if plain CORBA (as opposed to the ITUA middleware) were being used. The gateway also provides an infrastructure for implementing various replication and voting schemes, and for detecting and reporting faults at the replication-group level to the respective managers. Various components of the IT-Gateway are discussed below.

[3]IIOP is the Internet Inter-Orb Protocol, which specifies transfer syntax and message format to allow independently developed ORBs to communicate over TCP/IP.

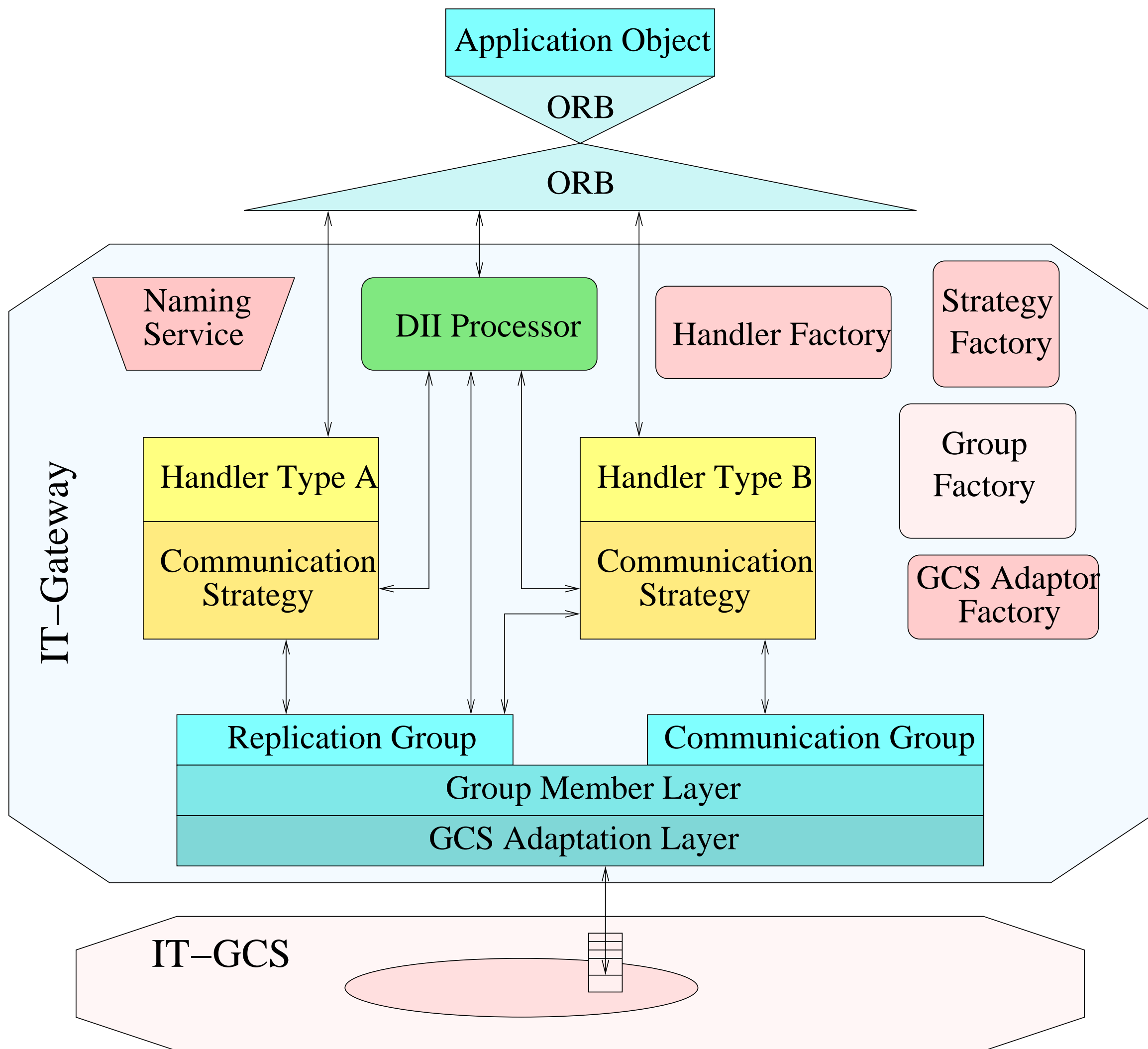

Figure 2.2: IT-Gateway Architecture in ITUA

Gateway ORB

The IT-Gateway uses a standard ORB (the TAO ORB [DoCS] is used in our implementation). The server ORB receives request invocations (IIOP messages) from the client, and returns the corresponding replies back to the client. It makes the IT-Gateway look like a normal CORBA server to the client. The client communicates with the IT-Gateway in the same way that it communicates with the standard CORBA server objects.

Naming Service

The naming service includes a naming context that is a table that maps object names to object references. It is a local naming service in a gateway, and provides a way for the client

application to use the gateway handlers.

In the gateway design, in order to direct all of the client's invocations to the gateway handlers instead of allowing them to go to remote server objects directly, the handler servants bind the naming service to associate the remote server objects' names with the handlers' own object references. Therefore, when a client application asks the naming service for the location of a remote server object, a handler servant's object reference is returned. The client is then able to forward its invocations to the gateway handler.

On the server side naming service, the server object binds the object reference associated with its name to the naming context, and the handler servants find the location of the server objects via the naming service instead of binding themselves to the naming service.

DII Processor

The DII processor is used to deliver invocations that it received from the handlers to the application object. In order to ensure strong data consistency among replicas, the DII processor contains a synchronous queue that ensures that the incoming invocations are delivered to the application in the order in which they were received from the group communication system. The construction for the invocation is provided by the Dynamic Invocation Interface. The Dynamic Invocation Interface allows the server gateway to pick the target server object at runtime and then dynamically invoke its methods. Using DII, the server gateway can invoke any operation without requiring precompiled stubs. This means that the gateway does not require compile-time knowledge about the server object's IDL methods. It dynamically constructs the request invocation by specifying the operation of the method and copying the arguments of the method from the gateway message. The use of DII in the server gateway creates a dynamic environment that allows the system to remain flexible and extensible. If the invocation is a synchronous CORBA message, the DII processor waits for a reply, and then returns the reply to the appropriate handler.

Handlers

Each handler servant is responsible for sending and receiving messages for a particular replicated object. When a client gateway handler receives an invocation from the application, it will save the invocation so that it can be used to reconstruct the CORBA message at the other end. It then constructs a gateway message that will carry the CORBA invocation to the gateways of the remote replicated servers, through the connection group.

Once a handler on the server side has received a gateway message, it removes the gateway header and gets the name of the operation and the arguments for the original IIOP invocation

from the gateway message payload. The handler uses its *Communication Strategy* to ensure that only a single request or response reaches the calling object based on multiple responses received from the group. The Communication Strategy of the handler implements consensus on every request to be sent to the application, using messages from all the replicas in the group. A *Strategy Factory* is used by the handler to obtain its communication strategy.

In order to send the invocation to the server object, the handler in the server gateway obtains the server's object reference from the naming service. It then constructs a new dynamic invocation based on the information from the gateway message and inserts the invocation into the synchronous queue in the DII processor.

Again, the server side handlers also participate in the replication protocols. As mentioned before, the choice of handler depends on the intrusion tolerance requirements of the application. A replica uses the *handler factory* to create the required handler. The handler factory has a handler repository that provides different types of handlers. We used the leader-only handler [Ren01] in our study.

Group Member Layer

The Group Member layer relieves the handler from having to know anything about the group communication system. It is used in the ITUA architecture, along with appropriate handlers, to construct Replication-Group Member and Connection-Group Member for the replication group and connection group, respectively. It uses functions provided by the GCS adaptation layer to provide the IT-Gateway with an interface to the GCS being used in the system. The Group Member layer is described in detail in the next chapter.

GCS Adaptation Layer

The GCS adaptation layer is the only component that knows what kind of GCS is being used for group communication. A GCS Adaptor is implemented specifically for a given GCS and provides a standard set of functions to the Group Member Layer. In the current implementation of the ITUA architecture, the GCS Adaptor interfaces with the HOT layer on top of the C-Ensemble group communication system. The gateway obtains the required GCS Adaptor for a given GCS from the *GCS Adaptor Factory*.

Chapter 3

The Group Member Layer

The Group Member Layer provides the handlers with a standard object-oriented interface to the group communication system. The interface visible to the handlers is independent of the GCS being used in the architecture. The implementation of this layer is done using the functions provided by the GCS adaptation layer. Thus, the Group Member layer can work on top of any GCS using the corresponding GCS Adaptor.

The Group Member layer also implements an intrusion-tolerant state transfer protocol that ensures removal of trust between members of the group during the state transfer process, which is discussed in detail in Section 3.4 and Chapter 4.

3.1 Interface Details

The group member layer provides the following functions to the other components of the IT-Gateway.

int register_handler (Replication_Handler &) This function can be called to register a handler with the group member. The Replication_Handler class can refer to any particular type of handler, like the leader-only handler that we use. The handler is added to the list of handlers maintained by the group member to which it delivers the messages received from the IT-GCS.

*int join (char **gcs_options)* This function is used by the IT-Gateway to join the group after the replica is started. The gcs_options can be set to correspond to the particular IT-GCS being used below the GCS_Adaptor, and according to the requirements of the application. Example options include the use of cryptography or reliability.

int leave (void) This function can be used to exit from the group of replicas.

int send (Member_Id&, Gateway_Message&) This function can be used to send a message to some other member in the group specified by the Member_Id. The message sent is a Gateway_Message that is converted to a GCS_Message (understood by the GCS) by the GCS Adaptor.

int send_to_leader (Gateway_Message&) This is an extension of the send function and is used to send a message to the leader of the group.

int suspect (Member_Id_List&) This is a very important function with respect to intrusion tolerance, as it enables the group member to send a suspect message corresponding to all the members it has detected to be corrupt in the group. If enough members suspect[1] a particular member, then that member will be removed from the group by the group membership protocol. Note that the Group Member Layer does not implement its own consensus algorithm to remove suspected members, but leverages an IT-GCS's group membership protocol to do so.

Member_Id& member_id (void) This allows the group member to retrieve its own reference.

int cast (Gateway_Message&) This function is used to cast a message to all the members in the current view.

int scast (Gateway_Message&) This function can be used to cast a message to all the stateful members in the group.

GCS_Adaptor_Base& GCS_adaptor (void) This function is called to obtain a reference to the GCS_Adaptor.

int GCS_adaptor (GCS_Adaptor_Base &) This function makes it possible to bind the GCS_Adaptor with the group member when the member process is started.

The following are functions in the Group Member that are used by the lower layer, that is, the GCS_Adaptor.

void recv_scast (Gateway_Message &) This function is called by the GCS_Adaptor to pass a scast message received from the IT-GCS to the group member layer.

void recv_cast (Gateway_Message &) This function is called by the GCS_Adaptor to pass a cast message received from the IT-GCS to the group member layer.

