RELIABLE DELIVERY AND ORDERING MECHANISMS FOR AN INTRUSION-TOLERANT GROUP COMMUNICATION SYSTEM

BY

PRASHANT PANDEY

B.ENGR., Birla Institute of Technology and Science, 1999

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, 2001

Urbana, Illinois

To my parents.

Acknowledgments

I would like to thank my advisor, Prof. William H. Sanders, for guiding and encouraging me through the course of this research. Dr. Michel Cukier has also helped me immensely with his many insightful suggestions about my work.

My fellow research students James Lyons and Hari Govind Ramasamy have been excellent colleagues. I would also like to thank Partha Pal, Franklin Webber, Ronald Watro, Richard Schantz, Paul Rubel, Joseph Loyall, Michael Atighetchi, and Chris Jones at BBN Technologies for many interesting discussions and for their contributions to the ITUA project.

I am grateful to the other occupants of my office, Kaustubh Joshi and Ryan Lefever, for making Room 230 CSL a great place to work. I would also like to thank other members of the PERFORM research group, Graham Clark, Tod Courtney, Mouna Seri, Sudha Krishnamurthy, Dave Daly, Salem Derisavi, and Yi-Wei Teo for many enjoyable conversations. I am grateful to Jenny Applequist for her help at all times, and in particular, in editing this thesis.

I am also grateful to the Defense Advanced Research Projects Agency for funding this research under contract F30602-00-C-0712. In particular, I would like to thank Dr. Jaynarayan Lala, program manager of OASIS, for providing guidance to the ITUA project.

I would like to thank Khagendra Gupta for being a very helpful roommate. I am grateful to my parents and sister for being supportive throughout my years in graduate school. I owe my deepest appreciation to my girlfriend, Gitanjali Kumar, for keeping me motivated.

Table of Contents

Chapter 1 Introduction 1
1.1 Intrusion Tolerance by Unpredictable Adaptation 3
1.1.1 The ITUA Environment 3
1.1.2 The ITUA Intrusion Model 4
1.1.3 ITUA Architecture Overview 5
1.2 Previous GCS Research 9
1.2.1 Practical Byzantine Fault Tolerance 9
1.2.2 Rampart 9
1.2.3 Ensemble 10
1.2.4 SecureRing 10
1.3 Research Contributions and Thesis Organization 10

Chapter 2 New Protocols in the ITUA GCS 12
2.1 System Model and Assumptions 13
2.2 Reliable Multicast 14
2.2.1 Reliable Multicast Protocol Description 15
2.2.2 Reliable Multicast Protocol Properties 22
2.3 Total Ordering Protocol 24
2.3.1 High-level Protocol Description 25
2.3.2 Detailed Protocol Description 26
2.3.3 Sequence Number Generating Functions 28
2.3.4 Total Ordering Protocol Properties 31

Chapter 3 Implementation Details 32
3.1 Layering Model 32
3.2 Protocol Stack for the Intrusion-tolerant GCS 35
3.3 Infrastructure Implementation 36
3.4 Cryptography Details 37
3.5 Reliable Multicast 39
3.6 Total Ordering 40

Chapter 4 Performance Measurement 43
4.1 Experimental Setup 43
4.1.1 Application R: Response Time for a Single Multicast 43
4.1.2 Application S: Simultaneous Multicast by All Group Members 45

4.2 Experimental Results 45
4.2.1 Cost of Total Ordering 46
4.2.2 Cost of Reliable Multicast 50
4.2.3 Cost of Cryptography 54

Chapter 5 Conclusions and Future work 56
5.1 Conclusions 56
5.2 Future Work 57

References 58

Chapter 1

Introduction

In recent times there has been an increase in the reliance of government and industry on large-scale distributed computer systems to store and process sensitive information. The proliferation of computer networks has caused a number of these critical systems to directly or indirectly become part of large networks. This has resulted in more accessible systems and made it easier for malicious entities to attempt attacks. Also, it is becoming increasingly common to build large-scale systems from commercial off-the-shelf components, which often have known security flaws. This has made large distributed systems particularly vulnerable if they are not able to tolerate subsystem failures due to faults and attacks. The economic and strategic importance of these systems has led to a widespread interest in ensuring their "survivability."

In [EFL+99], *survivability* is defined as a system's capability to fulfill its mission, in a timely manner, in the presence of attacks, failures, and accidents. Survivability research focuses on attack resistance, attack recognition, recovery, and adaptation. Classical computer security efforts, on the other hand, have concentrated on building secure systems whose privilege levels cannot be compromised. In spite of this effort, almost all existing systems have vulnerabilities that have been exploited. Systems have lacked mechanisms to mitigate damages once their security is breached. One was of ensuring survivability involves complementing security mechanisms with "intrusion tolerance."

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. Intrusion tolerance bears a resemblance to fault tolerance, in which the focus has often been building systems out of redundant components in such a way that component failure can be tolerated. The failed components cannot be expected to execute their desired job-functions. Similarly, when a system component is compromised by an adversary, the affected component can no longer be trusted. For that reason, concepts from

fault-tolerant computing may be useful in intrusion-tolerance research. One reason intrusion tolerance is more difficult to achieve is that simplifying assumptions like *fail-stop*[1] failures are not easily justifiable.

Another field that intrusion-tolerance research draws from, for concepts such as encryption, digital signatures, authentication, and secret-sharing, is cryptography. There are many projects currently exploring the use of fault tolerance, cryptography, and computer security concepts for achieving intrusion tolerance. The Intrusion Tolerance by Unpredictable Adaptation (ITUA) [PWL00, CLP+01] project, being undertaken at BBN Technologies and the University of Illinois, is one such effort to increase the survivability of critical distributed systems. The approach ITUA takes is to build middleware that provides intrusion tolerance by combining techniques of replication and unpredictable response. Other intrusion-tolerance efforts include ITTC [WMB99] at Stanford, Enclaves [DSS01] at SRI, ITDOS [MNS+01] at NAI Labs, and MAFTIA [VNC00] which is a joint project of many European research labs.

Intrusion-tolerant systems aim to protect one or more out out of three properties:

Confidentiality Information is disclosed only to authorized users.

Integrity Information being stored in the system and provided to users is correct and has not been altered or destroyed.

Availability Service desired by authorized users is available as advertised.

The projects mentioned above differ significantly in the ways they focus on these three properties. In the IITC project [WMB99], confidentiality and integrity are important goals and secret-sharing is used to ensure them. In MAFTIA [VNC00], applications use several facilities provided by the system to achieve different tolerance goals. An important component of the MAFTIA system is the Timely Trusted Computing Base (TTCB). The TTCB can be relied on to provide correct responses to a small set of queries with time guarantees.

Several projects, including Enclaves, ITDOS, MAFTIA, and ITUA make intrusion-tolerance properties available to applications through middleware. In ITDOS and ITUA, the middleware provides a CORBA[2] interface to applications.

The ITUA and ITDOS projects use replication to achieve survivability. Replication of server objects is a technique used commonly in fault tolerance to achieve redundancy. In intrusion tolerance, replication can help to increase the availability of the system, if the

[1] In this kind of failure, any component upon failure immediately stops all communication with other components.

[2] CORBA is the Common Object Request Broker Architecture [Gro95]. It provides a platform-independent way for applications to interact with each other.

replicas are running on different components of the distributed system. But replication can be harmful to the goal of confidentiality if information to be protected is known to each replica.

A common approach in building intrusion-tolerant systems is to recognize the properties to be provided by the system, model the system and environment at an abstract level, and build an architecture that will provide the stated properties. This has been done for the ITUA system and is described in the following section.

1.1 Intrusion Tolerance by Unpredictable Adaptation

The following sections describe the assumptions ITUA makes about the system being protected, the attacks that it is designed to tolerate, and the proposed architecture. These details of ITUA are also presented in [Wea01].

1.1.1 The ITUA Environment

The system to be protected is divided into a set of non-overlapping "security domains." A *security domain* implements a boundary that attackers have difficulty crossing. A security domain can be a single host or multiple hosts, depending on the configuration. If the hosts do not share administrative privileges, single hosts can be domains. Another example of a specific system domain is a LAN with firewalls separating it from other networks. There should be a minimum of mutual trust between domains, and boundaries should be protected by security techniques like firewalls, authentication and access control.

Every domain is in one of two states: "uninfiltrated" or "infiltrated." A domain is said to be *uninfiltrated* if all processes in the domain are adhering to their protocols and all entities in the domain have their expected privilege levels. A domain is *infiltrated* when some process in it stops adhering to its protocol, or some entity gains a privilege level allowing it to perform operations not allowed at its legal privilege level(s). All domains start in an uninfiltrated state; an intrusion attempt on a domain is said to be successful when the state changes to infiltrated. It is assumed that the attacker is able to control and damage resources freely in an infiltrated domain. For example, a domain that is a Unix host is infiltrated when an attacker has gained root privilege on that host.

Application and system processes can run in each security domain. The set of processes running in a domain is dynamic, i.e., existing processes may be stopped and new processes started. Each process is either "proper" or "corrupt." A *proper* process functions according to its specification; a *corrupt* process is one that is not proper. Components of the system

to be defended are replicated in process groups, with different members of the group on different security domains. A process within a replication group is called a *replica*.

1.1.2 The ITUA Intrusion Model

The *intrusion model* defines a set of possible attacks against the system being defended by ITUA. A type of attack is said to be *considered* if it belongs to this set. The attacks considered by the ITUA intrusion model are the attacks against which the ITUA project is most interested in defending. The intrusion model tries to draw a compromise between listing all possible attacks (and thus making defense almost impossibly difficult) and listing only easily defendable attacks. It is an attempt to consider most feasible attacks and at the same time allow some defense against the worst attacks. It identifies some important, but abstract, features of attacks.

The attacker's goal is to disrupt the normal functioning of the system. That can be achieved by corrupting processes. The attacker will continue to corrupt processes until the attack is repelled or the goal of the attack is reached. An attack can be either partially or completely successful. In a partially successful attack, some measure or measures of quality-of-service (QoS) will be degraded. In a totally successful attack, the system can no longer function as intended.

The attack-primitive in the ITUA model is infiltration of a single domain. The attacker may do this in a variety of ways, such as, by stealing a password, installing a Trojan horse, or exploiting an operating system or protocol flaw. It is assumed that it will always be possible to infiltrate any domain, but that it will be difficult and will take the attacker a non-negligible amount of time. The use of distinct security domains implies that infiltrating one domain does not make it easier to infiltrate another. In practice, if a domain intrusion is detected, the system raises its security level, making it more difficult to break into other domains.

Once a domain has been infiltrated, the attacker can corrupt any of its processes. Corrupting a process may involve simply killing the process, or it maybe more complicated, like causing it to change its behavior. Regardless of how process corruption is performed, it is assumed that this corruption occurs quickly once the domain has been infiltrated. For that reason, no process in an infiltrated domain can be trusted, even if it continues to behave according to its specification.

An important notion in the intrusion model is that of "staged attacks." In a *staged attack*, infiltration of domains occurs in stages, with an increasing number of domains being infiltrated over time; the rate of domain infiltration is bounded. Once a domain is intruded,

there is a possibility, depending on the attacker's actions, that the intrusion will be detected. The ITUA architecture, which is described at a high level in the next section, does not protect against a scenario in which all the security domains are infiltrated at once, since the attacker would then control all the processes at the same time. Another situation not considered is one in which the domains are infiltrated one at a time but the intrusions are not detected until all of the domains become infiltrated, since this is essentially the same as the first scenario.

The definition of security domains guarantees that an attacker must infiltrate a domain before any processes in that domain can be destroyed or their behavior altered. If the various firewalls and security devices are properly configured, it is reasonable to assume that breaking into systems takes time, and that if the security domains are heterogenous, this time is different for different security domains. If we also make the assumption that it is not possible for the attacker to instantaneously determine the domains in which replicas of a particular kind are running, the staged attack premise becomes valid. In addition to the staged attack assumption, another assumption of the ITUA design is that public-key cryptography is secure. ITUA assumes that digital signatures cannot be forged and that private keys cannot be stolen from processes in an uninfiltrated domain.

1.1.3 ITUA Architecture Overview

The basic architecture of ITUA is shown in Figure 1.1. The application makes requests to the middleware to spawn distributed objects with stated intrusion-tolerance requirements. The replication of these application objects is handled transparently by the middleware. The ITUA intrusion-tolerance mechanisms support both in-band and out-of-band adaptation. In the case of in-band adaptation, inter-object communication is intercepted and application behavior is altered. This is managed by the QuO [LBS+98] adaptive middleware. In out-of-band adaptation, intrusion response and recovery actions that involve managing and configuring system resources are taken independent of the application's inter-object interaction. The ITUA project is implementing a decentralized infrastructure to manage this kind of adaptation. It consists of components known as "managers" and "subordinate managers." Subordinate managers are called "subordinates" in the description below.

Each host runs either a manager or a subordinate. Each security domain has one host that runs a manager; the rest of the hosts in that domain run subordinates. All managers belong to a process group called the *manager group*, and all subordinates in a domain and the manager of that domain form a group called a *subordinate group*. The managers make system configuration decisions based on domain-wide information available to them. Each

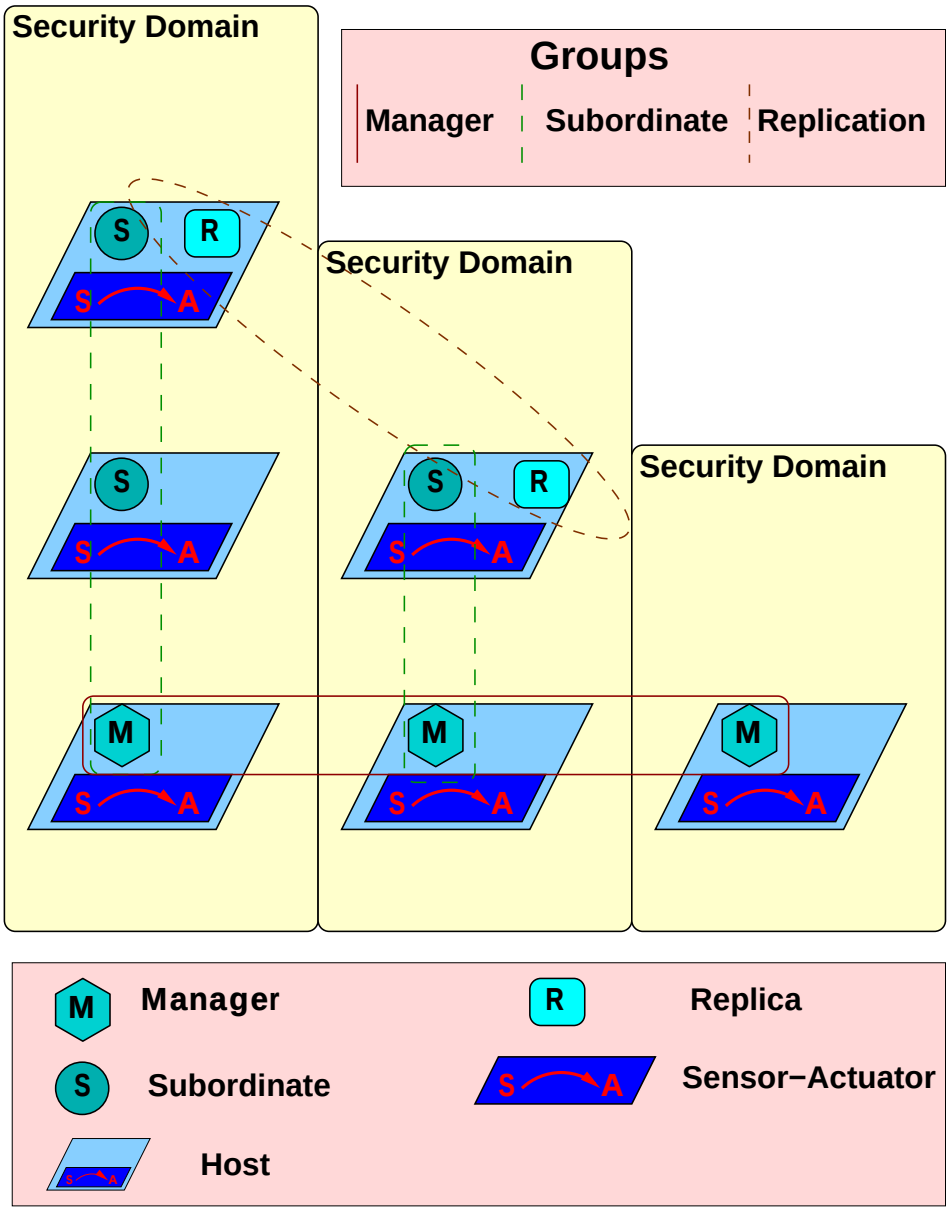


Figure 1.1: ITUA architecture

manager also performs all host-specific duties of a subordinate for its host. A subordinate's two principal responsibilities are *security advising* and *replication management*. The actions and decisions taken as part of these are always local, i.e. they involve resources associated with the host on which the subordinate runs and the host's network interfaces. However, in some cases (for instance, changing the tolerance mode of a replication group), the impact of a subordinate's actions may ripple through multiple domains, in which case multiple managers and subordinates cooperate. In the security advisor role, a subordinate makes use of multiple local *sensor-actuator* loops to

- collect information about potential intrusions and anomalous events,
- perform a quick "knee-jerk" local reaction to the observed event, and
- provide the security domain manager with host-specific information.

The application objects protected by ITUA are replicated by the middleware and distributed across security domains. Decisions about placement and starting/stopping of particular replicas are made by the managers in a distributed manner. In the replication management role, subordinates are responsible for starting or stopping replicas of application components on a particular host. A subordinate may determine, in its security advisor role, that the local host is under attack, and then consequently modify, in its replication management role, the tolerance mode of some groups that have replicas on that local host. However, note that even though that is a local decision, it affects other hosts in other domains too. Changing the tolerance mode of a replication group involves agreement of the members of that group, which in turn involves cooperation of multiple managers and subordinates. The local action merely initiates that process.

In addition to performing these management functions, ITUA also provides in-line support for (CORBA) remote method invocations made by distributed applications. It uses an "ITUA gateway" to do this. The ITUA gateway uses replication protocols, and an intrusion-tolerant group communication system, in order to make remote method invocations intrusion tolerant. The design of the ITUA gateway is shown in Figure 1.2.

The gateway has several functions: translating between object-level and process-level communication, providing an infrastructure for implementing various replication and voting schemes, and detecting and reporting faults to the manager or subordinate on the host. The gateway intercepts standard IIOP[3] messages generated at each CORBA object. It is the gateway's responsibility to transparently manage the communication of this message to the

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

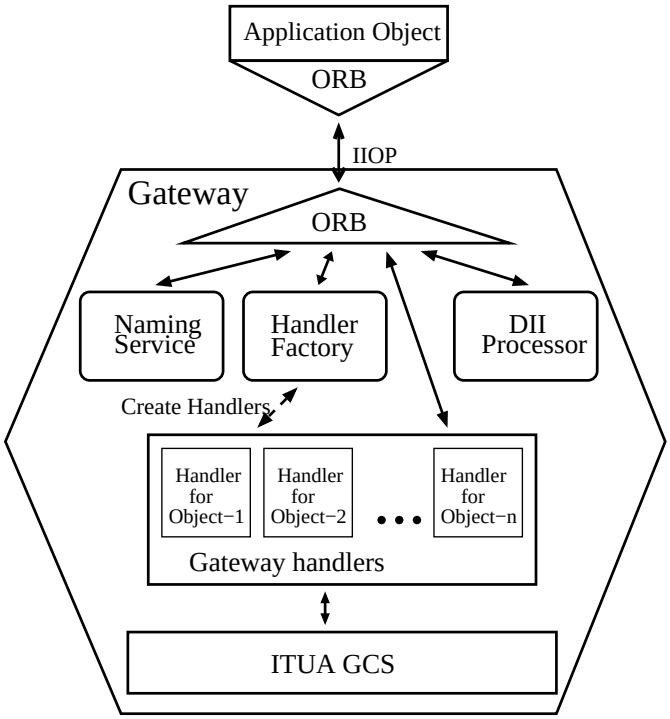


Figure 1.2: ITUA Gateway

recipient replication group and to allow only a single response to reach the calling object. The interception and sending of IIOP messages is done using the Dynamic Invocation Interface (DII) processor. The naming service allows the gateway to map names to CORBA object ID strings. The handler being used for a particular object decides how a single response will be passed to the object based on the multiple responses received by the gateway from a replication group. The choice of which handler to use for a particular object depends on the intrusion-tolerance requirements specified by the object.

There are many communicating process groups in the ITUA architecture, including replication groups, the manager group, and subordinate groups. These groups have several functions in common, including maintaining group membership information, point-to-point message delivery and reliable message delivery. Clearly, these common functions suggest the creation of a lower level abstraction that can provide these properties. To provide this functionality a new intrusion-tolerant group communication system has been developed for ITUA which is part of the gateway (Figure 1.2) and is used to form the manager group and subordinate groups.

Group communication systems are not new; several have been built in the past, and provide differing properties to system builders. In the next section, we review research

efforts in this direction that are related to our work.

1.2 Previous GCS Research

Over the past two decades there have been quite a few efforts to develop effective group communication systems. Early systems, such as the V kernel [CZ85], showed the feasibility and effectiveness of process groups. The ISIS system [Bir93, BvR94] pioneered the concept of achieving fault tolerance through process groups. Several other systems based on the concept of process groups exist today including Totem [MMSA+96] and SecureRing [KMMS98] from UCSB, Horus [vRBM96] from Cornell, and Transis [DM96] from the Hebrew University. There are now commercial projects, such as Phoenix at IBM and NT Clusters at Microsoft that are based on the concept of process groups. Projects like AQuA [Ren01, CRS+98] at UIUC and Eternal [MMSN99] at UCSB have used GCSs to develop higher-level abstractions to support reliable distributed computing. Some group communication projects that are particularly relevant to the research described in this thesis are described below.

1.2.1 Practical Byzantine Fault Tolerance

The PBFT protocol [CL99b] was developed by Castro and Liskov at MIT. The protocol's model differentiates between client and server processes. Clients make requests on a group of servers, which then jointly come up with a reply and individually send responses to the requesting client. The client verifies that enough replies are consistent and continues. The protocol does not rely on synchrony assumptions for safety (proved in [CL99a]), but does make some synchrony assumptions for liveness.

This protocol does not provide mechanisms for group members to join and leave the server process groups. Each process group goes through a series of views, and in each view, one process serves as the primary and the other servers are backups. A new view is installed when at least two-thirds of the servers agree that the primary is faulty. This protocol can function even when (at most) one-third of the servers exhibit Byzantine behavior.

1.2.2 Rampart

The Rampart toolkit [Rei95, Rei94a, Rei94b] was developed to aid in developing high-integrity distributed services. Processes join groups to communicate with each other. The underlying system has to provide a completely connected point-to-point network. Reiter developed a group membership protocol [Rei94b], along with reliable and atomic group mul-

ticast [Rei94a] which continue to function in the face of Byzantine faults, given that less than one-third of the group members are faulty. The protocol uses public-key cryptography for ensuring reliable message delivery. Rampart has been used to implement intrusion-tolerant applications (e.g. [RF96]).

1.2.3 Ensemble

Ensemble [Hay98, Hay01] is a flexible group communication system which is based on the concept of micro-protocols. A large set of micro-protocols has been implemented; Ensemble chooses the appropriate subset and creates a *protocol stack* based on the properties desired by the application. Many security features [RBD01] have been added to Ensemble but only crash faults of the group members are tolerated.

The flexibility of a configurable stack makes it easy to implement a new protocol in terms of micro-protocols and quickly prototype it in the Ensemble framework. Unfortunately, Ensemble is implemented in the ML programming language, making it inaccessible to most computer scientists unless they are willing to learn a new language. Mark Hayden, one of the original developers of the Ensemble system has created C-Ensemble, which is an implementation in the C language of the most important micro-protocols of Ensemble.

1.2.4 SecureRing

The SecureRing [KMMS98] GCS consists of a reliable delivery protocol, a group membership protocol, and a Byzantine fault detector. The group membership protocol makes use of the Byzantine fault detector to ensure that corrupt members are thrown out of the group and proper members have a consistent view of group membership. The message delivery protocol uses signed tokens, which are multicast and contain digests of messages that have been multicast by the holder of the token. Multiple application-level messages are collected to form a packet, which is sent out as a normal message. A normal message can be delivered only after the token verifying its authenticity is received. SecureRing's developers claim that signing tokens with multiple message digests makes their protocols more efficient than protocols where all messages need to be signed.

1.3 Research Contributions and Thesis Organization

As outlined in the ITUA architecture overview, an important architectural component of ITUA is an intrusion-tolerant group communication system. In the research described in

this thesis, message multicast protocols were designed for the ITUA intrusion-tolerant GCS. In particular, two GCS protocols have been developed. The first protocol, called the *reliable multicast protocol* guarantees that all messages multicast by a proper process will reach all other proper members unchanged, given that certain conditions are met. The second protocol, called the *total ordering protocol*, guarantees that all proper processes receive multicast messages in the same sequence. Chapter 2 describes the properties provided by these protocols more formally and states the conditions that have to be met for the protocols to work correctly, provides detailed algorithmic descriptions of the protocols, and argues why the protocols provide the stated properties.

These protocols have also been implemented in a layered protocol stack. Chapter 3 provides details about the implementation. It also describes the framework in which the protocols have been implemented, and the changes that were made to the framework to to make it possible to add these protocols. These infrastructure changes include the interfacing of a cryptographic library with C-Ensemble.

Important performance characteristics of the intrusion-tolerant GCS have also been measured. These performance measurements give an idea of the performance overhead involved in using the GCS to build systems like the ITUA middleware. More importantly, many techniques used in the multicast protocols of the ITUA GCS have also been used in other attempts at creating intrusion-tolerant systems. So the performance measurements can be used to gain an insight into the most expensive parts of an intrusion-tolerant GCS and motivate research for optimizing them or looking for alternatives. The results of the performance measurements are described in Chapter 4.

Chapter 5 outlines the conclusions that we have arrived at after implementing the protocols and making the performance measures. It also provides ideas for extending this work.

Chapter 2

New Protocols in the ITUA GCS

The architecture of the ITUA system described in the first chapter of this thesis suggests that it is possible to achieve intrusion tolerance by using multiple process groups, in which the group members are distributed across multiple security domains. These process groups need to maintain consistent information about group membership; faulty members need to be removed and new members need to be added to the groups. As mentioned in the previous chapter, this is a common problem in various application domains, and group communication systems have been developed to provide a process group abstraction to applications. GCSs can be used by applications to provide various properties, such as group membership, reliable message delivery, and atomic message delivery. The GCS used for a particular application should fit its requirements. The GCS required by ITUA needs to provide consistent group membership in the presence of intrusions that can cause processes to behave maliciously.

Another property required by ITUA is reliable delivery of multicast messages. In the manager group, the subordinate groups, and the replication groups, most of the information that is shared needs to reach all processes. For example, a manager might want to inform the manager group about the infiltration of a domain; all managers would need to receive this message. Furthermore, as described in Section 1.1.2, we assume that some security domains can be infiltrated without detection, and that processes in such domains can be corrupted. These processes can then send messages with arbitrary contents to other processes on the network. That implies that members of a group cannot blindly trust a message they get from the network. Multicast messages need to be delivered to all proper processes in spite of faulty behavior by some corrupt members. This problem is common to all process groups of ITUA, and suggests that an intrusion-tolerant reliable multicast primitive that guarantees reliable delivery should be made available to the processes by the GCS.

The ITUA middleware creates replication groups of application objects to make them tolerant to intrusions. Most application objects have some state information that changes

as they process information. Often, the final state after a set of incoming messages has been processed depends on the order in which messages were received by the object. Since we want the members of a replication group to have the same state, it is necessary for ITUA to ensure that messages are delivered to all of the replicas in the same order. This implies that, at least for the replication groups, the multicast primitive should ensure that messages are delivered in the same total order at all proper members. Again, a good way of providing this property is through the GCS used by ITUA.

Thus, ITUA needs a GCS that provides group membership, reliable delivery, and total ordering of multicast messages in the presence of corrupt group members. As no freely available GCS (that we know of) provides these properties, a new intrusion-tolerant GCS has been designed for ITUA. The most important protocols of this GCS are for group membership and message multicast. The group membership protocol is described in [Ram]. This thesis focuses on the reliable multicast and total ordering protocols that have been developed for ITUA.

The rest of this chapter describes the reliable multicast and total ordering protocols in detail. The system model used by these protocols is described in the next section. It was developed with the ITUA architecture in mind, but is more general and the developed protocols can be used in other scenarios that fit this model.

2.1 System Model and Assumptions

The communication model we consider consists of multiple hosts running several processes communicating over an unreliable asynchronous network. The group communication system protocols are concerned with one set of processes that wish to be in a communicating group. The group membership protocol installs a series of views, V^0, V^1, ..., each of which is a set of process identifiers of processes that are members of the view. The processes in a single view V have *ranks* from 0 to $|V| - 1$, and are denoted by p_0, p_1, ..., $p_{|V|-1}$. Each proper process conforms to the protocols; corrupt processes can exhibit arbitrary behavior, but are computationally bound. The maximum number of faults that can be tolerated is $t = \lfloor (|V| - 1)/3 \rfloor$.