[1] Suspecting a member implies multicasting a "suspect" message in the group, indicating that the specified member is corrupt.

void recv_ptp (Gateway_Message &) This function is called by the GCS_Adaptor to pass a point-to-point message received from the IT-GCS to the group member layer.

void recv_view (Gateway_Message &) This function is called by the GCS_Adaptor to notify the group member of the change in group membership of the system.

void flush_block_getstate (int) This function is used by the State Transfer Processor (see Section 3.4) to ask the upper layers to finish the processing of all the enqueued messages, block further processing, and if required, return the state. The integer argument of the function specifies whether the application has been asked for state or not. In the current implementation, this function flushes all the handlers, blocks the DII_Processor so that no non-state transfer messages are processed during state transfer, and obtains the application state if the integer argument is set.

int set_application_state (ACE_Message_Block &) This function is used by the group member to install the state provided as argument. In the current implementation, the application state is provided as argument to this function and the function calls the DII_Processor to install the state.

The Group Member Layer is built of three component processors: Receiving Processor(described in Section 3.2), Sending Processor(described in Section 3.3), and State Transfer Processor(described in Section 3.4).

3.2 Receiving Processor

The receiving processor is the message-receiving component of the group member. It implements the functions used by the GCS adaptor to deliver data and system messages to the group member. The messages received by the receiving processor are GCS messages. The receiving processor maintains a message queue, and any incoming messages are enqueued into it. The messages are dequeued, converted to gateway messages, and dispatched to either appropriate handlers or the state transfer processor depending on the opcode. All state-transfer-related messages and view-change messages are sent to the state transfer processor, and all other messages are passed to the handlers. Figure 3.1 shows the interactions of different components of the system with the receiving processor. The functions implemented by the Receiving Processor are:

*int put (GCS_Message *)* This function is called by the GCS Adaptor to enqueue a GCS message into the receiving processor queue, from which messages are taken and processed in FIFO order by the *svc()* function.

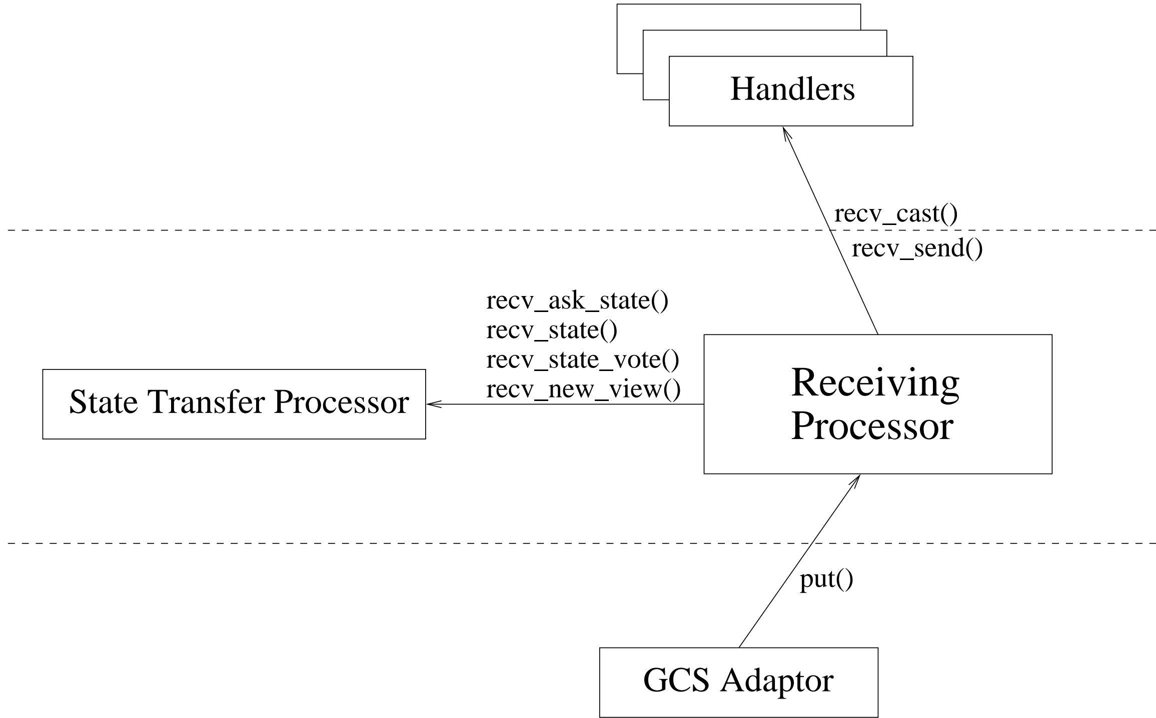

Figure 3.1: Interactions with the Receiving Processor

int svc (void) This function is indefinitely executed by a worker thread. It waits for and dequeues GCS messages from the queue and converts them to Gateway messages. It then checks whether the message is related to state transfer or is an application message. It processes any state transfer related message by calling appropriate functions in the State Transfer Processor (see Figure 3.1), as discussed in Section 3.4. It delivers any application messages to all handlers registered with the group member using functions *recv_cast* and *recv_send* (see Figure 3.1).

3.3 Sending Processor

The sending processor is the only way messages can be sent from the gateway or the group member layer to the group communication system. It does not maintain a message queue like the Receiving Processor does. As shown in Figure 3.2, the sending processor forwards all the gateway messages received from the handlers or the state transfer processor to the GCS adaptation layer after converting them to GCS messages.

int send(Gateway_Message &, Member_Id &, int gcs_opcode = GCS_Message::GW_DATA)
This function is called by other components of the group member layer to send a point-to-point message to a member. Since the message is in the Gateway format, it has

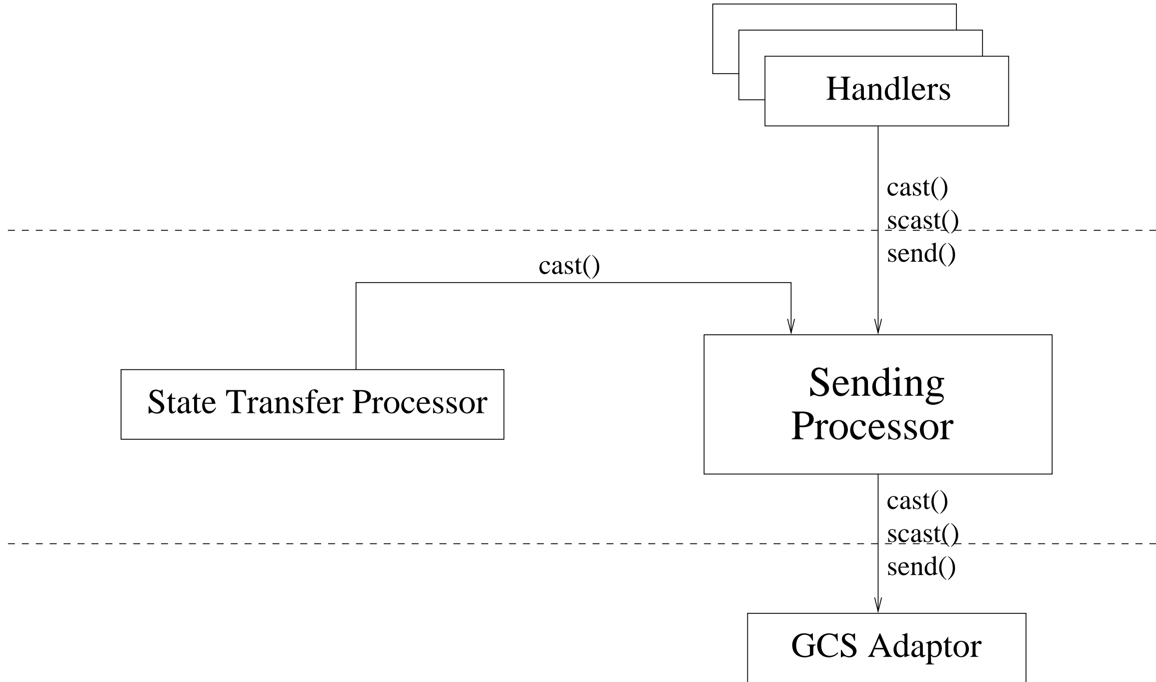

Figure 3.2: Interactions with the Sending Processor

to be converted to the GCS format, and for that the corresponding GCS opcode is specified by the caller. The default opcode is kept as *GW_DATA*, which is the opcode for application messages.

int cast(Gateway_Message &, int gcs_opcode = GCS_Message::GW_DATA) This function is called to cast a message to all the members in the current view.

int scast(Member_Id_List, Gateway_Message &, int gcs_opcode = GCS_Message::GW_DATA) This function is called to cast a message to a list of members specified by the data structure Member_Id_List.

int scast(Gateway_Message &, int gcs_opcode = GCS_Message::GW_DATA) This function is called to cast a message to all the stateful members in the current view.

int send_to_leader(Gateway_Message &, int gcs_opcode = GCS_Message::GW_DATA) This function is called to send a point-to-point message to the leader of the group.

3.4 State Transfer Processor

The state transfer processor implements the most important functionality of the group member layer, while the receiving processor and the sending processor mostly act as forwarders

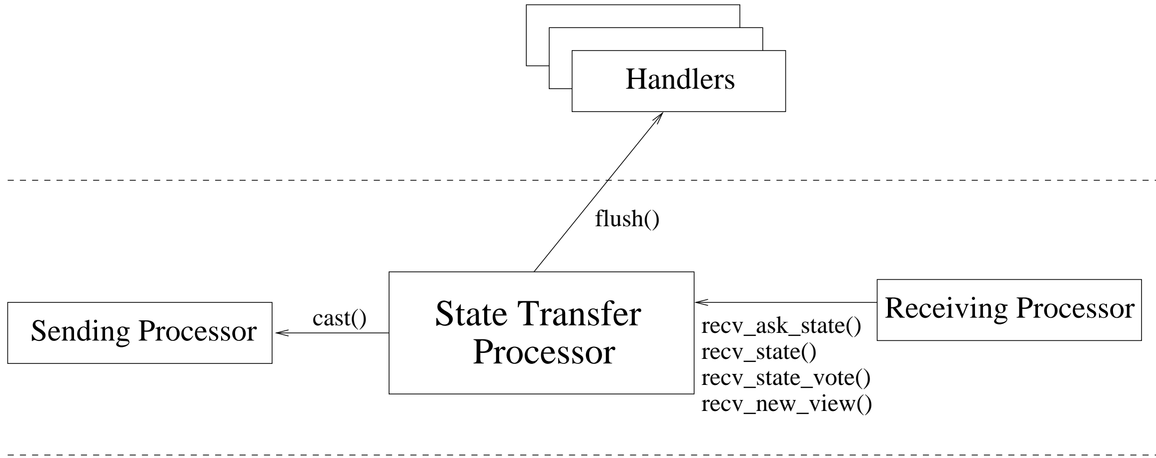

Figure 3.3: Interactions with the State Transfer Processor

and assist in converting messages from the gateway format to the GCS format and vice versa. The state transfer processor is pluggable into the system as long as it implements the same interface. So, any state transfer protocol can be plugged into the Group Member layer. The state transfer processor does not maintain any queue of messages. It receives state transfer related messages from the receiving processor, handles them appropriately to get information about the state transfer going on in the system, and uses the sending processor to send state transfer messages to the other members of the system. Figure 3.3 shows the interaction of the state transfer processor with the other components of the system. The State Transfer Processor calls function *cast()* on the Sending Processor and *flush()* on the handlers (by calling function *flush_block_getstate()* on the Group Member layer). The functions *recv_new_view()*, *recv_ask_state()*, *recv_state()*, and *recv_state_vote()* are called on the State Transfer Processor by the Receiving Processor upon receipt of messages with corresponding opcodes. Details of the implementation of this processor, along with details of the state transfer protocol for removing trust among replicas, are provided in Chapter 4.