The group membership protocol provides the interfaces *suspect(process_rank i, reason r)* and *faulty(process_rank i, reason r)* which allow the other protocols to inform it about suspected and faulty processes. The messaging protocols rely on the *fault detector* [KMMS97] implemented by the group membership protocol to circumvent the impossibility [FLP85] of consensus in an asynchronous environment. The fault detector makes some timing as-

sumptions about how long certain messages take to arrive. Processes that are causing the protocols to slow down are considered faulty and are removed even if they don't exhibit any other faults.

The underlying infrastructure provides a *cast_unrel(message m)* method, which the protocols can use to send unreliable multicasts to all group members. There is also a *send_rel(message m, process_rank i)* method, which can be used to send a message reliably to a peer. Messages, in addition to carrying their payload, can be "tagged." A message is tagged via a call to the function *tag(message m, key k, value v)*. The receiver of a message can extract these tags using the function *get_tag(message m, key k)*. Since tags are attached to messages, the content of the message envelope changes when a tag is added.

Cryptographic hashing functions are used to create message digests. The hashing functions take the message envelope as data. If two messages have the same contents but different tags, and their digest is created, the hash function will see different input data. If we assume that the message digest function is collision-resistant (such as SHA-1 [NIS95]), then the probability of the message digests of two such messages being the same is negligible. It is computationally infeasible for a corrupt process to create two message envelopes that have the same message digest.

A public-key cryptosystem (like RSA [RSA78]) is used for message authentication. Each process possesses a private key, known only to itself. The message multicast protocols require that all processes also possess the public keys required to authenticate other processes. It is assumed that private keys cannot be stolen from uncorrupted processes. It is also assumed that an adversary cannot forge digital signatures.

2.2 Reliable Multicast

Any reliable multicast protocol guaranteeing reliable delivery of messages to all proper group members, which operates according to the assumptions in Section 2.1, must deal with two main issues. The first issue is that of an asynchronous unreliable network where messages can be lost, reordered and delayed. This is usually taken care of via message buffering, sequence numbers and positive and negative ACKs. We have also taken that approach.

The second issue is that of guaranteeing that a multicast message is delivered correctly (without a change in its contents) across all proper processes, even in the presence of corrupt senders. To ensure that all proper processes deliver the same message, the processes have to come to a consensus about the contents of each message that is to be delivered. There are two approaches [LSP82] to solving the consensus problem. In the first approach, the processes

communicate in rounds, telling each other what they have heard in the previous round. Each process-to-process communication is assumed to take place over a secure channel. This *oral messages* approach requires at least $3f + 1$ processes to tolerate f arbitrary faults. The second approach is the *signed messages* approach, in which the messages being exchanged between processes have non-repudiable signatures that can be forwarded and verified by processes. In that approach solutions are possible with only $f + 2$ processes [LSP82], but multiple rounds are needed. The number of rounds needed can be kept small if we assume that to tolerate f faults, $3f + 1$ members are available.

We, like Reiter [Rei94a], have taken the signed message approach to solving the reliable multicast problem, for two reasons. First, as cryptographic algorithms and mechanisms improve, the process of digital signatures will become increasingly cost-effective as compared to multiple rounds of messages. Even with oral messages, the common way of ensuring secure point-to-point channels is through the use of cryptography. Second, assuming that public-key cryptography is secure, the signed message approach leads to much cleaner, easy-to-understand protocols with lesser chances of subtle errors in the implementation and the protocol itself.

The reliable multicast protocol described in the next section has the properties described below. They are similar to the properties provided by SecureRing [KMMS98] and Rampart [Rei94a].

Integrity For any message m and process p, a proper 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 proper throughout a view and delivers m in that view, then all processes that are proper throughout that view deliver m in the same view.

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

The conditions under which these properties are provided, and informal arguments about how the reliable multicast protocol provides these properties, are presented in Section 2.2.2.

2.2.1 Reliable Multicast Protocol Description

Figure 2.1 gives a high-level description of the sequence of events for a single message multicast when no process is misbehaving. When a message needs to be transmitted, the sender buffers a copy of the message, creates a signed digest for the message and sends the digest to the group. On receiving such a message the multicast protocol creates a signed reply to

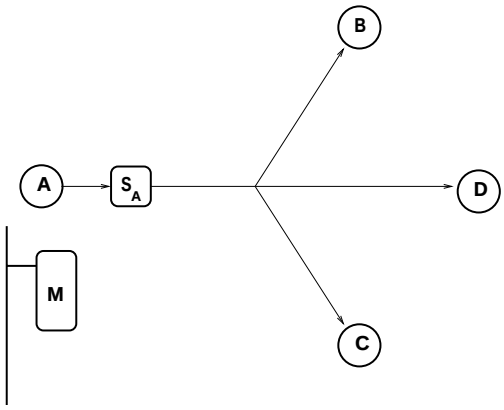
it and sends the reply back to the sender of the digest. The digest's sender collects messages from $2f + 1$ replies to its digest and then sends out the actual message with the $2f + 1$ signatures attached. On receiving such an authenticated message, the protocol checks the validity of the $2f + 1$ attached signatures and accepts the message. At this level the protocol is similar to the echo protocol of [Rei94a]. A detailed description of the phases in a single message multicast is given below.

The first phase begins when the user multicasts a message. This is done by calling the function, *cast_rel(message m)*, provided by the reliable multicast protocol. The protocol's actions when this function is called are shown in Figure 2.2. The message to be delivered is first buffered by the sender, and a sequence number is assigned to it. A proper process assigns consecutive sequence numbers to messages in the order in which calls to *cast_rel* are made. The sender then unreliably multicasts a "notice" for this message to the group.

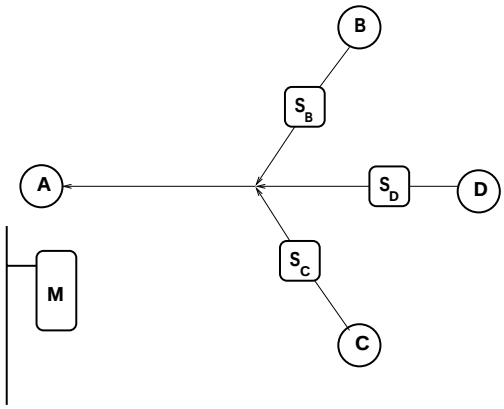
A *notice* is a signed (using the sender's private key) version of a data structure containing the digest, the sender's rank, the current view number, and the sequence number of the message. The message digest of m is denoted by $\delta(m)$, and the signed notice is denoted by $\nu_t(m,\ r,\ x,\ s)$, where the current view is V^x, t is the signer's rank, and s is the sequence number assigned to m by the sender r. Each process has $|V|$ *msg_store*s, one per group member, to store messages from the member. Similarly, processes have $|V|$ *notice_store*s to store notices. All process store notices from p_i in the $notice\_store_i$ until the corresponding message is sent or received. Each slot in the *notice_store* for the sender has a $resend\_counter$. This counter is incremented whenever a new notice is sent, and compared against a constant *resend_wait* to decide when to resend notices (see Figure 2.2).

When a multicast notice is received from the network, the GCS infrastructure calls the *received_notice(notice n, process_rank r)* function of the reliable multicast protocol. Figure 2.3 gives a description of this function. On receiving the notice $\nu_r(m,\ x,\ r,\ s)$, the receiver p_t checks the notice's validity and checks whether it has received another notice for the same sequence number with a different digest. It then responds with a *notice_reply* which is $\nu_t(m,\ x,\ r,\ s)$. The *verify_notice* procedure, used in line 3 in Figure 2.3, decrypts the encrypted portion of the notice and checks that the view, sender, and sequence number information in the notice are correct.

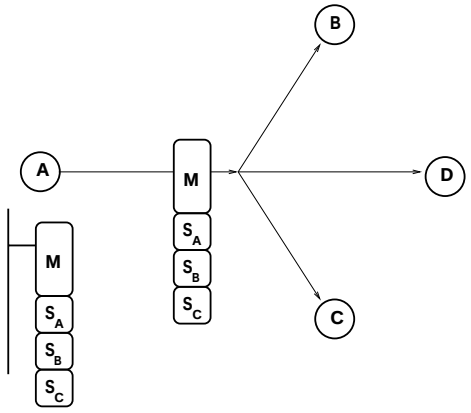
When the *notice_reply* (from p_r) is received from the network, the GCS infrastructure calls the *received_notice_reply(message m, process_rank r)*. The pseudo-code for this function is shown in Figure 2.4. The protocol obtains the message number s from the *notice_reply*, and if it is the first time this reply has been received by the sender of the notice, the protocol verifies the reply and stores it in $notice\_reply\_collect[s][i]$. *notice_reply_collect* is a collection of notice replies maintained for each message that the sender has multicast. When the



(a) Process A sends a signed digest, S_A, for message M



(b) B, C, and D respond with ACKS containing their signatures



(c) A now sends actual message with $2f + 1$ signatures attached

Figure 2.1: Protocol for a single *reliable* message send

```
cast_rel(message m) IN VIEW V^x
 1: Assign next available sequence number s to m
 2: Store m in the msg_store_{my_rank}[s]
 3: Create notice_{my_rank}(m) = ν_{my_rank}(m, x, my_rank, s)
 4: tag(notice_{my_rank}(m), 'seqno', s)
 5: tag(notice_{my_rank}(m), 'view', x)
 6: cast_unrel(notice_{my_rank}(m))
 7: Put notice_{my_rank}(m) in notice_store_{my_rank}[s]
 8: Set resend_counter of this notice to 0
 9: for all previous notices, n_old in notice_store_{my_rank} do
10:    if (number of replies for n_old) < (2f + 1) then
11:       Increment resend_counter of n_old by 1
12:       if  resend_counter = resend_wait for n_old then
13:          cast_unrel(n_old)
14:          Set resend_counter of n_old to 0.
15:       end if
16:    end if
17: end for
```

Figure 2.2: The *cast_rel* function

number of distinct *notice_replies* for a particular message reaches $2f + 1$ (including the sender's original notice), the sender knows that a majority of proper processes now have a copy of the notice originally sent by it. The process then creates an *endorsed* message, which is the original message along with $2f + 1$ *notice_replies* from distinct group members. The *msgs_delivered* array used in lines 24 and 25 of Figure 2.4 is initialized to all zeros at the time of view installation. It is used by the reliable multicast protocol at the time of view change; this is described later in this section.

The final phase in the transfer of a single message is conceptually simple. It is shown in Figure 2.5. When the endorsed message containing $2f + 1$ notice replies is received, the *received_rel_msg* function is called. In that function, the process has to verify the contents of the notice replies and check that the message digest in the notice replies corresponds to the actual message. Once this verification succeeds, the message is accepted and stored in the *msg_store* corresponding to the sender of the message.

The preceding description has shown how a single message is reliably multicast to the group. The rest of this section describes how lost messages and view changes are dealt with.

In case a notice is lost the *cast_rel* function resends it, as is shown in Figure 2.2. Since

received_notice(notice n, process_rank r) IN VIEW V^x

1: $s \Leftarrow get\_tag(n,$ 'seqno')
2: $y \Leftarrow get\_tag(n,$ 'view')
3: **if** ($verify\_notice(n,\ r)$ fails) **then**
4: Stop
5: **end if**
6: $n_{old} \Leftarrow notice\_store_r[s]$
7: **if** n_{old} exists **then**
8: **if** $n_{old} \neq m$ **then**
9: $faulty\_process(r,\ \{$'mutant message', $n, n_{old}\})$
10: **end if**
11: Stop
12: **end if**
13: Store n in $notice\_store_r[s]$
14: Create a $m_{send} = notice\_reply_{my\_rank}(m) = \nu_{my\_rank}(m,\ x,\ r,\ s)$ using the digest in n
15: $tag(m_{send},$ 'view', $x)$
16: $tag(m_{send},$ 'seqno', $s)$
17: $send\_rel(m_{send},\ r)$

Figure 2.3: The *received_notice* function

notice replies are sent using the reliable *send_rel* function, they cannot be lost, and do not need to be resent. Therefore, the only kind of scenario that the negative acknowledgement (NAK) scheme has to take care of is the loss of endorsed messages.

To keep track of sequence numbers that need to be NAKed, messages are stored in *msg_store*s, which are indexed by message sequence number. When a process receives a new endorsed message, it checks whether all messages with lower sequence numbers have been previously received. The receiver sends NAKs for all previous messages that have not been received. The NAK message is sent to the sender only, unless the sender has already been reported to have failed (by the group membership protocol). In case the process has failed, the request is sent to the whole group; any group member that has the message forwards it to the peer who sent the NAK. There is a timing mechanism that ensures that NAKs for the same message are not sent out too frequently.

When the group membership protocol tries to install a new view, it gives a list of processes that are being removed. The members of the current view that are also members of the next view begin a consensus round to decide which messages have been delivered by each of them

received_notice_reply(message m, process_rank r) IN VIEW V^x

1: $s \Leftarrow get\_tag(m,$ 'view')
2: $y \Leftarrow get\_tag(m,$ 'seqno')
3: $\delta \Leftarrow$(digest out of the notice in $notice\_store_{my\_rank}[s]$
4: **if** ($notice\_store_{my\_rank}[s]$ is empty) **or**($verify\_notice\_reply(\delta, m,\ r)$ fails) **then**
5: Stop
6: **end if**
7: **if** (number of entries in $notice\_reply\_store[s]$ is $2f + 1$) **or**(r has already sent this reply) **then**
8: Stop
9: **end if**
10: Store m in $notice\_reply\_collect[s][r]$
11: **if** (number of notice_replies stored in $notice\_reply\_store[s]$)$= (2f)$ **then**
12: Add $notice\_store_{my\_rank}[s]$ to $notice\_reply\_store[s]$
13: Get message m_{send} from $message\_store_{my\_rank}[s]$
14: **for all** notice_replies ν stored in $notice\_reply\_store[s]$ **do**
15: append ν to m_{send}
16: **end for**
17: $tag(m_{send},$ 'seqno', $s)$
18: $tag(m_{send},$ 'view', $x)$
19: Store the changed m_{send} in $message\_store_{my\_rank}[s]$
20: Delete the entry $notice\_reply\_store[s]$
21: Delete the entry $notics\_store_{my\_rank}[s]$
22: $cast\_unrel(m_{send})$
23: $i \Leftarrow s$
24: **while** $i = msgs\_delivered_{my\_rank} + 1$ **do**
25: $msgs\_delivered_{my\_rank} \Leftarrow i$
26: Increment i by 1
27: **end while**
28: **end if**

Figure 2.4: The *received_notice_reply* function

received_rel_msg(message m, process_rank r) IN VIEW V^x

1: $s \Leftarrow get\_tag(m,$ 'view')
2: $y \Leftarrow get\_tag(m,$ 'seqno')
3: **if** $(x \neq y)$ **or**(message numbered s has already been received) **then**
4: Stop
5: **end if**
6: **if** ($notice\_store_r[s]$ exists) and (original notice does not match m) **then**
7: $faulty\_process(r,$ {'mutant message', m})
8: Stop
9: **end if**
10: **if** notice_reply of sender (rank r) is not included **then**
11: Stop
12: **end if**
13: **if** (number of *notice_replies*) $\neq (2f + 1)$ **then**
14: $faulty\_process(r,$ {'incorrect proof', m})
15: Stop
16: **end if**
17: **for all** notice_replies ν_j (from member j) validating m **do**
18: **if** $(j \notin V^x)$ **or**(ν_j is invalid) **then**
19: $faulty\_process(r,$ {'incorrect proof', m})
20: Stop
21: **end if**
22: **end for**
23: Store m in msg_store_r
24: **if** $s = msgs\_delivered_r + 1$ **then**
25: **while** $msg\_store_r[msgs\_delivered_r + 1]$ has a message **do**
26: $rel\_deliver(m,\ r)$
27: Increment $msgs\_delivered_r$ by 1
28: **end while**
29: **end if**

Figure 2.5: The *received_rel_msg* function

at this point. Each process unreliably multicasts a signed message with its *msgs_delivered* array. After that, there is a second round in which every process multicasts a message containing all signed copies from the previous round. Finally, for each entry of the array, the process that claims to have the highest value of that entry has to multicast all messages that the process with the lowest value (of that entry) claims to not to have received. Each of these rounds has timeouts associated with it, and members that obstruct progress are reported as suspects to the group membership protocol.

2.2.2 Reliable Multicast Protocol Properties

This section makes informal arguments that the reliable multicast protocol presented above provides the properties of *integrity*, *agreement*, and *per-sender FIFO* described in Section 2.2. We believe that these arguments can be used to formulate formal proofs of correctness of our protocol. Because of the complexity of formally proving the correctness distributed protocols in an asynchronous environment, such proofs are beyond the scope of this thesis.

The arguments below assume that in any view V^x, if f faults are being tolerated, then $|V^x| = 3f + 1$. Another assumption is that the fault detector implemented by the group membership protocol works correctly and that all corrupt group members are eventually detected and removed from a view. We express the properties as one or more claims and argue that the claims are true.

Integrity

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

- *Each proper process delivers m from p at most once*

  Each message that is multicast has a process rank and a sequence number[1]. A proper process delivers a message with a particular sequence number only once even if it receives the message multiple times (line 3 of Figure 2.5). It should be noted that two messages with the same contents but different sequence numbers are considered distinct.

- *A proper process delivers m from p only if p multicast m, given p is proper*

  A proper process accepts a multicast message only if a *notice_reply* of the sender is attached to the message (lines 10–12 of Figure 2.5). A *notice_reply* is digitally signed

[1]We assume that the processes do not exhaust the space of valid sequence numbers during a run of the protocol.

and has the digest of the message. If p is proper, no other process can create a valid *notice_reply* for m that appears to come from p without p's private key. So, p must be the process that multicast m.

Agreement

Definition: If process p is proper throughout a view and delivers m in that view, then all processes that are proper throughout that view deliver m in the same view.

- *If two proper processes deliver a message from p with the same sequence number s, the contents of the messages are the same*

  If a message m with sequence number s from p is delivered at a proper process q, it must have been endorsed with signed *notice_replies* from $2f + 1$ group members (lines 13–22 of Figure 2.5). Since only f group members can be faulty in a view, at least $f + 1$ proper processes must have sent *notice_replies* to p for sequence number s. A proper process will send only one *notice_reply* to a sender for a particular sequence number (lines 7–12 of Figure 2.3).

  Suppose a message m' from p with the same sequence number s (from the last paragraph) is delivered at some other proper process. Both messages must have $2f + 1$ *notice_replies*. Since there are $3f + 1$ members, at least $f + 1$ must have sent *notice_replies* for both m and m'. If m' is distinct from m then at least 1 proper member sent *notice_replies* for two distinct messages from p with the same sequence number. That is not possible. Therefore, m and m' must be the same message.

- *All processes that are proper throughout a particular view deliver the same number of messages from each sender in that view*

  The reliable multicast protocol ensures that this is true by the exchange of *msgs_delivered* arrays and forwarding of messages at the time of view change. Each process finds out the number of messages from each sender that have been delivered at each group member. Processes then forward endorsed versions of messages that they have delivered but other processes have not. At the end of this exchange, each proper process will have delivered an equal number of messages per sender. Due to the time-outs enforced during the exchange, processes that make false claims are removed from the view.

Messages are delivered in the order of their sequence numbers (per sender). That implies that if the number of messages from a sender that have been delivered at two receivers is equal, the messages delivered had the same sequence numbers.

Per-sender FIFO

Definition If p and q are proper and q delivers m_1 and then m_2 from p, p must have multicast m_1 and m_2 in that order.

Since p is proper it assigns sequence numbers in order to messages that it sends out. Since q is proper, it delivers messages from p in the order in which they are received. These two facts together imply the per-sender FIFO property.

2.3 Total Ordering Protocol

As mentioned before, total ordering is an important property required by replication groups. Other groups, like the ITUA manager group, can also use this property to simplify their operation. For example, if several managers multicast messages asking for configuration changes, the managers would need to come to a consensus about the order in which these changes need to be made. Instead, the manager group can choose to use the total ordering property and make the configuration changes in the order in which the messages are received.

Total ordering can be achieved in a variety of ways. In Rampart [Rei94a], a sequencer-based approach is used in which a special group member called the sequencer decides the order in which messages will be delivered. Totem [MMSA+96], on the other hand, uses a token-based approach in which ordering decisions are made based on the order in which the different processes hold the token.

The total ordering protocol designed for the ITUA GCS achieves total order by assigning sets of global sequence numbers to the group members at the time of view installation. Individual sequence numbers are assigned by processes to messages they multicast, and messages are delivered in the order of the sequence numbers by all processes. This scheme can be seen as an example of a "born-order" protocol [Bir96] for total ordering, in which the messages contain information about the order in which they should be delivered. Details of the protocol and the properties provided by it are described in this section.

The total ordering protocol assumes that the services of a reliable multicast protocol (like our protocol from the previous section), that guarantees FIFO ordering (per sender) are available. The total ordering protocol provides the following property:

Order If proper processors p and q both deliver m_1 and m_2, then they deliver them in the same order.

An informal argument that explains how the total ordering protocol provides this property is presented in Section 2.3.4.

2.3.1 High-level Protocol Description

Intuitively, the total ordering protocol functions by assigning globally unique sequence numbers to all messages transmitted in a view. Sequence numbers in each view belong to a totally ordered, countably infinite set[2]. Messages are generated asynchronously by the group members. The protocol ensures that sequence numbers assigned to messages by different processes are globally unique by partitioning the set of all possible sequence numbers and assigning a partition to each process. Each proper process sends messages with these sequence numbers, using one sequence number per message in increasing order. The total ordering protocol at each process delivers incoming messages in the order of sequence numbers; no sequence numbers are missed[3] or repeated. Therefore, for a message with sequence number s to be delivered, all messages with sequence numbers less than s must be delivered first.

Sequence numbers are partitioned across the processes via an association of each process p_i, at view installation time, with an initial sequence number seq_orig_i and a sequence number generating function gf_i. The gf_is and seq_orig_is for each member process in a particular view are known to all group members. Each function is monotonically increasing, and $f_i(x) > x$. Each process p_i generates a series of sequence numbers, that starts with seq_orig_i; each subsequent sequence number is generated through the application of gf_i to the previously generated sequence number. Each process can generate the sets of sequence numbers for all processes. The sets of sequence numbers, $S_i = \{seq\_orig_i,\ gf_i(seq\_orig_i),\ gf_i(gf_i(seq\_orig_i)),\ \ldots\}$, generated by a proper process have the following properties:

1. The sets are pair-wise disjoint, i.e. $i \neq j \Rightarrow S_i \cap S_j = \phi$
2. The sets taken together contain all possible sequence numbers, i.e. $\bigcup_{i=1}^{n}(S_i) = S$, where S is the set of all sequence numbers.

The efficiency of this protocol depends on the relationship between the actual message traffic generated by the group members and the ordering forced by the sequence numbers assigned to them. If there is a close match between the two, this protocol will be more efficient than protocols that need sequencers or depend on some form of distributed consensus, because the extra step of deciding the order is avoided. It is reasonable to expect good performance in the case of replication groups, since they have a predictable pattern of message traffic under normal circumstances.

The protocol can be held up by a process that does not send a message with a particular sequence number. That problem is avoided by forcing group members to transmit protocol-

[2]These sequence numbers are never actually transmitted over the network, so we don't have to worry about encoding them in a finite-sized field.

[3]except for *null* messages, this is explained later in this section.

level messages with no payload (*null* messages) if they don't have any other messages to send. All processes monitor the progress of other processes. If some process is not sending any messages, and thus is holding up the progress of the protocol, this fact is reported to the fault detector implemented by the group membership protocol of the GCS.

2.3.2 Detailed Protocol Description

This section provides a more detailed description of the total ordering protocol. The protocol has to maintain some state information at each process. We begin by giving details about how this information is stored.

Each process $p \in V^x$ maintains a set of $|V^x|$ queues, one for each group member, to hold messages from the member. Each queue q_i (for process p_i) has an associated variable $first\_seq_i$. $first\_seq_i$ denotes the (global) sequence number of the first message in q_i. If q_i is empty, $first\_seq_i$ is the sequence number of the next message to be received from p_i. p also has a *send_queue*, initially empty, to buffer messages in case p has been sending out too many messages. Another variable that the protocol keeps track of is *next_seq*, the next sequence number that has to be delivered.

On a view installation, the protocol is initialized with the following parameters:

- *my_rank:* rank of this process.
- gf_is*:* set of sequence number generating functions (gf_i corresponds to p_i).
- seq_orig_is*:* set of starting sequence numbers.

The protocol also has a list of pre-defined constants:

- *q_len_lo:* largest size of q_{my_rank} before the protocol begins buffering outgoing messages.
- *q_len_hi:* largest queue size that is counted towards the calculation of total queue length to avoid suspicion of proper processes caused by corrupt processes that are sending too many messages.
- *pending_msgs_lo:* value of total queue length when a *null* message is sent by the process if q_{my_rank} is empty.
- *pending_msgs_hi:* value of total queue length when suspicions are generated for processes that correspond to empty queues. It is up to the group membership service to decide whether the member is faulty and remove it, if it is.

The protocol begins by initializing the set of queues and variables. The queues are initialized to empty; each $first\_seq_i$ is initialized to seq_orig_i. $next\_seq$ is initialized to the lowest amongst the seq_orig_is.

Figure 2.6 describes the processing done by the protocol when a message is to be multicast. The process first checks if it has been sending too many messages, and stores the outgoing message in *send_queue* if it has. If the message can be sent immediately it is stored in q_{my_rank} and then multicast reliably. The *check_queue* function (described later in this section) is used to check if any new messages can be delivered.

```
cast_total(message m) IN VIEW V^x
 1: if size(q_my_rank) > q_len_hi then
 2:    buffer message in send_queue.
 3:    Stop.
 4: end if
 5: tag(m, 'type', 'Single')
 6: Enqueue message in q_my_rank.
 7: Call cast_rel(m).
 8: if size(q_my_rank) = 1 then
 9:    Call check_queue(my_rank).
10: end if
```

Figure 2.6: The *cast_total* function

When the reliable multicast protocol delivers a message, the *rel_deliver* function shown in Figure 2.7 is called. The total ordering protocol is greatly simplified by the reliable, per-sender FIFO delivery guaranteed to it by the reliable multicast protocol. Messages from p_i have to be queued in q_i in the order in which they are received. Sequence numbers are calculated at each process. Since the set of generating functions at each group member in a particular view is the same, each message gets assigned the same sequence number (see Section 2.3.4 for a more detailed argument about this). Again, the *check_queue* function is used to check whether any new messages can be delivered in total order.

The *check_queue* function shown in Figure 2.8 checks whether the next message to be delivered, *next_seq*, is now available to be delivered in one of the queues. If it is, the message is delivered using the *deliver_msg* function (Figure 2.9). *check_queue* also checks to see if a *null* message needs to be sent out, to ensure that other processes don't get blocked waiting for a message from this process. *null* messages are treated like normal messages by the total ordering protocol in the assignment of sequence numbers, but since they are generated by

rel_deliver(message m, process_rank i) IN VIEW V^x

```
1: Enqueue m in q_i.
2: if size(q_i) = 1 then
3:     call check_queue(i).
4: end if
```

Figure 2.7: The *rel_deliver* function

the protocol, they are not delivered. If a process is not sending a normal or a *null* message, and the protocol is being blocked because of this, the guilty process is reported as suspected to the group membership protocol. If the *check_queue* function delivers a message from the current group member's queue, it checks to see if the *send_queue* has any messages, and sends them out in a single composite message. The protocol does this to accommodate processes that generate messages much faster than the expected rate.

When a message needs to be dequeued and delivered, the *deliver_msg* (Figure 2.9) function is called. This function checks to see whether the message is a normal, composite, or *null* message. Normal messages are delivered immediately. Composite messages are broken into individual messages and delivered. Messages marked as *null* are discarded.

Before a view change can happen, the group membership protocol must inform all other protocols so that they don't send any more messages in a particular view. After that, the total-ordering protocol sends out all messages pending in its *send_queue*. The final function that gets called in a view is *clear_queues* (Figure 2.10), which is called by the infrastructure only after it has been ascertained that no process is going to send any more messages in this view. *clear_queues* uses the *deliver_msg* function to deliver messages. If there are sequence numbers for which a message is missing, they are ignored, and *next_seq* is incremented. Since the reliable multicast protocol is used by this protocol, it is guaranteed that messages not delivered at this process would not have been delivered at any other proper process either.