Chapter 4

The State Transfer Protocol

In the ITUA architecture, the state transfer protocol is implemented in the State Transfer Processor (discussed in Section 3.4) to ensure removal of trust during state transfer between members in a group. To tolerate malicious intrusions in the system, the ITUA architecture must ensure that all the members of the group coordinate among themselves to make any decisions, without trusting any of the other members in the group. The removal of trust is particularly important in a dynamic environment because there, trust is not permanently associated with a particular host or machine, but depends only on whether the host or domain has been compromised due to an attack. In such a scenario, past performance of a member cannot be used as a basis for assuming its future behavior or estimating its "trustworthiness," because of the possibility of successful attacks.

Removal of trust between members requires a protocol that gets useful information from other group members, ensuring that the correctness of the information received does not depend in any way on whether the other group members are correct or corrupt, but is only dependent on certain reasonable assumptions about the current state of the system. The assumptions include certain properties, such as assured delivery of messages to all of their intended recipients (*reliable delivery*) [Pan01], delivery of all messages to all the recipients in the same order (*total order*) [Pan01], and consistent view of the composition of the group across all the members of the group (*group membership*) [Ram02], as discussed in detail in Section 2.2.2.

When a new member or members join a group, the group membership protocol ensures that the new view, with the new member(s) included, is updated in all the members of the group, including the new member(s). A group can consist of both stateful and stateless members as explained in Section 4.1. The state transfer protocol ensures that if some assumptions about the group size are met, a new stateful member will get the correct state of the system, consistent with the already existing correct stateful members of the group,

irrespective of the number of corrupt members or stateless members in the group.

A point to be noted here is that the current ITUA implementation allows only one member to join the group at a time. Also, the current architecture neither requires nor allows both stateful and stateless members to exist in the same group. The state transfer works within the current ITUA implementation but has been designed to be generic and to work equally well with any underlying GCS that supports the joining of many members to the group together or that allows both stateful and stateless members to be present in the group, provided that the assumptions stated in Section 4.1 are satisfied.

The rest of the chapter describes the state transfer protocol in detail. In particular, the next section discusses the system model and assumptions. Then we present a high-level description of the state transfer protocol, followed by section with more of its details. The last section discusses the correctness of the protocol and the rationale behind its design.

4.1 System Model and Assumptions

The state transfer protocol is built on top of the IT-GCS model discussed in Section 2.2.2. Each member in the group is either stateful or stateless. The stateful members maintain the application state of the system based on requests received or any processing done. Stateless members, on the other hand, do not maintain *any* state consistent with other members of the group, although they might have some local state of their own. If they do, it will not influence the replies sent out to the group, in response to the requests received. The provision for stateless members that are not concerned with the state of the system can be relevant in a connection group that has been formed for message exchange only. In ITUA architecture though, even the connection group members are stateful because of the requirement to maintain consistent handler states. Stateful members constitute a replication group that maintains some state. However, a replication group can also consist of stateless members if the application does not need the members to maintain a consistent state. A group consisting of both stateful and stateless members can be useful in scenarios in which a stateful replication group is interacting with some managers.

A member needs to know which members in its *view* are stateful and which are stateless. Otherwise, a corrupt stateless member could pretend to be stateful and hamper the state transfer protocol. This knowledge of other members is called *view_state*, because it is associated with the current view and its composition, irrespective of the application running on top of the system and its state. Stateful members maintain the correct *view_state* of the system by maintaining separate lists for the stateful and stateless members; stateless

members do not have the total information and maintain only partial *view_state* by automatically considering all members present in the group to be stateless. Another goal of the state transfer protocol is to update the correct *view_state* information in the new member that wants to be stateful.

The group membership protocol of a group containing both stateful and stateless members is required to ensure that the leader is always stateful. The process group can continue to provide correct service if there are no more than $f = \lfloor (|V| - 1)/3 \rfloor$ corrupt processes. Another requirement in a group with both stateless and stateful members is that the number of correct-stateful members should be at least $f + 1 = \lfloor (|V| + 2)/3 \rfloor$. The rationale behind these assumptions is explained in Section 4.4.

4.2 High-level Protocol Description

The state transfer protocol is essentially a voting protocol based on *pull*-type state transfer. The protocol supports both stateful and stateless members and state transfer is desired only when a stateful member joins the group. Thus, in absence of any state-related information from the group communication system, it needs to be a *pull*-type protocol. The protocol consists of two phases: the *state-cast phase* and the *voting phase* (see Figure 4.1). When a new member joins the group, the group membership protocol updates the view of all the existing and new members. If the new member joining the group intends to be a stateless member, no state transfer takes place, and the group can continue with its functions.

The *state-cast phase* is initiated when the new joining member (that intends to be stateful) multicasts an *ask_state* request to the other members of the group, as soon as it has received the view change. When the existing members receive that request, they block any further message processing except for state-transfer-related messages. The leader of the group responds to the request by blocking itself and multicasting its state to the whole group. If the leader is corrupt and does not reply to the *ask_state* message, it will be suspected by the correct stateful members of the group after a certain *state-cast timeout*. If that happens, the state transfer process will start again from the state-cast phase after the leader has been removed from the group and a new leader elected.

The *voting phase* starts when the state cast by the leader is received by the members in the group. When the new member receives the state, it stores the state but does not install it. The stateless members just cast a *neutral* vote on receiving the state from the leader. When a stateful member receives the state from the leader, it compares it to its own state. If the leader's state does not match its own state, the stateful member suspects the leader,

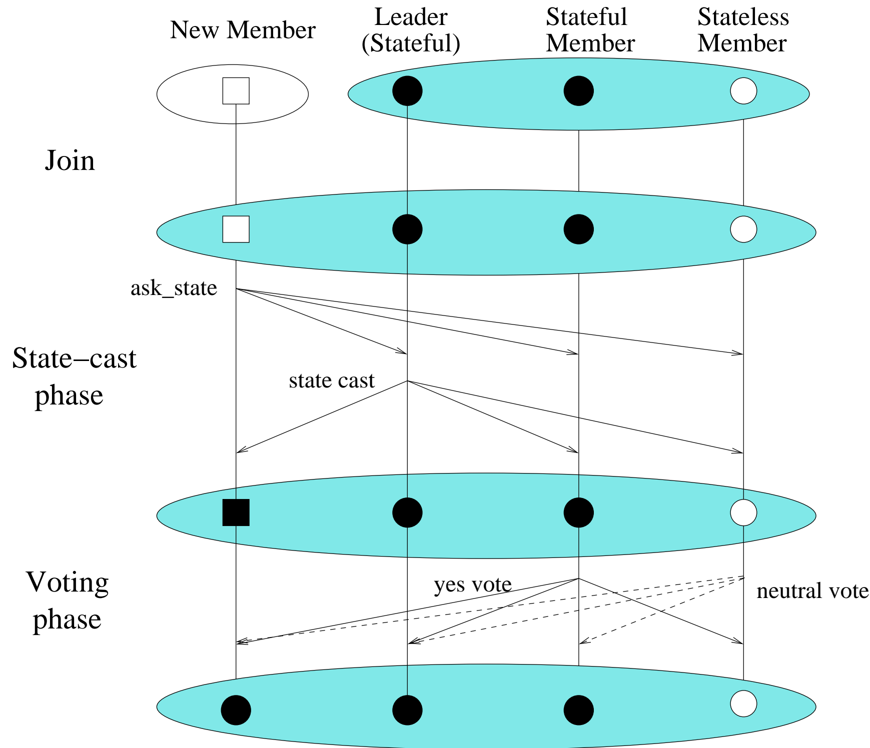

Figure 4.1: The Two-Phase State Transfer Protocol

sends a *no* vote and stops participating in the protocol. By that process, a corrupt leader will become suspected by *enough* correct stateful members (when it does not send a state or sends a wrong state), that it will be removed from the group. A new leader will be elected, and state transfer will start again from the state-cast phase after a view change.

If the leader is correct, all the correct stateful members multicast a *yes* vote after confirming that the state cast by the leader matches their own. The new member keeps collecting the *yes* votes from the members and adding them to its local list of stateful members in the group. All members also keep a count of the number of *yes* votes received. When all the votes have been received or a *voting timeout* has occurred, the protocol is said to have *completed* if all of the members have received at least $f + 1$ *yes* votes; otherwise, all the suspected members will be removed via a consensus among the correct members, and protocol will be repeated. If the protocol *completes*, the new member(s) install the state and update their *view_states*. If that happens, the stateful members will also update their *view_states* by moving the new member(s) to their list of stateful members, and will suspect any stateful members that voted *no* or did not vote at all or any stateless members that voted *yes* or *no*. Stateless members keep all the members in the view as stateless members and make no distinctions based on state.

We will show in Section 4.4 that if the assumptions in Section 4.1 hold, the state transfer protocol will always *complete*.

4.3 Detailed Protocol Description

This section provides a detailed description of the state transfer protocol. We start with a description of additional information that must be kept at each member for state transfer to work without the new member(s) trusting any existing members in the group.

At any time, each member of the group is in one of the following protocol_states: INITIAL, JOINING, NONBUFFERING, GETTINGSTATE, STATEFUL, STATELESS, STATEXFER, or STATELESSXFER.

- INITIAL: The protocol_state of a member when it starts.
- JOINING: This is the protocol_state when the member has formed a singleton group and is in the process of merging with the main group.
- STATEFUL: This is the protocol_state of a stateful member during the normal functioning of the group.
- STATELESS: This is the protocol_state of a stateless member during the normal functioning of the group. It is also the protocol_state of a new stateful member before it has initiated the state transfer process.
- NONBUFFERING: This is the protocol_state of a new member in the *state-cast phase* of the protocol (after it has sent out the request for state, and before it has received the state from the leader). While in this protocol_state, the member does not process any messages that come to it.
- GETTINGSTATE: This is the protocol_state of a member in the *voting phase* of the protocol (when it has received the state and is waiting for the state transfer protocol to *complete* before installing the state and proceeding as a stateful member). GETTINGSTATE is a blocking protocol_state, and a member in GETTINGSTATE protocol_state buffers any messages unrelated to state transfer.
- STATEXFER: This is the protocol_state of any stateful member during the period of state transfer. In this protocol_state, all messages not related to state transfer are blocked and buffered.

- StatelessXfer: This is the protocol_state of any stateless member during the period of state transfer. Again, all messages not related to state transfer are blocked and buffered.