2.3.3 Sequence Number Generating Functions

The efficiency of the total ordering protocol depends upon choosing good sequence number generating functions. In our system, each process has a rank associated with it. If there are n processes in a view, then the ranks are $0, 1, \ldots, n-1$. In ITUA, a majority of the groups are replication groups, and replicas can be expected to generate similar traffic. A good set of functions for groups where all members generate similar traffic is: f_i such that $f_i(x) = x + n$.

```
check_queue(process_rank r) IN VIEW V^x
 1: total_pending_msgs ⇐ 0
 2: for i = 0 to |V^x| − 1 do
 3:    if size of q_i > q_len_hi then
 4:       total_pending_msgs ⇐ total_pending_msgs + q_len_hi
 5:    else
 6:       total_pending_msgs ⇐ total_pending_msgs+ (size of q_i)
 7:    end if
 8: end for
 9: for i = 0 to |V^x| − 1 do
10:    if (size of q_i > pending_msgs_lo) and (i ≠ my_rank) then
11:       Call suspect(i, "Pending Null messages")
12:    else if i = my_rank then
13:       Create a null message m_null
14:       tag(m_null, 'type', 'Null')
15:       Enqueue message in q_my_rank
16:       Call cast_rel(m_null)
17:    end if
18: end for
19: j ⇐ r
20: while true do
21:    if (first_seq_j = next_seq) and (q_j is not empty) then
22:       Dequeue message m from q_j
23:       Call deliver_msg(m, j)
24:       next_seq ⇐ next_seq + 1
25:       if (j = my_rank) then
26:          Dequeue all messages in send_queue and create a composite message
             m.
27:          tag(m, 'type', 'Composite')
28:          Enqueue m in q_my_rank
29:          Call cast_rel(m)
30:       end if
31:    else
32:       Stop. (No more messages can be delivered)
33:    end if
34: end while
```

Figure 2.8: The *check_queue* function

```
deliver_msg(message m, process_rank i) IN VIEW V^x
 1: msg_tag ⇐ get_tag(m, 'type')
 2: if msg_tag = 'Single' then
 3:    Call total_deliver(m, i).
 4: else if msg_tag = "Composite" then
 5:    for all msgs m_j packed within m do
 6:       Call total_deliver(m_j, i).
 7:    end for
 8: else
 9:    Stop. (null messages are discarded)
10: end if
```

Figure 2.9: The *deliver_msg* helper function

```
clear_queues() IN VIEW V^x
 1: while there is a non-empty queue do
 2:    for i = 0 to |V^x| do
 3:       if first_seq_i = next_seq then
 4:          next_seq ⇐ next_seq + 1
 5:          if q_i is not empty then
 6:             Dequeue message m from q_i
 7:             deliver_msg(m, i)
 8:          end if
 9:       end if
10:    end for
11: end while
```

Figure 2.10: The *clear_queues* function

All the functions are the same, but the initial sequence number assigned to each process is its rank. Consequently, each process generates a pair-wise disjoint sequence of numbers. Each message gets an equal number of slots to transmit. Our protocol will work even if a bad choice is made for the generating functions, but the penalty because of *null* messages will be high, leading to low performance.

An interesting possibility for sequence number generating functions is based on the concept of statistical time-division multiplexing, where time slots are allotted to communicating entities based on the previous traffic generated by them. In this scheme, the processes mon-

itor the message traffic and start a *gf-change* protocol if the traffic is not well-matched with the current generating functions. The processes communicate about the queue lengths and the number of messages they have pending. They then execute a consensus protocol to arrive at a new set of sequence number generating functions. The total ordering of messages proceeds even while this consensus protocol is proceeding. The processes have to choose a sufficiently large global sequence number, *change_num*, such that the new generating functions will be used to order messages with sequence numbers greater than *change_num*. This is done so that no messages are held up while the consensus stabilizes. This idea has not yet been implemented in the ITUA GCS.

2.3.4 Total Ordering Protocol Properties

This section provides an informal argument about how the total ordering protocol provides the *order* property described in Section 2.3.

Order

Definition: If proper processors p and q both deliver m_1 and m_2, then they deliver them in the same order.

Suppose m_1 is the x^{th} message from r delivered by p in the current view. As stated before, we assume that the multicast protocol used by the total ordering protocol provides reliable delivery and per-sender FIFO properties. Therefore, m_1 must also be the x^{th} message from r delivered by q in the current view. Since p and q are initialized with the same gf_is and seq_orig_is, they will assign the same sequence number to m_1 (by the definition of sequence number generating functions). The same argument applies to m_2 and it can be shown that p and q will assign the same sequence number to it. Since proper processes deliver messages in the order of their sequence numbers, m_1 and m_2 will be delivered in the same sequence by both p and q.

The fact that messages are delivered in the order of their sequence numbers can be seen by following the variable *next_seq* in the execution of the protocol. The variable is initialized to the lowest of the seq_orig_is and is incremented by 1 only when a message is being delivered (see line 24 of Figure 2.8 and line 4 of Figure 2.10). A message is delivered only when their sequence number is equal to the *next_seq*; this implies that they are delivered in the order of their sequence numbers.

Chapter 3

Implementation Details

The previous chapter described the reliable multicast and total ordering protocols developed for ITUA at an abstract level. To investigate the efficiency of these protocols, we have implemented them as part of a prototype intrusion-tolerant group communication system. A modified version of an existing GCS (C-Ensemble) was used as the basis for this rapid prototyping. The group membership protocol (GMP) described in [Ram], which is essential to the correct functioning of the multicast and ordering protocols, was concurrently implemented.

This chapter describes implementation issues that arose during the translation of the abstract protocols of Chapter 2 into working parts of an intrusion-tolerant GCS. We start by explaining the design of C-Ensemble at a high level in Section 3.1. Section 3.2 shows the main features of the new GCS, and Section 3.3 describes the implementation of the infrastructure. We end with three sections that describe the major implementation work done for this thesis. Section 3.4 describes how cryptographic support has been added to the infrastructure; Section 3.5 describes the implementation of the reliable multicast protocol; and Section 3.6 describes the implementation of the total ordering protocol. The material in the last two sections is presented in a way that assumes that the reader has read Chapter 2 and understands the total ordering and reliable multicast protocols described therein.

3.1 Layering Model

C-Ensemble is a C language implementation of the Ensemble [Hay98, Hay01] GCS. It follows a layered approach to communication protocol design. In the layered paradigm, the group communication system is divided into distinct modules, called *layers*, whose communication with other layers is restricted such that they directly communicate with only two other local layers, one of which is said to be *above* and the other *below*. The bottom layer interacts directly with the network infrastructure. As we go higher in the stack of layers, the level

of abstraction usually increases. Each layer may use the abstraction provided by the layers below it. Conceptually, each layer interacts directly with the corresponding layers in the protocol stacks of peer processes.

Layering makes it easier to change a system incrementally, as it is possible to replace one layer at a time. Layering also makes it possible for an application to choose the properties it desires from a GCS. For example, an application that does not require total ordering of messages can choose to exclude the layer(s) that provide this property from the stack. Another benefit of layering is that it makes it easier to validate the correctness, and evaluate the performance, of individual protocols. The cost of layering is in the overhead involved in messages traveling through various layers, and the abstraction barriers between layers. Often, in implementations of GCSs, it is necessary for non-adjacent layers to exchange information, requiring an event to be sent that passes through intermediate layers, with those intermediate layers not taking any action for the event.

The layering model used by Ensemble (and C-Ensemble) is described in [Hay98]. An overview of the main components is given below and shown graphically in Figure 3.1.

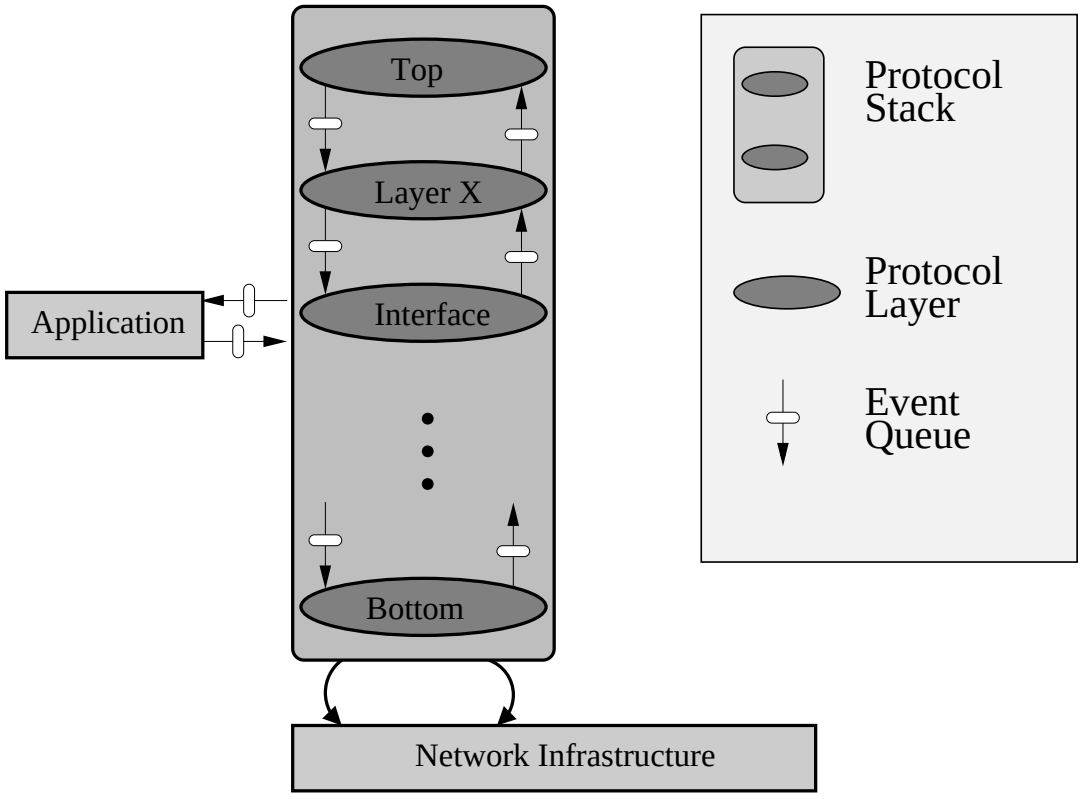


Figure 3.1: Main components of the Ensemble Layering Model

Protocol Layers and Stacks

The protocols implemented in Ensemble are broken down into various micro-protocols. Micro-protocols, individually or combined with other micro-protocols, provide a certain *property* to the application that is using the GCS. Micro-protocols are encapsulated as *layers*, which are combined together to build protocol *stacks*. A protocol stack provides the application with multiple properties.

Events and Event Queues

Events are records used by the layers for intra-process communication. An event may contain a reference to a message if the event carries information about that message. Events travel up and down the protocol stack through directional *event queues*. Events are placed on one end of a queue by one layer and removed from the other end in FIFO order. Only adjacent layers are connected by event queues.

Network and Application

The bottom of the stack communicates with the network. Events for sending messages that reach the bottom layer result in the transmission of the associated messages over the underlying network. At the receiving end, incoming messages are converted to receive events that enter the stack at the bottom layer. The application communicates with the protocol stack using application events, which include group join/leave and message send/receive events. Conceptually, the application can be thought of as being above the protocol stack. But as shown in Figure 3.1, this is not necessary. The interface layer in Ensemble is usually placed as low in the stack as possible to minimize delays to the application messages.

Messages

Messages are objects that carry information and are transmitted over the network. As is common in networking protocols, in Ensemble a message is divided into two parts: the payload and headers. The payload is the information content of the message created either by the application or by a layer. *Headers*, containing control information, are added to the message by the layers as the message passes through them and are removed at the receiving end. Protocol stacks being used by processes in a single group have layers in the same order. So, each layer in the receiver's stack accesses headers added by the corresponding layer in the sending stack. A layer normally cannot access headers of other layers.

Processes and Endpoints

A *process* is a unit of state and computation and is an abstraction provided by the operating system of the host. A single process can contain multiple identifiers, which are used by the GCS to identify individual group members. An *endpoint* corresponds to an individual group member. For simplicity, in this thesis, a single process is assumed to contain only one endpoint[1] at a time, so the two terms are used interchangeably.

View State and Local State

As explained in Chapter 2, the GMP manages and updates membership information by installing a series of views at each one of the group members. The GMP installs a new view whenever some key attribute of the group (e.g. membership) changes. The *view state* is a record containing all the information about the view that a member needs for communicating with the group. Therefore, a view change essentially consists of updating the contents of the view state record at all group members. The view state has fields for the number of members, the group id, and the list of network addresses of members in the group. In each view, the GMP ensures that the view states of all proper members are the same (given that the resilience condition is met). Each layer has access to the view state, and also to a common record containing the local state. The *local state* stores member-specific information, like the member's rank and its network address.

3.2 Protocol Stack for the Intrusion-tolerant GCS

The protocol stack for the prototype intrusion-tolerant GCS is shown in Figure 3.2. The layers in this customized stack together provide the properties of group membership, reliable delivery and total ordering required by groups in ITUA. Other layers of the original C-Ensemble stack, such as *Primary* and *Xfer* have not been modified[2]. These layers, which are not part of the customized stack, may not work in the modified C-Ensemble GCS because of infrastructure changes we have made. Configurability is not a goal of the ITUA GCS, so the ability to add these layers is not required. If configurability is desired, these layers could be modified to work in the ITUA GCS.

In the customized stack, the new *ITUA_Group* layer [Ram], along with the layers *Heal, Suspect, Leave, Inter*, and *Intra*, provides the intrusion-tolerant group membership properties required by the reliable multicast and total ordering protocols. The *Total* and *Reliable* layers

[1] This is not a requirement of the protocols or the implementation.

[2] The exception is *Mnak*, which was required for the performance measures described in Chapter 4.

are the implementation of the total ordering and reliable multicast protocols described in Chapter 2.

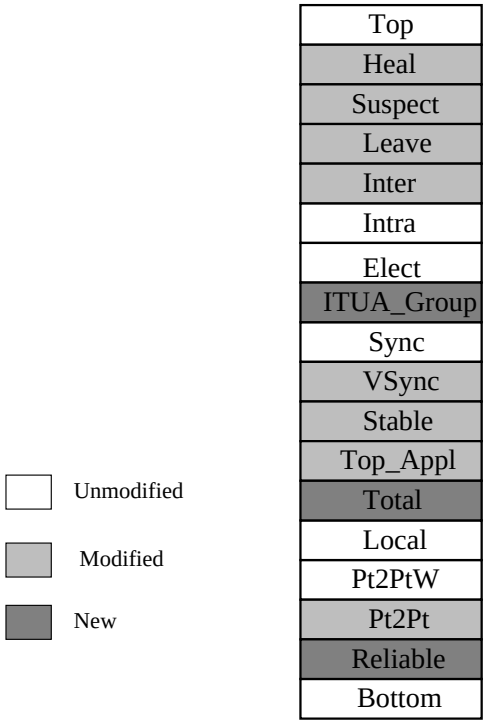


Figure 3.2: Modified C-Ensemble stack

3.3 Infrastructure Implementation

Each layer is implemented as a set of event handlers, which are called by the infrastructure whenever events need to be processed by a layer. Events can be passed up and down by the layers. When a layer passes an event up or down, the event is stored in a corresponding event queue. The event queues are processed in FIFO order by a scheduler, and eventually the *up* or *down* event handler of the layer immediately above or below the original layer is called. Layers are not aware of exactly which other layers are present in the stack, but each layer does make assumptions about the properties provided by the other layers. New events can enter the stack in the following ways:

- The infrastructure receives a message for the stack from the network, in which case an *up* event handler is called on the *bottom* layer, with a reference to the message.