Every member keeps a number of lists locally to keep track of the protocol_state and other information about other members of the group:

- *stateful_list:* In this list, the stateful members keep the identifiers of those members in the group that are known to be stateful. This list is empty for stateless members.
- *stateless_list:* This lists members in the group known to be stateless. Stateless members are not aware of whether the other members are stateful or stateless and keep identifiers corresponding to all the group members in this list.
- *xfer_list:* This lists members in the group that have requested state from the group and are in the process of getting state. This list is non-empty only during the duration of the state transfer protocol.
- *suspect_list:* This list consists of members that are suspected of bad behavior and should be removed. This list also is non-empty only during the state transfer process.
- *view:* This list consists of all the members in the group. It also is non-empty only during state transfer.

The *stateful_list* and *stateless_list* constitute the *view_state* of the system, although stateless members maintaining only partial *view_state* use only the *stateless_list*.

A new member starts up in the Initial protocol_state. The member has a variable *want_state* (initialized from the configuration files), which indicates whether this member wants to get the state of the group upon joining. *want_state* is 1 for stateful members, and 0 for stateless members. Thus, no state transfer is required for members with *want_state* equal to 0. The member makes a *Join* call that asks the IT-GCS to make it a part of the group. The IT-GCS first makes a singleton group consisting only of the new member, and the member receives a view-change notification. The singleton group then merges with the main group to complete the join process, and the second view change is received. Two view-change notifications are received even if the member is the first member of the group.

Protocol_State Transition: The protocol_state transition diagram corresponding to the state transfer protocol is shown in Figure 4.2. In a steady state, all the members are either in the protocol_state Stateful or in Stateless. A member starts with the protocol_state

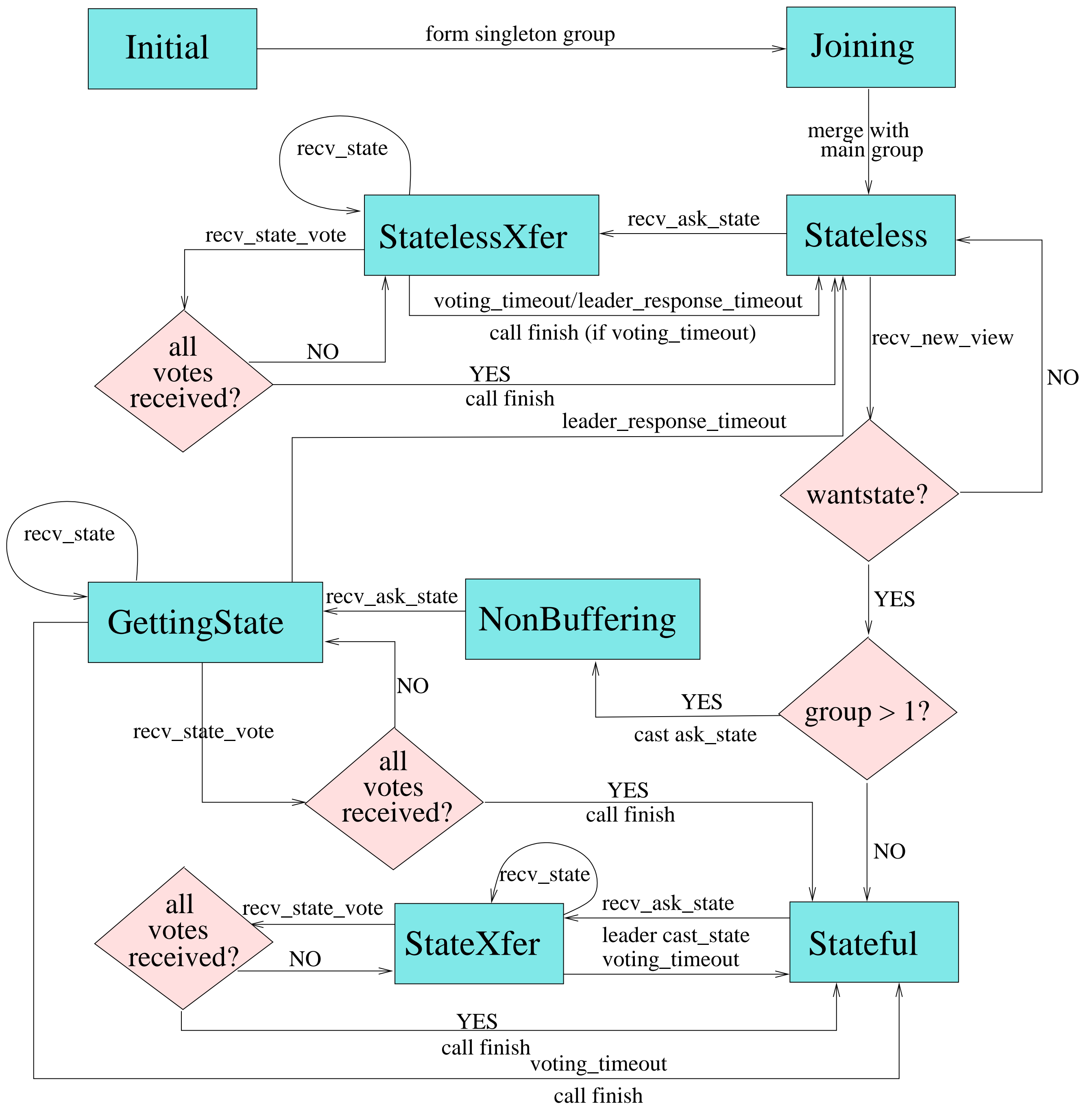

Figure 4.2: The Protocol_State Transition Diagram

INITIAL. It forms a singleton group and moves to protocol_state JOINING when it tries to merge with the main group. When a view change corresponding to the merge is received (function *recv_new_view*, Figure 4.3), the member moves to protocol_state STATELESS and checks whether it wants to become stateful. A member that does not want to become stateful stays stateless. A member that wants to become stateful simply moves to protocol_state STATEFUL if it is the only member in the group. Otherwise, it moves to protocol_state NONBUFFERING and sends an *ask_state* request. Function *recv_ask_state* (Figure 4.4) is called on all members when they receive the request for state. STATEFUL members move to protocol_state STATEXFER, and STATELESS members to STATELESSXFER. The new member moves to protocol_state GETTINGSTATE. The leader casts its state. If the leader does not send its state, *leader_response_timeout* occurs, STATELESSXFER members move back to protocol_state STATELESS, and STATEFUL members move back to protocol_state STATEFUL. Function *recv_state* (Figure 4.5) is called when a member receives the state from the leader. The members stay in the same protocol_state and those in protocol_states STATEXFER and STATELESSXFER cast their votes on the state received. When a vote is received, function *recv_state_vote* (Figure 4.6) is called. All members check whether they have received votes from everybody else, and, if they have not, stay in the same protocol_state. If the members have received all the votes or if the *voting_timeout* occurs (Figure 4.7), all execute the *finish* function (Figure 4.8). STATELESSXFER members move back to protocol_state STATELESS, STATEFUL members move back to STATEFUL and the members in protocol_state GETTINGSTATE also move to protocol_state STATEFUL. Thus, the protocol completes with all the members back in protocol_state STATELESS or STATEFUL.

recv_new_view: Figure 4.3 describes the action taken by members in different protocol_states when a view-change notification is received (and a call to *recv_new_view* is made). When the view change for the singleton view is received, the member (protocol_state INITIAL) changes its protocol_state to JOINING. After that, the IT-GCS merges the singleton group with the group to be joined and sends a new view. When the second view change is received by the new member (protocol_state JOINING), it changes to STATELESS and stores all the members in the view in *stateless_list*.

When the view change is received, any stateful or stateless members (in protocol_state STATEFUL or STATELESS, respectively) just update the view information that they have in the *stateful_list* and *stateless_list* by removing members no longer in the group and adding any new members in the group to the *stateless_list*. The stateless members install a new view with all members in the *stateless_list*. The stateful members install the new view with the already existing stateless members and the new member(s) in the *stateless_list*, and all

```
RECV_NEW_VIEW(view_data vd)
 1  cancel any pending timers
 2  view ← vd.member_list()
 3  switch (protocol_state)
 4     case INITIAL :
 5            protocol_state ← JOINING
 6     case JOINING :
 7     case NONBUFFERING :
 8     case GETTINGSTATE :
 9            protocol_state ← STATELESS
10            stateless_list ← view
11     case STATELESS :
12     case STATELESSXFER :
13            protocol_state ← STATELESS
14            stateless_list ← view
15     case STATEFUL :
16     case STATEXFER :
17            protocol_state ← STATEFUL
18            stateful_list ← stateful_list ∩ view
19            stateless_list ← view \ stateful_list
20  if protocol_state = STATELESS & want_state = 1
21     then if only one member in the group
22               then protocol_state ← STATEFUL
23                    move my_id from stateless_list to stateful_list
24               else protocol_state ← NONBUFFERING
25                    cast ask_state message
26  unblock message processing, if blocked
```

Figure 4.3: The *recv_new_view* Function

the stateful members in the *stateful_list*.

A view change can sometimes be made to occur during the state transfer protocol to restart the protocol from the beginning. If that happens, any member waiting to receive or install state during the state transfer process (protocol_state NONBUFFERING or GETTINGSTATE), when it receives a view change, assumes that the state transfer process was terminated, discards the state, and becomes STATELESS. Any stateless or stateful members that receive the view change when the state transfer protocol is progressing (protocol_state being STATELESSXFER or STATEXFER, respectively) also consider the state transfer to have

terminated, and move to protocol_states STATELESS or STATEFUL, respectively. They also update the *stateful_list* and *stateless_list* using the view information received. Function *recv_new_view* ensures that all members are in protocol_state STATELESS or STATEFUL and have the updated *view_state* information in the lists *stateless_list* and *stateful_list*. In case the view change terminates the state transfer process, the members also unblock message processing (blocked for the duration of state transfer) on receiving the view change.

If no stateless member in the group wants to become a stateful member, the state transfer protocol is not started, and the group can start normal functioning. On the other hand, if any stateless member has *want_state* equal to 1 and the number of members in the group is more than 1, the member sends a request for state (*ask_state* message) to the group, blocks itself, and moves to protocol_state NONBUFFERING (see Figure 4.3). This marks the beginning of the *state-cast phase* of the state transfer process. However, if the number of members in the group is 1 (that is, this member is the first member of the group), if it has *want_state* equal to 1, it just moves to protocol_state STATEFUL and does not try to get state from other members (because this is the only member). Any stateless members with *want_state* equal to 0 stay stateless.

recv_ask_state: When the group members receive a request for state, function *recv_ask_state*, as shown in Figure 4.4, is called. If the requester is not in *stateless_list*, the request is ignored. If any stateful member sees that the request is from a stateful member, it suspects the requester. Any stateful or stateless member, on getting an *ask_state* message from a stateless member, moves the member from *stateless_list* to *xfer_list*. It also flushes its handlers and starts blocking and buffering any incoming messages other than the state transfer messages. The leader of the group multicasts its state to the group. All non-leaders start a *state_cast_timer* and start waiting for the state from the leader.

If a member that has itself requested state (protocol_state NONBUFFERING) receives an *ask_state* message, it checks whether the requester is in the view and if it is, starts buffering non-state-transfer-related messages, moves the requesting member from *stateless_list* to *xfer_list* and moves to protocol_state GETTINGSTATE. Any GETTINGSTATE member that receives *ask_state* just moves the requesting member from *stateless_list* to the *xfer_list*. Stateful or stateless members that are already in the process of state transfer (protocol_state STATEXFER or STATELESSXFER) only move the requester from *stateless_list* to the *xfer_list* and continue with the protocol. At that point, if the request was a valid request (that is, it was from a NONBUFFERING member), all members move to GETTINGSTATE, STATEXFER, or STATELESSXFER protocol_state. However, if the request was invalid, there is no change in the members, and any malicious member might be suspected.

```
RECV_ASK_STATE(Member_Id origin)
 1  switch (protocol_state)
 2     case NONBUFFERING :
 3           if origin ∈ stateless_list
 4              then move origin from stateless_list to xfer_list
 5                   protocol_state ← GETTINGSTATE
 6                   block message processing
 7     case GETTINGSTATE :
 8           if origin ∈ stateless_list
 9              then move origin from stateless_list to xfer_list
10     case STATELESS :
11           if origin ∈ stateless_list
12              then protocol_state ← STATELESSXFER
13                   move origin from stateless_list to xfer_list
14                   block message processing
15                   start state_cast_timer
16     case STATEFUL :
17           if origin ∈ stateful_list
18              then suspect_list.insert(origin)
19                   SUSPECT(suspect_list)
20              else  if origin ∈ stateless_list
21                       then protocol_state ← STATEXFER
22                            move origin from stateless_list to xfer_list
23                            block message processing
24                            stored_state ← application state
25                            start state_cast_timer
26                            if is_leader
27                               then CAST_STATE(stored_state)
28     case STATELESSXFER :
29           if origin ∈ stateless_list
30              then move origin from stateless_list to xfer_list
31     case STATEXFER :
32           if origin ∈ stateful_list
33              then suspect_list.insert(origin)
34                   protocol_state ← STATEFUL
35                   SUSPECT(suspect_list)
36              else  if origin ∈ stateless_list
37                       then move origin from stateless_list to xfer_list
```

Figure 4.4: The *recv_ask_state* Function

If a non-leader in protocol_state StateXfer or StatelessXfer does not receive the state cast from the leader and the *state_cast_timer* goes off, the non-leader moves to protocol_state Stateful or Stateless, respectively, and suspects the leader. In the presence of certain minimum number of suspects, the normal view change starts, a new leader is elected, and the state transfer resumes after *recv_new_view*. As discussed in Section 4.4, the number of suspects needed to remove a corrupt member, is $f + 1$.

recv_state: The *voting phase* starts when all of the members have received a state cast (before the timeout) and *recv_state* is called, as shown in Figure 4.5. If the state was received from a non-leader, it is neglected. If the state was received from the leader, but the recipient is not in any of the "state-transfer protocol_states", that is, StateXfer, StatelessXfer, or GettingState, the sender is suspected, because the state transfer is not taking place. This prevents a malicious leader from stopping the group's functionality by initiating state transfer arbitrarily, and avoids Denial of Service attacks.

Members in protocol_state StateXfer or StatelessXfer cancel the *state_cast_timer* on getting the state from the leader. The StatelessXfer members cast the vote *neutral* because they do not have knowledge of the actual current state and cannot check whether the state cast by the leader is correct. Still, the vote cast by the stateless members is important for synchronization after the voting process is over. The StateXfer members compare the received state with their own. If the states match, a *yes* vote is multicast to the group. If the states do not match, the leader is assumed to be corrupt and a *no* vote is cast. If that happens, the member also moves to protocol_state Stateful and suspects the leader. The change of protocol_state indicates that this member no longer participates in the protocol and aims to participate only after the exclusion of the corrupt leader from the group. Any StatelessXfer or StateXfer members with matching state start waiting for votes with a certain *voting_timer*. They copy the *stateless_list* and *stateful_list* to *voter_list*. The *voter_list* contains the identifiers for members from which votes are to be received, including the member's own identifier. If the state was sent by the leader, any member in protocol_state GettingState (one that wants state) saves the state temporarily, and waits for votes on this state.

recv_state_vote: Every member maintains a *yes_count*, which is the number of *yes* votes received for the state cast by the leader. The function *recv_state_vote*, as shown in Figure 4.6, is called whenever a member receives a vote on the state cast by some member of the group. The sender of the vote is removed from the *voter_list*. If the sender was not in the list, the vote is neglected. If the sender was present in the *voter_list*, a GettingState or

```
RECV_STATE(Member_Id origin, ACE_Message_Blockstate)
 1  if protocol_state = STATELESS || protocol_state = STATEFUL
 2     then suspect_list.insert(origin)
 3          SUSPECT(suspect_list)
 4          return
 5  if origin ≠ leader()
 6     then suspect_list.insert(origin)
 7          return
 8  yes_count ← 0
 9  switch (protocol_state)
10     case GETTINGSTATE :
11          stored_state ← state
12          voter_list ← view \ xfer_list
13          start voting_timer
14     case STATELESSXFER :
15          cancel state_cast_timer
16          voter_list ← view \ xfer_list
17          Cast vote 'neutral'
18          start voting_timer
19     case STATEXFER :
20          cancel state_cast_timer
21          if stored_state = state
22             then voter_list ← view \ xfer_list
23                  Cast vote 'yes'
24                  start voting_timer
25             else Cast vote 'no'
26                  suspect_list.insert(origin)
27                  protocol_state ← STATEFUL
28                  SUSPECT(suspect_list)
29     case default :
30          suspect_list.insert(origin)
31          SUSPECT(suspect_list)
```

Figure 4.5: The *recv_state* Function

STATELESSXFER member also increments the *yes_count* for a *yes* vote and does nothing else for a *neutral* or *no* vote. A GETTINGSTATE member also adds the sender to *stateful_list*. That is important, because the state transfer protocol aims not only to install state in the new member(s) but also to inform them about the *view_state* of the group. A STATEXFER

```
RECV_STATE_VOTE(Member_Id origin, string vote)
 1  if origin ∉ voter_list
 2     then return
 3  voter_list.remove(origin)
 4  if vote =' neutral'
 5     then if protocol_state = GETTINGSTATE
 6                then stateless_list.insert(origin)
 7                else  if protocol_state = STATEXFER & origin ∈ stateful_list
 8                          then suspect_list.insert(origin)
 9                               stateful_list.remove(origin)
10     else  if vote =' yes'
11                then yes_count ++
12                     if protocol_state = GETTINGSTATE
13                        then stateful_list.insert(origin)
14                        else  if protocol_state = STATEXFER & origin ∈ stateless_list
15                                  then suspect_list.insert(origin)
16                                       stateless_list.remove(origin)
17     else
18           if protocol_state = STATEXFER
19              then suspect_list.insert(origin)
20  if voter_list = φ
21     then cancel voting_timer
22           FINISH()
```

Figure 4.6: The *recv_state_vote* Function

member checks which list the sender of the vote lies in. A *neutral* vote should be cast only by a member of the *stateless_list* and a *yes* or *no* only by a member of *stateful_list*; otherwise, the sender is suspected. For a *yes* vote cast by a member of *stateful_list*, the *yes_count* is incremented, while for a *no* vote, the member is added to *suspect_list* but not immediately suspected. That allows the correct members to suspect all the corrupt members together in the end, so as to reduce the time required by the protocol. It also prevents an intelligent attacker from delaying the state transfer by making a corrupt member misbehave by casting a *no* vote every time the state is cast. The *view_state* of the new member can get corrupted because of corrupt members' wrong votes, but those members will be suspected and removed from the group, and the correct *view_state* will be restored.

Figure 4.6 shows that after every vote, the members check whether the *voter_list* is empty. If all the votes have been received, the *voting_timer* is canceled, and the protocol is completed

by a call to *finish* (which will be discussed later in this section).

voting_timeout_handler: The *voting_timeout_handler*, as shown in Figure 4.7, is called if the protocol does not finish before the *voting_timer* goes off. It simply moves all the members from the *voter_list* (a list of those whose votes were not received before the timeout) to the *suspect_list*, and calls *finish* to complete the protocol.

```
VOTING_TIMEOUT_HANDLER()
1  if protocol_state ≠ GETTINGSTATE
2     then suspect_list ← suspect_list ∪ voter_list
3  FINISH()
```

Figure 4.7: The *voting_timer* Timeout Handler

finish: The function *finish*, as shown in Figure 4.8, *completes* the protocol. The protocol completion condition is that the *yes_count* should be greater than $\lfloor (|V| - 1)/3 \rfloor$ (that is, at least 1 more than f). This implies that at least one correct member has voted for the state. Thus, the state cannot be wrong, and the protocol should end with the installation of state in the new member and its addition to the *stateful_lists* at all the members. When *finish* is called and the completion condition holds true, the STATEXFER members add the members of the *xfer_list* to the *stateful_list*, while the STATELESSXFER members just add them to the *stateless_list*. The stateful or stateless members also move to protocol_state STATEFUL or STATELESS, respectively. The GETTINGSTATE members install the state when the completion condition holds true, add the *xfer_list* to the *stateful_list*, and move to protocol_state STATEFUL. They put all the remaining members into the *stateless_list*.

Whether the completion condition holds true or not, all stateful or stateless members also suspect the members in their *suspect_list*. All unblock further message processing and move to protocol_state STATEFUL or STATELESS.

4.4 Correctness

The state transfer protocol discussed above aims to enable state transfer from the existing stateful replicas in the group, to a new stateless replica that wants state, without any implicit trust being placed on the replica actually sending the state or the replicas voting for the state.

```
FINISH()
 1  switch (protocol_state)
 2    case STATEXFER :
 3        if yes_count > (view_size − 1)/3
 4          then stateful_list ← stateful_list ∪ xfer_list
 5               unblock message processing
 6        protocol_state ← STATEFUL
 7        if suspect_list ≠ φ
 8          then SUSPECT(suspect_list)
 9    case STATELESSXFER :
10        if yes_count > (view_size − 1)/3
11          then stateless_list ← stateless_list ∪ xfer_list
12               unblock message processing
13        protocol_state ← STATELESS
14        if suspect_list ≠ φ
15          then SUSPECT(suspect_list)
16    case GETTINGSTATE :
17        if yes_count > (view_size − 1)/3
18          then stateful_list ← stateful_list ∪ xfer_list
19               stateless_list ← stateless_list \ stateful_list
20               protocol_state ← STATEFUL
21               INSTALL_STATE()
22          else protocol_state ← STATELESS
```

Figure 4.8: The *finish* Function

Moreover, this protocol installs the *view_state* into the new member and tries to remove any members from the current *view_state* that misbehaved during the state transfer protocol. During this protocol, care is taken at every step to look suspiciously at every member's actions and to proceed further only after that action is proved to be correct by a majority or proved incorrect, in which case the member is suspected.

The protocol has been designed to tolerate malicious behavior of any member in the group as long as certain assumptions hold true. The possible intrusions are described below.

The leader is corrupt: If the leader is corrupt, it can try to harm the state transfer in two possible ways: by not casting the state or by casting the wrong state. The protocol handles both cases, but the handling does affect the performance of the protocol. The impact of the leader's corrupt behavior on the performance of the protocol is discussed in detail in

Section 5.2.2.

- If the leader does not cast the state, the *state-cast* timeout occurs, and the leader is suspected by all correct members (even stateless members keep track of this timer). Thus the protocol is restarted and the performance is degraded as the state-cast phase of the protocol is repeated.