- The application sends a message, or sends some control information to the stack. In that case, the *down* event handler of the interfacing layer *top_appl* is called.
- The infrastructure sends some control information, e.g., the *init* event (for initialization), to the stack. Such control events can enter the stack either at the *bottom* or *top* layer.
- A layer creates a new event while processing some event in an event handler.

Every layer needs to maintain some state information, which is preserved across calls to the event handlers. This information is stored in the *state* record of the layers. The *state* record contains references to the stack-wide view state (*vs*) and local state (*ls*) records. When a new view is installed, the infrastructure calls the *init* function of each layer of the new stack. This function is where each layer implements its initialization routines. References to the *vs* and *ls* records are passed to the layers in this initialization call.

Certain events, e.g., *cast* and *send*, are associated with messages that are to be sent or have been received. The layers involved with message delivery need to add headers to these messages, to convey information to the corresponding layers in the peer stacks. This is done in C-Ensemble using the *marsh* function. On the receiving side, the layers use the *unmarsh* function to retrieve the headers attached to the message.

When a received message passes through a layer in the stack, the layer should be able to decide whether or not to process the message and remove headers. To enable this decision, all layers of the sending stack add a nominal header to the outgoing message. If the message is not processed at all by a sending layer, this header has the value NOHDR, which tells the corresponding receiving layer to pass up the received message without processing it.

3.4 Cryptography Details

The *reliable* layer and the *ITUA_Group* layer frequently use digital signatures to verify the authenticity of messages received. One of the changes we had to make to the C-Ensemble infrastructure was the addition of cryptographic support. We used Peter Gutmann's Cryptlib [Gut01] as the core cryptographic library, and wrote wrapper functions around it.

Table 3.1 describes some of the wrapper functions written for the new GCS. The reliable multicast protocol encapsulates most of its cryptographic requirements in the creation and verification of *notices* and *notice_replies*. In the implementation, a data structure *message_digest_t* (Figure 3.3) has been added, which corresponds to the *notices* described in Section 2.2.1.

```
struct message_digest_t{
  rank_t  my_rank; /* rank of sending process            */
  seqno_t seqno;   /* local sequence number of message   */
  unsigned char view_str[MAX_VIEWID_STR]; /* view-id     */
  unsigned char hash[CRYPT_MAX_HASHSIZE]; /* hash digest */
  len_t hash_len;  /* length of hash digest              */
};
```

Figure 3.3: The *message_digest_t* data structure

The *reliable* layer makes extensive use of the wrapper functions described in Table 3.1. For example, the *crypt_hash_buf* wrapper function is used to create the hash of the message being sent, which is then stored in a *message_digest_t*. The *crypt_sign_buf* wrapper function is used to sign the *message_digest_t*. These two steps together create a *notice* as shown in line 2 of Figure 2.2. The *crypt_sign_buf_verify* wrapper function is used when the notice is received (line 3 of Figure 2.3). The *crypt_hash_buf_verify* wrapper function is used to check that the hash stored in a notice corresponds to the contents of a message (line 6 of Figure 2.5). The *marsh_crypt_sign* and *marsh_crypt_verify* wrapper functions are used by the layers to sign and verify messages that are not being reliably multicast. They are used, for example, when the *msgs_delivered* arrays are being exchanged by processes at the time of a view change (described in Section 2.2.1).

Function name	Arguments	Description
crpyt_hash_buf	buffer b	returns the hash digest of b
crypt_hash_buf_verify	buffer b, hash h	checks whether h is the correct hash for b
crypt_sign_buf	buffer b, RSA key k	returns a signature using digest of b signed with k
crypt_sign_buf_verify	signature s, buffer b, RSA key k	checks s is a valid signature on b
marsh_crypt_sign	message m, key k	signs m using k; sign is added to message as a header
marsh_crypt_verify	message m, key k	removes the last header from m and verifies that it's a valid signature

Table 3.1: The cryptographic support functions added to the infrastructure

Many layers, including the GMP layers, needed access to cryptographic key information about the group members. This information was added to the *local state*. The following new fields were added to *local state*:

- *private_keyset:* reference to a file which stores the private key of the group member.
- *passwd* password for accessing the key stored in *private_keyset*.
- *public_keyset:* reference to a file which stores public keys of all possible members of the group. Each key is associated with a *key_id*, which is used to access a particular key.
- *key_id_list:* list of *key_ids* needed to access the *public_keyset*.
- *my_key_id: key_id* of the group member. These keys are exchanged by members during group formation and when a member joins a group.

Other changes had to be made to add cryptography to C-Ensemble. The view state record was modified; it now contains a field *key_ids* which stores the key ids of the processes present in this view in order of rank. A new data-structure *sign_collect_t* was added to make it easy and efficient to store *notice_replies* from peers.

3.5 Reliable Multicast

The implementation of the *reliable* layer closely follows the protocol described in the previous chapter. This section describes interesting issues that arose during the implementation and how they were solved.

Messages that *reliable* delivers cannot be immediately discarded, because a NAK for the message might be received and the message must be available so that it can be forwarded to members who didn't receive it. However, if messages are buffered indefinitely, the message queues will grow without bound and the process will run out of memory. To avoid this problem, the stack has a *stable* layer through which all processes exchange information about the number of messages that they have delivered. Once *all* group members agree that messages up to a particular sequence number[3] from a sender have been delivered, all processes can discard messages from that sender with lesser sequence numbers. Messages which all processes have delivered are called *stable* and the sequence number of the highest stable message is called the *stability threshold*. Processes ignore any messages or retransmission requests for sequence numbers lesser than the stability threshold. Processes also ignore

[3]Recall that sequence numbers in *reliable* are per-sender, while in *total* there is a global set of sequence numbers.

messages with sequence numbers that are higher than the *overflow threshold*. The overflow threshold is obtained by adding a window size to the stability threshold. This avoids a situation in which a corrupted process can fill up the message buffers of all other group members. The message queue for a single sender is shown in Figure 3.4.

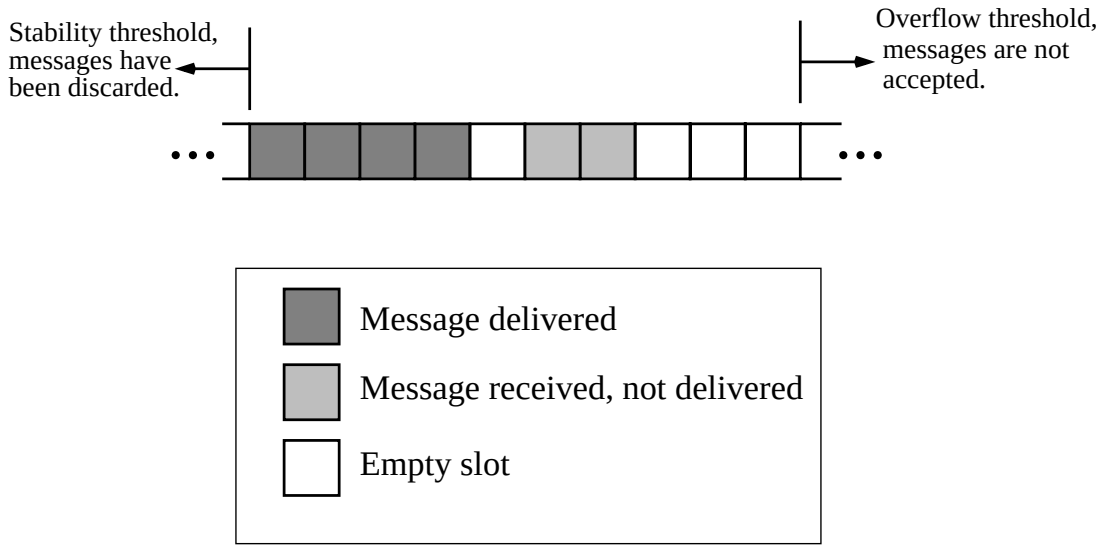


Figure 3.4: Message buffer for a single sender, showing sequence number thresholds.

Higher-level layers assume that the messages they send are reliably delivered, and that they therefore don't need to buffer messages. Consequently, a protocol layer that assumes reliable delivery cannot discard any messages, because its peer layer will not be able to retransmit any messages. However, a higher layer's buffer might be filling up and it could therefore want to stop receiving messages from a particular member until it has processed some of the messages already in its buffer. For that reason, we have added an event *ignore_peer* to the infrastructure. *reliable* ignores messages from a particular sender if it has received an *ignore_peer* event for the sender. It continues to do so until it receives another event telling it to start accepting messages from the peer again.

3.6 Total Ordering

The total ordering protocol described in the previous chapter has been implemented as a single layer called *total*. This layer operates on top of the *reliable* layer, and buffers the reliably delivered messages until they can be delivered in a total order. The *msg_queues* described in Section 2.3 are implemented as an array of linked lists. The most important events related to *total* are *up(cast)* and *down(cast)*, both of which have a reference to a

message. The *down(cast)* event is created by a higher layer (the *top_appl* layer, if the application sends a message), and has a field, *req_total*, that specifies whether total ordering is desired. When a *down(cast)* event reaches *total*, the layer checks to see if *req_total* is set. If it isn't set, *total* just attaches the NOHDR header to the message and passes the send event down. If total ordering is desired, the layer follows the steps of the protocol described in Figure 2.6. In the implementation the out-going message is not enqueued in the sender's own queue, as is done in the protocol. This is because the *local* layer, which bounces multi-cast messages to the sender, makes this step unnecessary. The header added by the *total* layer has three parts:

- *ignore:* flag to indicate whether other fields are present or not. A value of NOHDR, indicates that there are no other fields.
- *type:* field to specify whether the message is "Single," "Compound," or "Null." This information is needed by the *deliver_msg* function shown in Figure 2.9.
- *appl:* flag to indicate whether the message is an application message.

When an *up(cast)* event is received, the layer strips the first header field and checks if it is a NOHDR; if this is the case the event is simply handed over to the next layer. If it isn't, the layer gets the rank of the sender from the *origin* event field and enqueues the message in the corresponding *msg_queue*.

In the absence of the *total* layer, the *top_appl* layer keeps track of the number of messages received by the application from each group member. A new view installation is not allowed to proceed until all messages that were sent, in this view, by the application at the other members have been received. A record of the number of messages that were sent by the application at each process is maintained by the *vsync* layer. This guarantees that all messages that are sent by the application in a particular view will be received by all proper processes in the same view.

In order to call the *clear_queues* function (Figure 2.10) and send up messages stored in its *msg_queues*,the *total* layer needs to receive the event signalling a view change. There can be a deadlock if there are application messages in the *msg_queue* that can only be delivered when a new view is about to be installed. The *top_appl* layer will not allow this new view notification to proceed until it has received the application messages, and *total* will not send up the messages until it receives the notification. To resolve this situation, a new event, *msg_reliable*, has been added to the infrastructure. That event is sent up is sent up by *total* when it receives a new message with the *appl* header field set. When *top_appl* receives a *msg_reliable* event it knows that a new application message has been received and buffered

at *total* and will eventually be delivered in this view. Thus, *top_appl* will allow a view change to proceed even though it has not received all application level messages, if these messages are buffered in the *total* layer.

The set of sequence number generating functions (gf_is from Section 2.3.2) is implemented as a single function that takes two arguments, a sequence number and a process rank. The sequence number passed in should be a valid sequence number for the process rank. The next sequence number for the group member is calculated and returned. A reference to this function should be passed to *total* in the *init* function of the layer. The default generating function, described in Section 2.3.3, simply adds the the number of members in the current view to the input sequence number and returns the result.

Chapter 4

Performance Measurement

An important reason for developing a prototype implementation of the ITUA GCS was to measure the cost of building intrusion-tolerance mechanisms at the group communication level. We have written two applications with process groups that communicate over the network, and have studied the variation of message delivery times with varying group sizes, stack configurations, and layer parameters. This chapter describes the measurements done to gauge the different performance costs associated with an intrusion-tolerant GCS.

Section 4.1 describes the distributed environment in which the experiments were carried out. It also gives descriptions of the two applications used to measure the performance and the parameters that were varied during the experiments. Section 4.2 shows some of the results that we obtained, and explains the significance of these measures. The costs associated with cryptography and with our reliable multicast and total ordering protocols are presented.

4.1 Experimental Setup

The tests were carried out on a testbed of ten 200 MHz Pentium Pro CPU computers with 128 MB RAM. The computers were connected by a full-duplex 100 Mbps 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.

4.1.1 Application R: Response Time for a Single Multicast

In application R, each process is started with the same group name. The group membership protocol ensures that all processes join a single group. When the group size reaches

group_size, the process with rank 0, p_0, multicasts a message of size *msg_size*. On receiving this message, p_1 multicasts a new message, which when it is received by p_2, causes the process to multicast another message, and so on. Process p_0 multicasts its next message when it receives the multicast sent by $p_{(group\_size-1)}$. Each multicast message is signed using RSA cryptography with keys of size *key_size*. The parameters *group_size*, *msg_size*, and *key_size* are command-line arguments to the process. In order to keep time measurements local, each process stores the local time whenever it receives a message from p_0. Processes continue to multicast messages in this fashion until they have *num_rounds*+1 time measurements. They then calculate the difference between subsequent time measurements, and divide each of the *num_rounds* number of differences obtained, by *group_size* to obtain an estimate of the time it takes for a single multicast to be delivered.

This experiment was repeated for values of *group_size* ranging from 4 to 10. The value of *num_rounds* used was 30. We noticed that the *msg_size* parameter did not have a significant impact on the times measured, so a constant *msg_size* of 1 KB was used. The reason for this is explained in Section 4.2.3.

The same application was run on top of four different protocol stacks:

mnak-no_total: This stack has no *total* layer, and the reliable delivery property is provided by the *mnak* layer from C-Ensemble which tolerates only crash faults. This layer does not use cryptography.

reliable-no_total-dummy_crypt: This stack uses the new *reliable* layer, but a dummy version of the cryptography library. This dummy cryptography library returns from function calls immediately without performing the expensive cryptographic routines needed to sign, verify, encrypt, or decrypt messages.

reliable-no_total: This stack uses the *reliable* layer, but does not have the *total* layer. The *reliable* layer uses the normal cryptography functions.

reliable-total: This stack has both the *reliable* and *total* layers. Again, the *reliable* layer uses the normal cryptography functions.

These stacks were chosen so that we would be able to compare the individual costs of various layers. Comparing the message delivery times for the **reliable-no_total** and **reliable-total** stacks gives us a good estimate of the latency caused by adding the *total* layer. The difference between delivery times for **reliable-no_total-dummy_crypt** and **reliable-no_total** is the overhead caused by cryptography. It should be noted that the *reliable* layer depends on cryptography for correctness, and that the **reliable-no_total-dummy_crypt**

stack does not provide intrusion tolerance. Another comparison we made was between the **reliable-no_total-dummy_crypt** and **mnak-no_total** stacks to see the overhead caused by the gathering of *notice_replies* before sending a message. That gives us a lower bound on the overhead of using the intrusion-tolerant *reliable* layer, as opposed to the crash-tolerant *mnak* layer, when the cryptography operations take negligible time.

4.1.2 Application S: Simultaneous Multicast by All Group Members

In application S, the processes are started and, as in the previous experiment, wait for the group size to reach *group_size* before beginning to transmit messages. Each process then records the start time and sends *num_init_casts* initial multicasts to all members. After this burst, another multicast is sent out every time the process receives the number of messages indicated by *group_size*. Each message sent out is of size *msg_size*, and RSA cryptography with keys of size *key_size* is used. The end time is noted when the process has received 10 $\times$ *group_size* messages and the elapsed time is calculated and written to a file. As in the previous experiment, *group_size*, *num_init_casts*, *key_size* and *msg_size* are command-line arguments to the processes.

The results reported here were obtained from 10 independent runs, each run collected data for varying parameter values. The runs were repeated for the four protocol stack configurations mentioned for Application R. For each stack configuration, a run varied the *group_size* from 4 thru 10, and if the stack used cryptography, *key_size*s of 512, 768, and 1024 were used. The measurements reported here are for the value of *num_init_casts* for which the best timings were obtained. When the value of *num_init_casts* is too low, the rate of multicasts by processes gets slowed because processes need to wait for messages before multicasting. If the value is too high, the network becomes congested because of the initial burst of messages from all processes, and many messages have to be resent, slowing down the application.

4.2 Experimental Results

The rest of this chapter presents the results of the various experiments conducted, and draws some conclusions about the cost of intrusion tolerance. In the tables presented n denotes the number of group members, and f denotes the number of process corruptions being tolerated. All confidence intervals shown in tables are for 95% confidence and were computed by assuming that the samples were from a normal distribution.

The variations of message delivery time with changing group sizes and key sizes for a stack with *reliable* and *total* are shown in Figures 4.1 (Application R) and 4.2 (Application S). The details of the corresponding results are shown in Tables 4.1 (Application R) and 4.2 (Application S). In both experiments, the sharp rise in cost as the key size increases is expected and is in keeping with the rise in the cost of public-key cryptography operations with larger keys.

It can be seen from the graphs, that in both applications, the time taken for message delivery suddenly rises at the group size values of 7 and 10. This is because at these group sizes, the number of faults being tolerated changes. Details of this are given in Section 4.2.3.

There are two reasons why the timings reported for experiment S are higher than those for experiment R. The first reason is that in S we measure the time for $10 \times$ *group_size* multicasts to complete, whereas in R the time taken for a single multicast is measured. In S however, *group_size* number of processes are multicasting at the same time, as opposed to a single member in R. So, as an approximation, if we assume that *group_size* number of multicasts are proceeding at the same time in S, we expect S to take approximately 10 times more clock ticks than R. The second factor is that experiment S causes a lot more load on the CPUs and the network because all processes are multicasting at the same time.

n	f	512-bit RSA		768-bit RSA		1024-bit RSA	
		Mean	Confidence interval	Mean	Confidence interval	Mean	Confidence interval
4	1	211	(211, 212)	261	(261, 261)	341	(341, 341)
5	1	211	(211, 211)	262	(262, 262)	341	(341, 341)
6	1	211	(211, 211)	262	(262, 262)	342	(342, 342)
7	2	231	(230, 231)	287	(286, 287)	373	(372, 373)
8	2	231	(231, 231)	287	(287, 287)	373	(373, 373)
9	2	231	(231, 231)	289	(289, 289)	373	(373, 373)
10	3	250	(250, 250)	312	(312, 312)	406	(406, 406)

Table 4.1: Application R: Effect of increasing group and key sizes. (Stack with *reliable* and *total*)[Time measures represent 10^5 clock cycles]

4.2.1 Cost of Total Ordering

The performance of the *total* layer is highly dependent on the accuracy with which the sequence number generating functions mirror the actual traffic generated by the individual group members. Suppose that we chose the default sequence number generating functions described in Section 2.3.3, with the global sequence numbers evenly divided among the

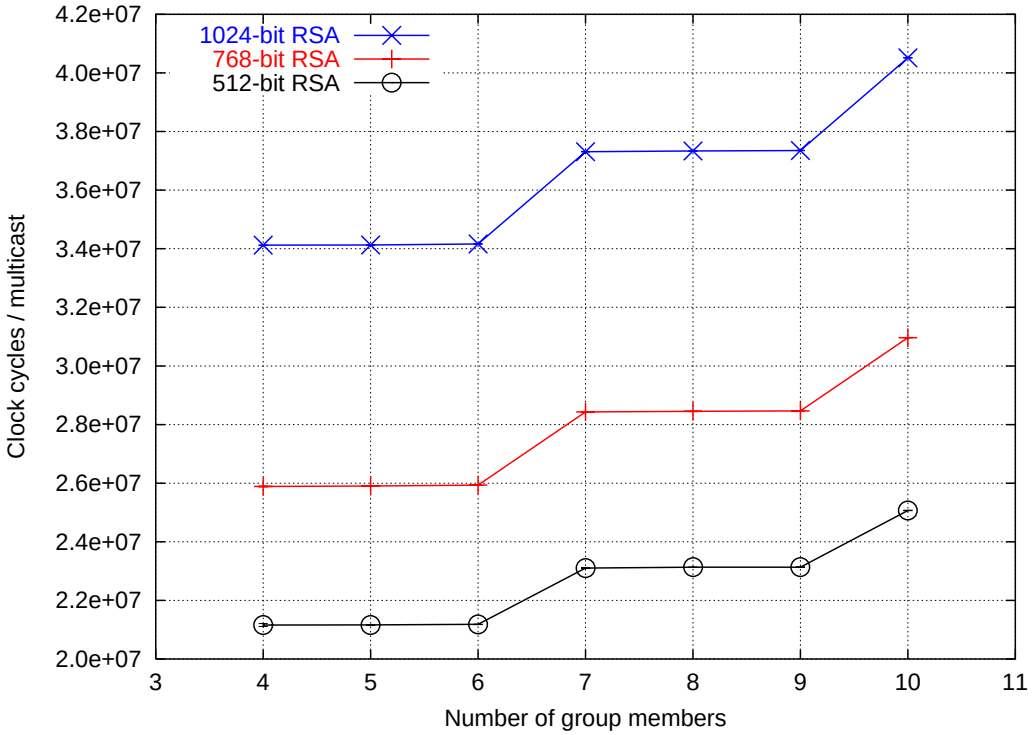


Figure 4.1: Application R: Effect of increasing group and key sizes. (Stack with *reliable* and *total*)

n	f	512-bit RSA		768-bit RSA		1024-bit RSA	
		Mean	Confidence interval	Mean	Confidence interval	Mean	Confidence interval
4	1	440	(438, 441)	548	(546, 549)	714	(713, 715)
5	1	596	(594, 599)	744	(739, 749)	969	(965, 973)
6	1	717	(715, 719)	892	(889, 896)	1180	(1145, 1215)
7	2	908	(905, 911)	1134	(1132, 1136)	1536	(1525, 1548)
8	2	1033	(1030, 1037)	1311	(1301, 1322)	1746	(1735, 1758)
9	2	1168	(1163, 1173)	1526	(1519, 1534)	2037	(2008, 2067)
10	3	1452	(1431, 1472)	1879	(1862, 1897)	2593	(2540, 2646)

Table 4.2: Application S: Effect of increasing group and key sizes. (Stack with *reliable* and *total*) [Time measures represent 10^6 clock cycles]

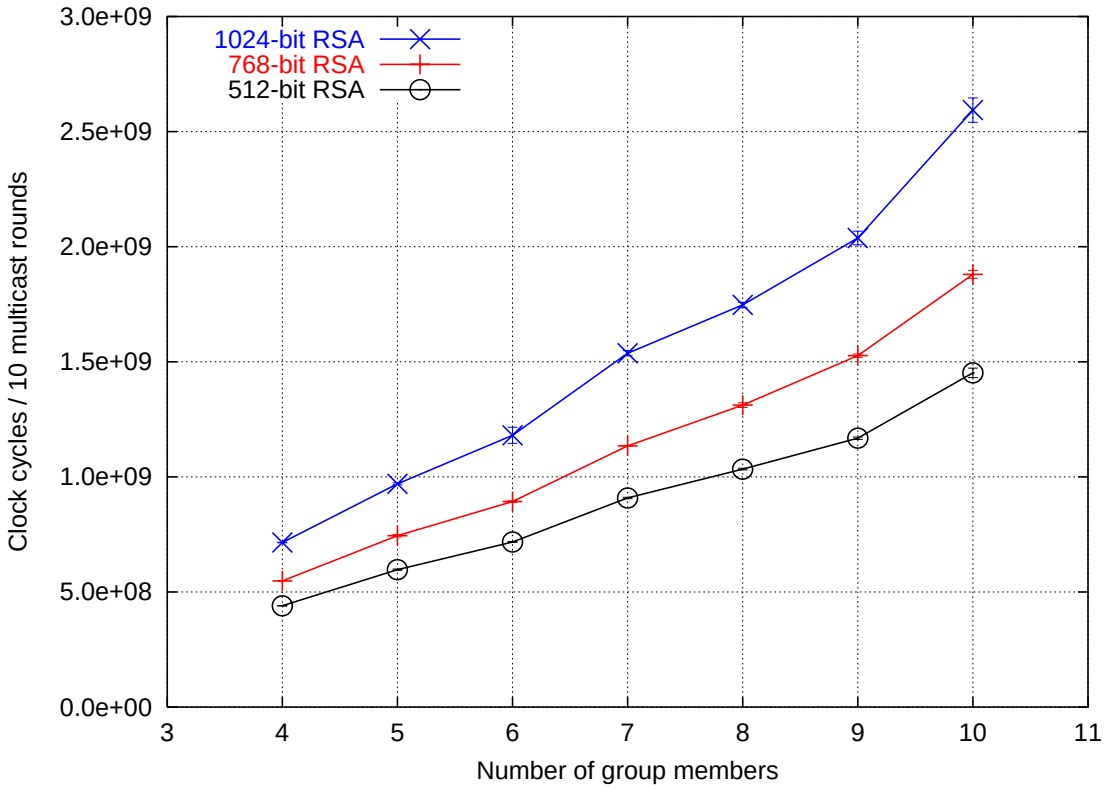


Figure 4.2: Application S: Effect of increasing group and key sizes. (Stack with *reliable* and *total*)

processes. If the actual traffic consisted of a single process sending all the messages, our protocol would have a high overhead, because of *null* message delays. We believe that our use of the ITUA GCS will involve groups that have somewhat predictable trends in generated traffic, so we have kept the traffic fairly close to what the generating functions assume.

Application R represents the best case for the default generating functions. The overhead of *total* is just the amount of time taken for the message to pass through the layer, with the addition or removal of a header and some processing. No message is ever buffered by the layer, since the arriving message is always the next one in the pre-determined global sequence. Any other pattern of traffic will have extra overhead because of messages being buffered while the protocol waits for other messages. Figure 4.3, shows a comparison between the variation of multicast delivery times with group size in two cases: one stack has both the *reliable* and *total* layers, the other has only *reliable*. The figure shows that the cost of adding the *total* layer is small compared to the cost of reliably multicasting messages. It can be seen by a comparison of Tables 4.1 and 4.3 that the cost of adding *total* is consistently less than 2%.

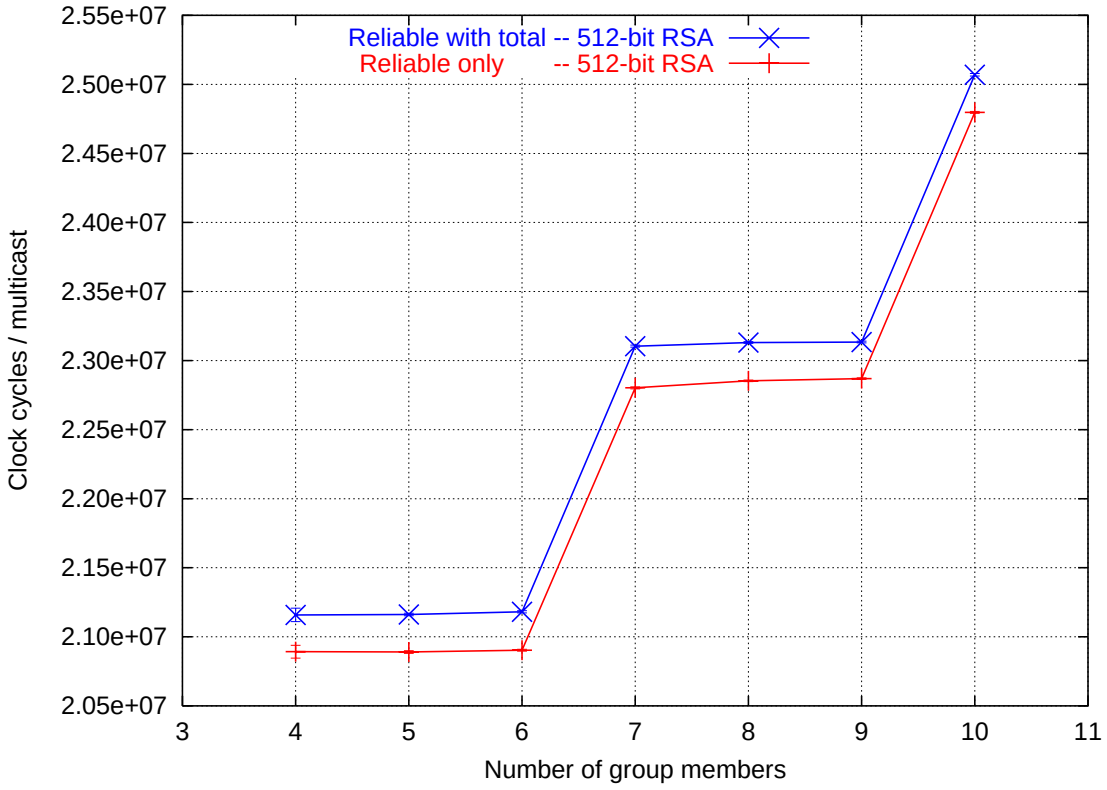


Figure 4.3: Application R: Cost of total ordering layer

n	f	512-bit RSA		768-bit RSA		1024-bit RSA	
		Mean	Confidence interval	Mean	Confidence interval	Mean	Confidence interval
4	1	208	(208, 209)	261	(261, 261)	341	(341, 341)
5	1	208	(208, 208)	262	(262, 262)	341	(341, 341)
6	1	209	(208, 209)	262	(262, 262)	341	(341, 341)
7	2	228	(227, 228)	287	(286, 287)	373	(372, 373)
8	2	228	(228, 228)	287	(287, 287)	373	(373, 373)
9	2	228	(228, 228)	287	(287, 288)	373	(373, 373)
10	3	247	(247, 248)	312	(312, 312)	405	(405, 405)

Table 4.3: Application R: Effect of increasing group and key sizes. (Stack with *reliable* and no *total*) [Time measures represent 10^5 clock cycles]

In application S, the default generating function is still applicable, because processes send messages at roughly the same rate. However, this application's traffic does not strictly follow the default generating function, because the system is asynchronous and the multicasts are not in strict order of the global sequence numbers. For that reason, the *total* layer does have to buffer messages until messages with lower sequence numbers arrive. Figure 4.4 compares two stacks that both have the *reliable* layer but only one of which has *total*. Comparing Tables 4.2 and 4.4 shows that the overhead caused by *total* is below 12% in almost all cases.