- If the leader is corrupt and casts the wrong state, the assumption that there will be at least $f + 1$ correct stateful members implies that at least $f + 1$ suspects will be generated when the leader's state is checked, and that the leader will be suspected and removed. Again, this will also lead to degradation in performance, as the protocol will have to be restarted.

A non-leader member is corrupt: If a non-leader stateful or stateless member is corrupt, it can try to disrupt the state transfer in many ways. The protocol handles that with some degradation in performance, which is discussed in detail in Section 5.2.3. The ways in which a non-leader can misbehave are outlined below.

- If a non-leader misbehaves by sending state, the protocol ignores it. State sent by a non-leader is not even checked for correctness.

- A member can cast a wrong vote, but that will lead to its elimination from the group in the next view change. This won't even degrade performance, because all the noticeably corrupt members are removed only after the state transfer is complete.

- A corrupt member can degrade performance by not voting at all and forcing the *voting_timeout*, but that will lead to its exclusion from the group after the state transfer is complete.

New member is corrupt: If the new member is corrupt, it can misbehave by not asking for state, in which case it will be considered stateless. It can send a state itself, but the state would be neglected, as this member is not a leader. It can try to send a wrong vote, but that vote would also be neglected, as new members are not in the *voter_list*. The new member cannot do anything to corrupt other members' states or delay the state transfer process.

We made some important decisions during the design of this protocol to ensure removal of trust with little compromise on the performance of the protocol. Some design decisions and assumptions have been discussed below.

Significance of the leader casting the state: One important decision was to have the state cast to the new member by the leader of the group and not by any arbitrary member. Although, this imposes on the group membership protocol the restriction that it must have a stateful member as the leader of the group (if not all members are stateless), it has some advantages. Having the leader cast state means that there is none of the overhead that would result from a decision algorithm used to decide which member should cast the state. Also, it prevents faulty non-leaders from slowing down the protocol by casting the state when they should not. If the state cast by the leader is incorrect, the other members first remove the leader from the group and restart the protocol. It is necessary to restart the state transfer protocol and obtain a fresh state cast to allow the new member to get the correct state.

$f + 1$ **correct-stateful members necessary:** The group membership protocol, as discussed in Section 2.2.2 and in detail in [Ram02], requires that at least one-third of the group *suspect* a member before it decides that the member is faulty and has to be removed. In other words, if $\lfloor (|V| + 2)/3 \rfloor$ members suspect a particular member, then that member is removed from the group. This number is equal to $f + 1$, and implies that even if one correct member suspects a member, then that member should be eliminated from the group. During the state transfer process, a limitation is that only the stateful members know whether some other member is misbehaving by casting either the wrong state or a wrong vote. Hence we have the additional requirement that at least $f + 1$ members be stateful and correct. That, along with the assumption that at most f members can be simultaneously corrupt, implies that $2f + 1$ stateful members in the group will be sufficient for intrusion tolerance, though not necessary.

Rationale behind having stateless members cast a *neutral* vote: The protocol aims to implement transfer of both the application state and the *view_state* with no implicit trust in an asynchronous environment. The new member has no knowledge of the *view_state* of the system and uses the votes cast by the members only to construct the *view_state*. That is why the protocol involves the stateless members also casting a *neutral* vote on receiving the state. The *view_state* could have been constructed even without the *neutral* vote if the new member assumed that all members that have not cast their votes are stateless. However, that approach would require the protocol to continue to a timeout even when all members are correct, and thus would add delays. The timeout approach could be avoided by having stateful members cast a *transfer_complete* as they have the correct *view_state* and would know when all the stateful members have cast their votes, but that is not done because of the high additional message-passing and synchronization overhead involved.

Rationale behind waiting to receive votes from all the members during voting phase: The members wait to receive votes from all the members, or the *voting_timeout* to occur, before calling the function *finish*, instead of completing the protocol after receiving $f+1$ *yes* votes. The reason is that the state transfer protocol also aims to establish consistency in state across replicas. Thus, any malicious or accidentally corrupted replicas can be removed from the system. This helps in establishing the correct intrusion-tolerance level of the system so that corrective action (such as starting more replicas) can be taken.

Chapter 5

Performance Measurement

Facilitating state transfer without trusting the replica sending the state does impose additional protocol overhead. This chapter describes the measurements done to gauge the performance overhead and discusses the additional cost incurred due to our protocol. In particular, Section 5.1 describes the environment in which the experiments were conducted. It also discusses the metrics used to judge the performance of the protocol. Section 5.2 discusses the performance results obtained and their significance.

5.1 Experimental Setup

The tests were carried out on a testbed of ten 1GHz Pentium III CPU computers with 256Mb RAM each. The computers were connected by a full-duplex 100Mbps switched Ethernet network. The machines were otherwise unloaded, and a single process ran on each machine. The time measurements were taken in units of clock cycles using an assembly-level instruction provided by the Pentium instruction set; we converted the measurements to milliseconds for clarity and ease of presentation, by multiplying them by the appropriate constant multiplier.

The experiments were done using a host key size of 1024 bits. The state transferred had a size of 20 bytes. 50 runs of each experiment were done. The metrics measured were time taken for the state-cast and voting phases of the state transfer protocol and also the total time taken for state transfer to complete. Mean values and confidence intervals were calculated.

In addition to measuring the time spent in various phases of the protocol when the group members work according to the protocol specification, the performance measurements also examined cases in which the leader was corrupt or some non-leader member was misbehaving. For measuring the performance of the leader misbehaving when the group size is n, we started the leader with a command-line argument that configured it to misbehave when the group

size reaches n. Similarly, errors in non-leader group members were also introduced from the command line. Command-line arguments were also used to specify the type of misbehavior displayed, such as failing to respond to a request or sending a bad reply.

5.2 Experimental Results

This section presents the results of the various experiments we conducted, and draws some conclusions about the cost of removing trust from the state transfer process. In the graphs, all confidence intervals depicted by error lines perpendicular to the graph represent 95% confidence and were computed under the assumption that the samples were from a normal distribution.

We conducted performance studies for our state transfer protocol when all the members in the group were correct. We also introduced corrupt processes and did performance analysis for cases in which the leader was corrupt or some non-leader members were corrupt. The results and analysis follow.

5.2.1 Performance in the Absence of Corruption

We compared the performance of the state transfer protocol with the time taken for the state-cast phase of the protocol. It should be noted that the state-cast phase of the protocol consists only of the new replica casting a message and asking for state and the leader casting the state. This step will be part of any *pull*-type state transfer protocol, which is required if we want to have both stateless and stateful members in the group. Thus, the time taken during the state-cast phase of the protocol is expected to be less than the time taken by any protocol for state transfer. Thus, a comparison of the total protocol time with the time for state cast gives a reasonable estimation of the overhead incurred in removal of trust.

As can be seen in Figure 5.1, for small group sizes, the protocol overhead represented by the voting phase is almost negligible compared to the state-cast time. However, as the group size increases, we observe that the fraction of total time spent in protocol overhead increases from a small 7.3% for a group of size 4 to almost 20% of the total time for a group of size 10. A point to be noted here is that the time measured for state transfer does not include the time required to install the state at the new replica. The reason is that the time to install state is highly dependent on two things: the actual size of the state and, more importantly, the time taken by the IT-gateway to process the *install state* directive. Thus, if we were to include the time for state installation in the total time, that would further reduce the relative effect of protocol overhead on the total time taken by state transfer and installation.

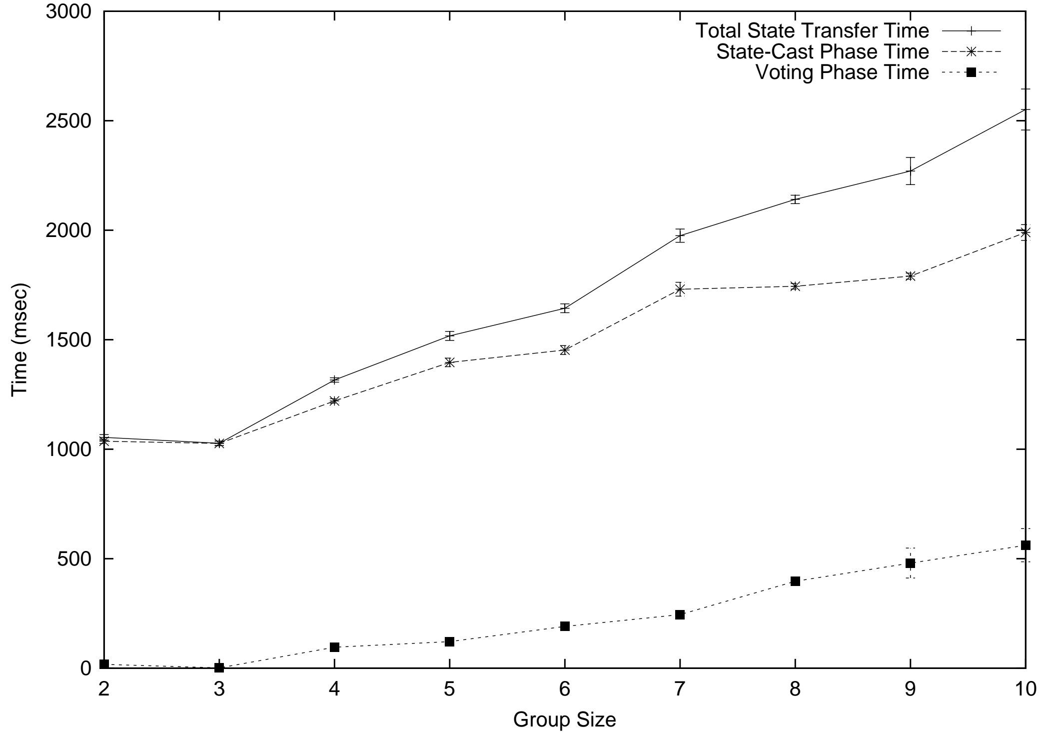

Figure 5.1: Performance in the Absence of Corruption

We notice in the figure that the voting time is almost 0 for group sizes 2 and 3, and even state-cast time does not change for these group sizes. The reason is that the IT-GCS does not use cryptography for group sizes less than 4, as it does not ensure any reliability for such small groups. That is why, the time for state transfer rises steeply for group size 4. The state-cast time increases with the increase in the number of groups and shows noticeable jumps for group sizes 7 and 10. The reason is that group sizes equaling $3f + 1$ increase the intrusion tolerance of the group, thus requires that IT-GCS have to check additional signatures to establish that the majority of replicas agree with any message sent [Pan01]. This overhead accounts for the increased time taken. On the other hand, no such noticeable jumps are observed in the time taken by the voting phase of the protocol. The reason is that in the voting phase, the new member waits for *all* the votes before installing the state, in order to enable the state transfer protocol to suspect and remove misbehaving members from the group. Therefore, every additional member in the group increases the time it takes for the members to receive and process the votes almost linearly, so we see an almost linear increase in the voting phase time.

5.2.2 Performance When the Leader is Corrupt

The protocol relies on the leader to respond to the request for state by sending its own state. Thus, to remove trust, it is necessary to consider a scenario in which the leader is corrupt and tries to harm the functioning of the group by not casting the state or by casting a wrong state. It is also essential to ensure that the performance degradation caused by a corrupt leader is minimal.

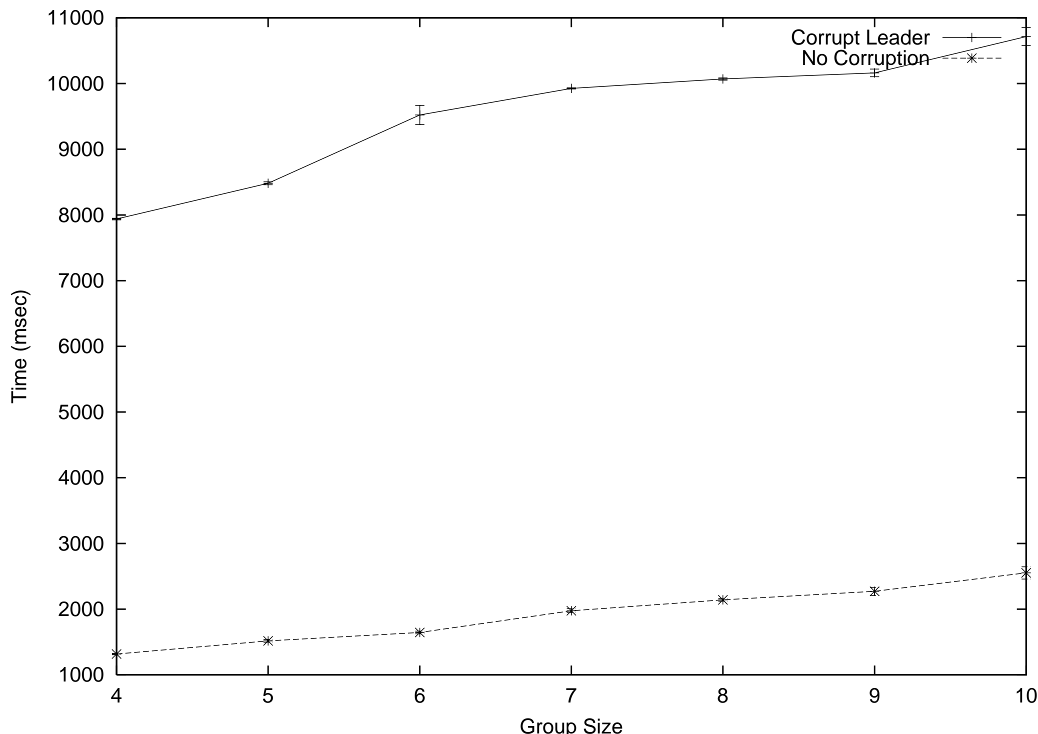

Figure 5.2: Performance When the Leader Does Not Reply

Corrupt Leader Does Not Respond: Figure 5.2 shows the performance degradation in state transfer when the leader does not reply to the request for state. Since the time taken for the leader to cast state is seen to be as much as 2 seconds for group size 10 (see Figure 5.1), the timeout value for the leader to cast state is safely taken to be 5 seconds. Note that a better estimate of timeout value could help improve the performance of the system. It is observed that when the leader is corrupt and does not reply to state requests, the time it takes for state transfer to complete for group size n includes the time it takes for state-cast timeout to occur, the time it then takes to suspect and remove the leader from the group, the time it takes for a view change to occur, and then the time it takes for state

transfer to occur in a group of size $n-1$. In the figure, while the time for state transfer in the absence of corruption increases from approximately 1.25 seconds for group size 4 to about 2.5 seconds for group size 10, the time when the leader is corrupt increases by almost 3 seconds from about 8 seconds for group size 4 to almost 11 seconds for size 10. The reason is that a higher group size leads to an increase in the time it takes to suspect and remove the leader, form a new group, and perform state transfer in the group after the view change.

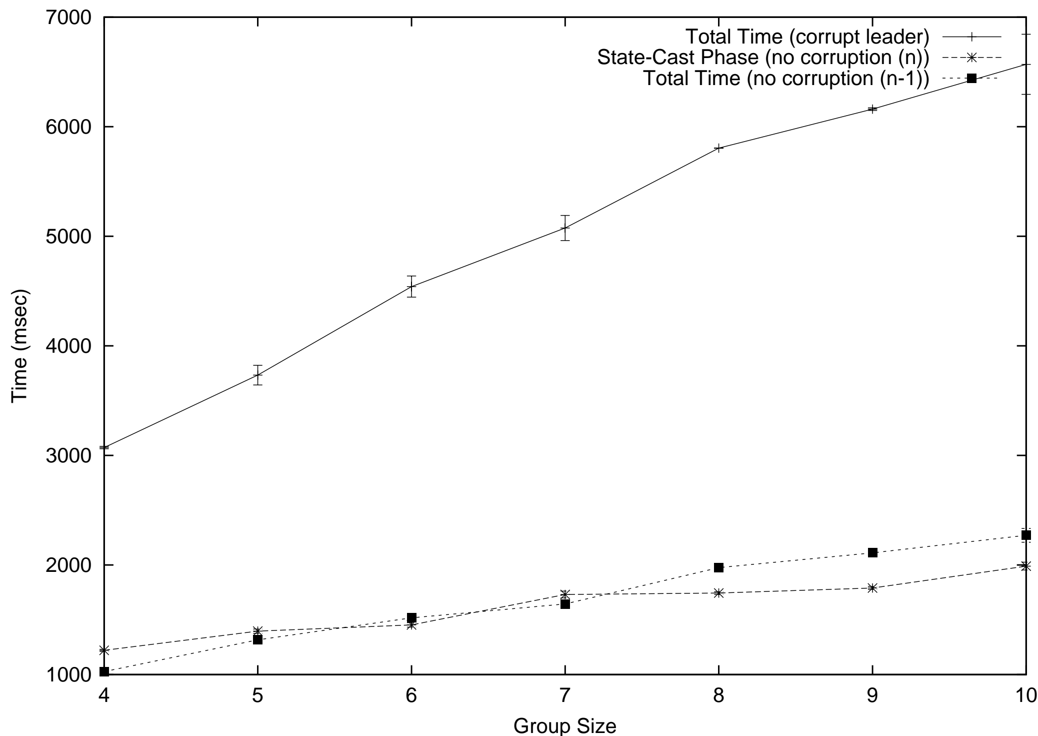

Figure 5.3: Performance When the Leader Sends a Wrong State

Corrupt Leader Sends Wrong State: A corrupt leader can also try to disrupt the functioning of the group by sending a wrong state to the new member. The leader would then be suspected and removed by the existing correct stateful members before state transfer is restarted in the new group with one less member. Figure 5.3 shows the degradation in performance because of the corrupt behavior of the leader. The time taken does not depend on the choice of timeout values, but only includes the time for two state-cast phases and one voting phase. Voting phase does not happen twice because on confirming the receipt of a wrong state, the correct members immediately suspect the leader, thus starting the process of consensus on suspects in the GCS. The figure also shows graphs of the times taken for state-cast when there is no corruption in the n-member group and the total time taken for

state transfer when there is no corruption in the $(n-1)$-member group. The graph of a leader responding with bad state would be the sum of these two graphs plus the time it would take for suspects to cause new group formation and view change. It can be seen that as group size increases, the corresponding increase in time when the leader is corrupt is more than double the increase we see when the leader is not corrupt. The reason is that the effect of increasing group size is doubled if the leader is corrupt because state-cast will happen twice. Also, increasing the group size increases greatly, the time it takes to suspect and remove the corrupt leader.

5.2.3 Performance When a Non-leader Member is Corrupt

Any corrupt member can try to disrupt the state transfer process by voting against a correct state sent by the leader, by voting for a bad state, or by not voting on the state.

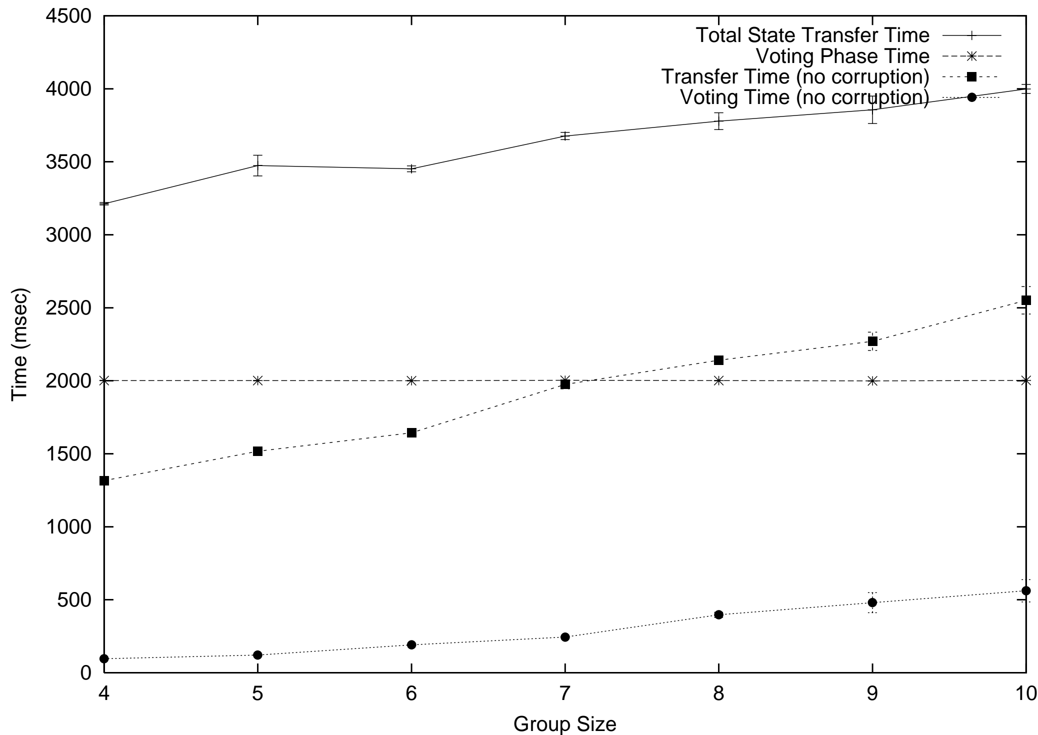

Figure 5.4: Performance When a Member Does Not Cast a Vote

Corrupt Non-leader Member Does Not Vote: When a member does not cast a vote on the state, the protocol waits until the voting timeout occurs. As seen in Figure 5.4, the timeout value is currently set to be 2 seconds, which is why the voting phase time is

constant at 2 seconds for all group sizes when some member does not cast its vote. There is no additional overhead, because the corrupt members are removed only after the state transfer process is complete. Thus, the total transfer time if there is a corrupt member reflects the increase in voting time, relative to the scenario with no corruption. We also notice that the increase in state transfer time with increasing group size is less if there is a corrupt member. The reason is that the voting phase time is constant when there is a corrupt member, whereas if there is no corruption, when increasing group size increases voting time, it also increases the total state transfer time.

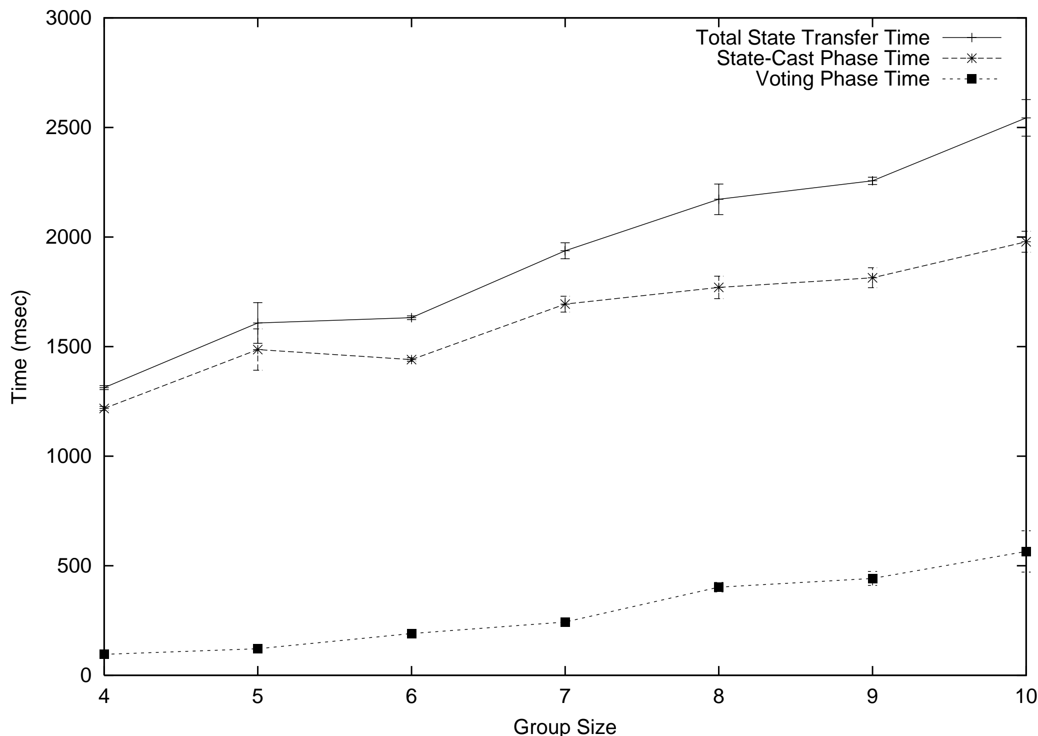

Figure 5.5: Performance When a Member Casts a Wrong Vote

Corrupt Non-leader Member Casts a Wrong Vote: When a member misbehaves by casting a wrong vote (e.g., a stateful member casts a *no* vote even when the state is correct, or casts a *yes* vote even when the state is bad, or a stateless member casts a *yes* or *no* vote instead of a *neutral* vote), the protocol does not suffer any degradation in performance. This is apparent if Figures 5.5 and 5.1 are compared. When a corrupt member casts a *yes* vote on a bad state, the protocol proceeds by suspecting and removing the leader, and the performance degradation is as shown in Figure 5.3. In other cases, when a wrong vote is cast and the state is correct, the misbehaving member is just put into a list of members

to be suspected, but a suspect message is not sent until the end of state transfer. Thus, if the assumptions in Section 4.1 are met, enough correct votes will be received to attain a majority, and state transfer will be completed in the same time it takes for state transfer to complete with all correct members.

It is observed that the state transfer protocol can handle all the corruption cases as long as the assumptions in Section 4.1 are met, but that some performance degradation is observed. We have studied the dependence on group size of both the time it takes to perform the two phases of state transfer and the total state transfer time. When timeouts occur in any of the phases, the observed performance degradation is considerable. The performance degradation in those cases, depends on the choice of timeout values and can be improved to some extent by more intelligent selection of the respective values for the state-cast and voting timeouts.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis discussed the need for an interface to a group communication system to facilitate ease of implementing intrusion-tolerant applications, independent of the type of GCS being used. It detailed the implementation of one such interface, describing the different components, their interactions among themselves, and details of the interface provided to the IT-Gateway, as well as functions and properties required from the group communication system. We also designed an intrusion-tolerant state transfer protocol that does not depend on implicit trust between replicas and ensures that correct state is installed in a new member even in the presence of corrupt members in the group. The protocol also aims to enable state transfer in groups with both stateful and stateless members. The protocol was implemented and extensively tested.

We did extensive performance analysis of our protocol in order to estimate the performance overhead associated with removal of trust during state transfer. Comparing the time taken for state transfer with the time taken for the state-cast phase shows that minimal overhead is introduced by the additional message exchange because of voting on the state. However, we observed that the overhead increases almost linearly with increasing group size. Thus, removing trust from the state transfer process does come at a price. Comparing the performance of the protocol when there are faulty non-leaders with the performance when there is no corruption, shows that the overhead caused by the presence of faulty members is very small when the members do not vote, and that in fact, no overhead is introduced when the members cast wrong votes on the state. On the other hand, the presence of a faulty leader does affect the performance considerably, because a large amount of time is spent in excluding the leader from the group and repeating phases of the state transfer protocol. Performance is negatively affected whenever timeouts occur because of the leader or non-

leaders not replying, although that effect can be reduced to some extent by a better choice of timeout values.

The state transfer protocol relies on certain assumptions about the group membership and certain properties provided by the IT-GCS, but the Group Member layer can be put on top of any GCS through use of an appropriate GCS Adaptor.

6.2 Future Work

The state transfer protocol has been designed to be generic so that it can work on top of any group communication system and allow the presence of both stateless and stateful members in groups of applications using the interface provided. The implementation has been tested and shown to work within the ITUA framework. It can interact with C-ensemble and Ensemble GCSs, and the IT-Gateway implemented in the ITUA architecture. However, the current implementation of the IT-Gateway has two types of stateful members: those that maintain only handler state (communication group members), and those that maintain both handler state and application state (replication group members). Further work is needed to modify the ITUA handlers to present a consistent notion of state to the Group Member layer.

The correctness of the state transfer protocol has been discussed informally, but we have not proven it formally. Considering the high possibility of errors in designing complex distributed systems, formal validation of the protocol is an important future research step.

We observed that the slower phase of the protocol was the state-cast phase. More work can be done in the area of improving the time taken to send the state. For example, we might have more than one stateful member cast portions of state to the new member, and check whether the time taken for the slower state-cast phase is reduced. Furthermore, if any improvement is observed, strategies for voting and removal of trust in such a scenario should be looked into.

The protocol uses two timeouts, which were fixed for the timing experiments. In fact, the performance in the presence of corrupt members that do not respond, depends heavily on those timeout values. Thus, we need to pursue ways to obtain better timeout values. In particular, we might look into making the group member learn to adjust the timeout values dynamically based on the message passing delays observed.

References

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*, chapter The Virtually Synchronous Execution Model, page 293. Manning, 1996.

[BvR94] Kenneth P. Birman and Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[CF99] Flaviu Cristian and Christof Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[CL99a] Miguel Castro and Barbara Liskov. Authenticated Byzantine fault tolerance without public-key cryptography. MIT/LCS/TM 589, MIT Laboratory of Computer Science, 1999.

[CL99b] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, February 1999.

[CLP+01] Michel Cukier, James Lyons, Prashant Pandey, HariGovind V. Ramasamy, William H. Sanders, Partha Pal, Franklin Webber, Richard Schantz, Joseph Loyall, Ronald Watro, Michael Atighetchi, and Jeanna Gossett. Intrusion tolerance approaches in ITUA. In *Supplement of the 2001 International Conference on Dependable Systems and Networks (Fast Abstracts)*, pages B64–B65, Göteborg, Sweden, July 2001.

[DM96] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[DoCS] Washington University Department of Computer Science. Real-time corba with tao (the ace orb). http://www.cs.wustl.edu/ schmidt/TAO.html.

[DSS01] B. Dutertre, H. Saïdi, and V. Stavridou. Intrusion-tolerant group management in Enclaves. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*, pages 203–212, Göteborg, Sweden, July 2001.

[EFL+99] R.J. Ellison, D.A. Fisher, R.C. Linger, H.F. Lipson, T.A. Longstaff, and N.R. Mead. Survivability: Protecting your critical systems. *IEEE Internet Computing*, Nov–Dec 1999. Vol 3.

[Hay98] Mark Hayden. *The Ensemble System.* PhD thesis, Cornell University, 1998.

[Hay01] Mark Hayden. *Ensemble Reference Manual.* Cornell University, August 2001.

[KMMS97] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 61–75, December 1997.

[KMMS98] Kim Potter Kihlstrom, Louise E. Moser, and P. Michael Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st IEEE Hawaii International Conference on System Sciences*, volume 3, pages 317–326, Kona, Hawaii, January 1998.

[LPSP+00] L.E.Moser, P.M.Melliar-Smith, P.Narsimhan, L.A.Tewksbury, and V.Kalogeraki. Eternal: Fault tolerance and live upgrades for distributed object systems. In *Proceedings of the IEEE Information Survivability Conference*, Hilton Head, SC, January 2000.

[MMSA+96] Louise E. Moser, P. Michael Melliar-Smith, Deborah A. Agarwal, Ravi Krishna Budhia, and C. Amy Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[Pan01] Prashant Pandey. Reliable delivery and ordering mechanisms for an intrusion-tolerant group communication system. Master's thesis, University of Illinois at Urbana-Champaign, 2001.

[Ram02] Hari Ramasamy. A group membership protocol for an intrusion-tolerant group communication system. Master's thesis, University of Illinois at Urbana-Champaign, 2002.

[RBD01] Ohad Rodeh, Ken Birman, and Danny Dolev. The architecture and performance of security protocols in the Ensemble group communication system. TR2000 1822, Cornell University, October 2001.

[Rei94a] Michael K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, 1994.

[Rei94b] Michael K. Reiter. A secure group membership protocol. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 176–189, May 1994.

[Rei95] Michael K. Reiter. The Rampart toolkit for building high-integrity services. *Lecture Notes in Computer Science*, 938:99–110, 1995.

[Ren01] Yansong [Jennifer] Ren. *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[RF96] Michael K. Reiter and M. K. Franklin. The design and implementation of a secure auction service. *IEEE Transactions on Software Engineering*, 22(5):302–312, May 1996.

[RFLW96] Michael K. Reiter, M. K. Franklin, J.B. Lacy, and R.N. Wright. The Omega Key Management Service. *Journal of Computer Security*, 4(4):267–287, 1996.

[SMN+02] David Sames, Brian Matt, Brian Niebuhr, Gregg Tally, Brent Whitmore, and David Bakken. Developing a Heterogeneous Intrusion Tolerant CORBA System. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, June 2002.

[Vay98] Alexey Vaysburd. *Building Reliable Interoperable Distributed Objects with the Maestro Tools*. PhD thesis, Cornell University, 1998.

[VNC00] Paulo Veríssimo, Nuno Ferreira Neves, and Miguel Correia. The middleware architecture of MAFTIA: A blueprint. DI/FCUL TR 00–6, Department of Computer Science, University of Lisbon, September 2000.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[Wea01] Franklin Webber et al. The ITUA intrusion model. available at http://www.dist-systems.bbn.com/projects/ITUA/model.html, August 2001.