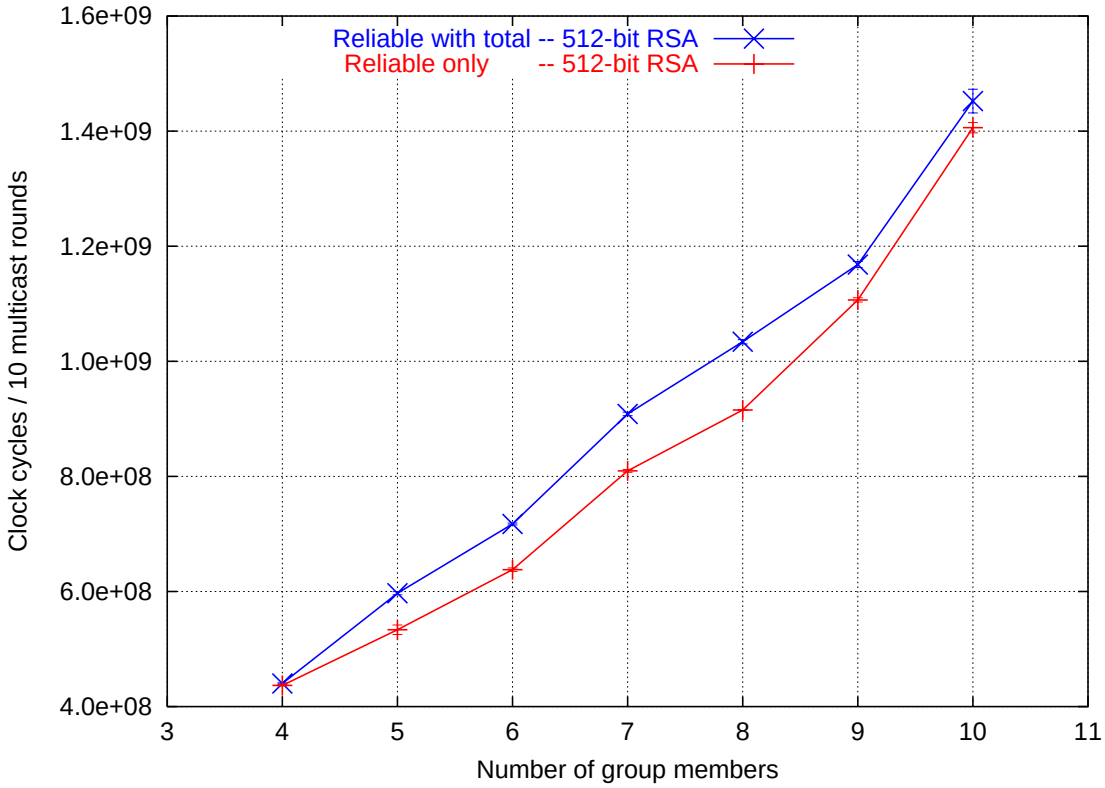


Figure 4.4: Application S: Cost of total ordering layer

4.2.2 Cost of Reliable Multicast

The cost of reliable multicast can be broken down into two parts. The first is the cost introduced by the cryptographic functions, which is discussed in the next section. The other cost is caused by the extra round of communication needed by *reliable* to exchange *notices* and *notice_replies* before the actual message is sent. To estimate this overhead, we compared the delivery times when the original crash fault-tolerant *mnak* layer was used and

n	f	512-bit RSA		768-bit RSA		1024-bit RSA	
		Mean	Confidence interval	Mean	Confidence interval	Mean	Confidence interval
4	1	436	(436, 437)	542	(540, 543)	708	(707, 709)
5	1	533	(525, 541)	652	(647, 656)	859	(848, 871)
6	1	638	(634, 641)	782	(777, 786)	1094	(1089, 1099)
7	2	809	(807, 812)	1033	(1014, 1052)	1436	(1422, 1449)
8	2	915	(914, 916)	1209	(1193, 1224)	1693	(1664, 1723)
9	2	1106	(1102, 1110)	1394	(1378, 1409)	1950	(1912, 1987)
10	3	1405	(1397, 1414)	1876	(1851, 1901)	2483	(2428, 2538)

Table 4.4: Application S: Effect of increasing group and key sizes. (Stack with *reliable* and no *total*) [Time measures represent 10^6 clock cycles]

when the new *reliable* layer was used, but the cryptographic functions were replaced by dummy versions.

The comparison for application R is shown in Table 4.5. Figure 4.5 shows the behavior of the two protocols with changing group size. The crash-tolerant delivery time grows very steadily with increasing group size, because of the way multicasts are handled at the lower layers. The growth of delivery times for the stack with *reliable* is similar, except there are steeper jumps when the number of processes reaches 7 and 10. This is because 7 and 10 are of the form $3f + 1$, and the number of faults being tolerated changes at those group sizes. Thus, when the group size changes from 6 to 7, the number of faults tolerated, $\lfloor (n-1)/3 \rfloor$, goes from 1 to 2. Correspondingly, the number of *notice_replies* being collected $(2f + 1)$ changes from 3 to 4. The sending process, therefore, must wait for 3 replies (counting its own reply as the first) from others instead of 2; this causes more delays. There is a similar change when the group size goes from 9 to 10.

The crash-tolerant protocol sends a message to a peer, and if the message does not get lost it's delivered at the receiver. On the other hand, in our protocol the process sends out a *notice*, waits for a *notice_reply*, and then sends the message. This explains the approximately three-fold difference between the two timings seen in Figure 4.5.

Similar behavior is seen in the case of application S. This is shown in Table 4.6. Figure 4.6 shows that the time taken for reliable delivery worsens more rapidly than for application R. That is because all processes are simultaneously multicasting in this application, so the traffic on the network increases rapidly with growing group size.

Group size	Crash-tolerant protocol		Reliable without cryptography	
	Mean	Confidence interval	Mean	Confidence interval
4	131	(131, 133)	308	(306, 310)
5	138	(138, 139)	325	(323, 327)
6	147	(146, 148)	341	(340, 343)
7	157	(155, 159)	395	(392, 397)
8	164	(163, 166)	409	(407, 411)
9	173	(172, 175)	429	(426, 431)
10	183	(181, 185)	487	(486, 488)

Table 4.5: Application R: Comparing crash-tolerant reliable multicast with intrusion-tolerant reliable multicast (Stack with *reliable* using dummy crypto, and stack with crash-tolerant *mnak*) [Time measures represent 10^3 clock cycles]

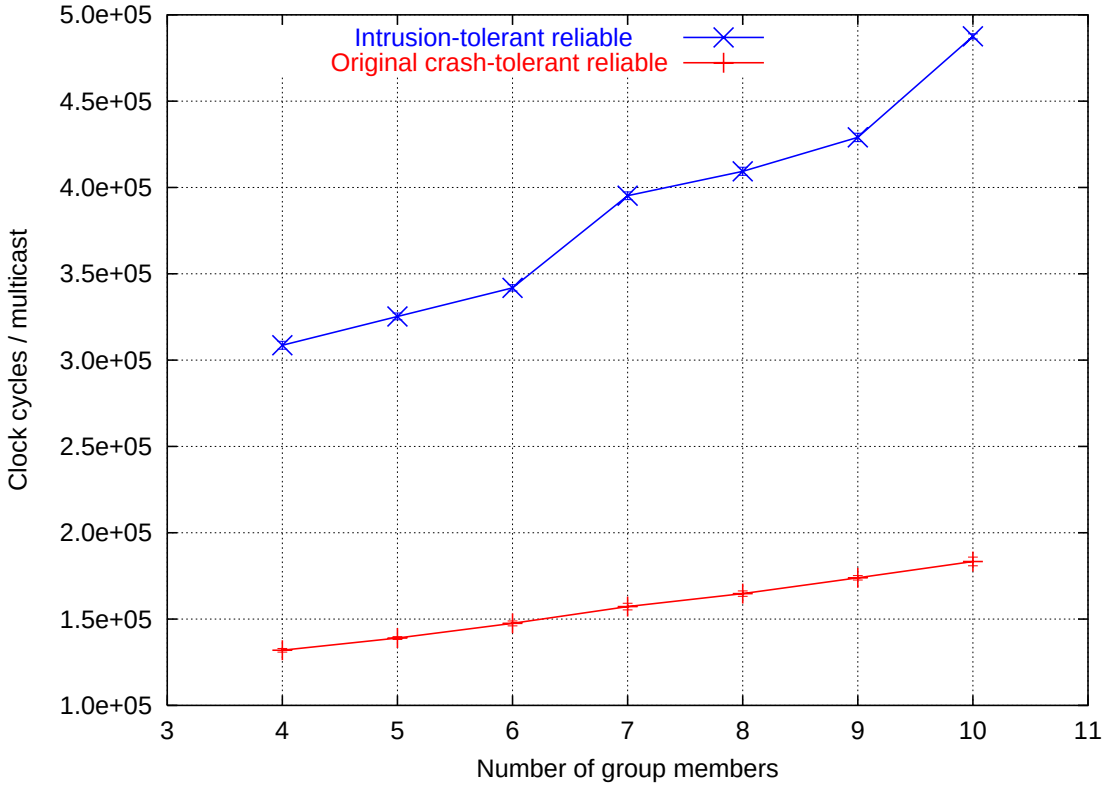


Figure 4.5: Application R: Measuring the overhead of the extra round in *reliable*

Group size	Crash-tolerant protocol		Reliable without cryptography	
	Mean	Confidence interval	Mean	Confidence interval
4	225	(220, 231)	584	(580, 588)
5	291	(285, 297)	764	(733, 795)
6	358	(352, 363)	939	(915, 963)
7	407	(401, 413)	1141	(1124, 1158)
8	524	(519, 529)	1314	(1300, 1327)
9	792	(782, 801)	1482	(1474, 1489)
10	1117	(1104, 1131)	1874	(1806, 1941)

Table 4.6: Application S: Comparing crash-tolerant reliable multicast with intrusion-tolerant reliable multicast (Stack with *reliable* using dummy crypto, and stack with crash-tolerant *mnak*) [Time measures represent 10^4 clock cycles]

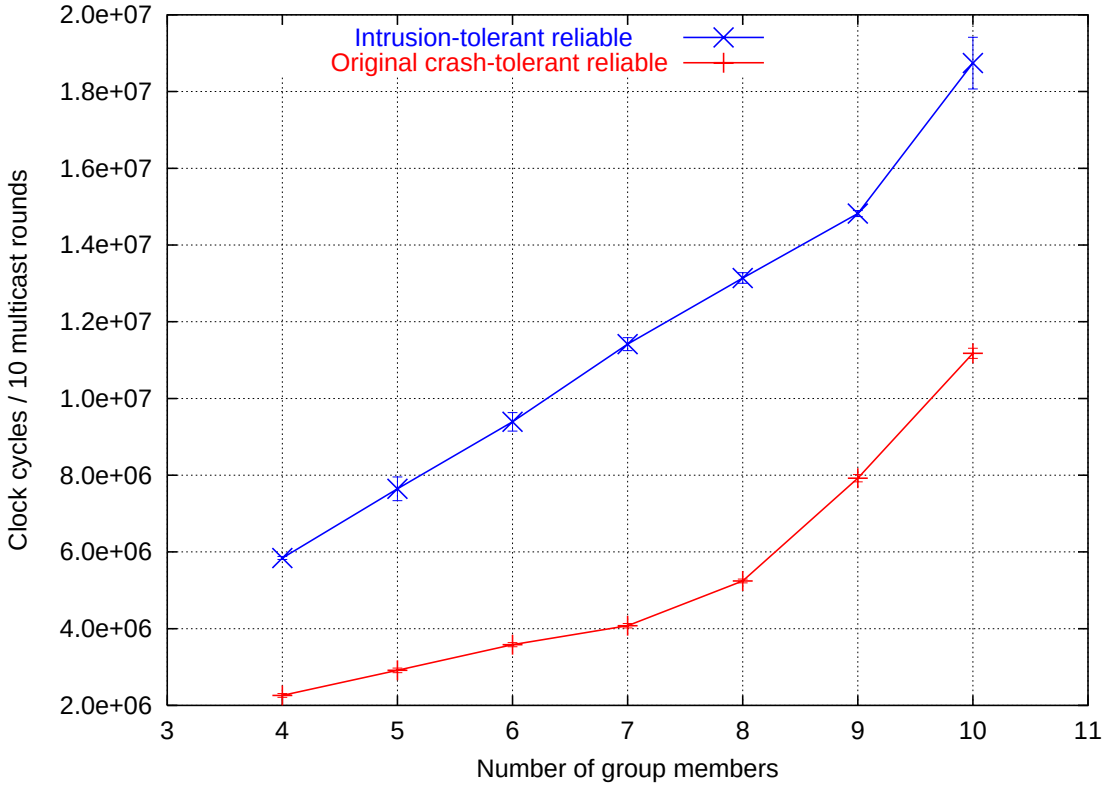


Figure 4.6: Application S: Measuring the overhead of the extra round in *reliable*

4.2.3 Cost of Cryptography

The performance costs due to the use of cryptography can be determined by comparing at the performance of the stack with *reliable*, which uses the cryptographic library and the performance of a similar stack in which the cryptographic routines have been replaced by dummy routines. We saw a two orders of magnitude difference between the performance of the two stacks in both experiments. This is shown in figures 4.8 (Application R) and 4.7 (Application S).

These results were expected, because of the high computational costs of public-key cryptography. One thing to be noted, however, is that for the experiments with cryptography, the CPU was the bottleneck, but the CPU was hardly utilized when the stack did not use cryptography. It follows that the relative performance cost of the cryptographic routines will be much less severe when more powerful machines are used.

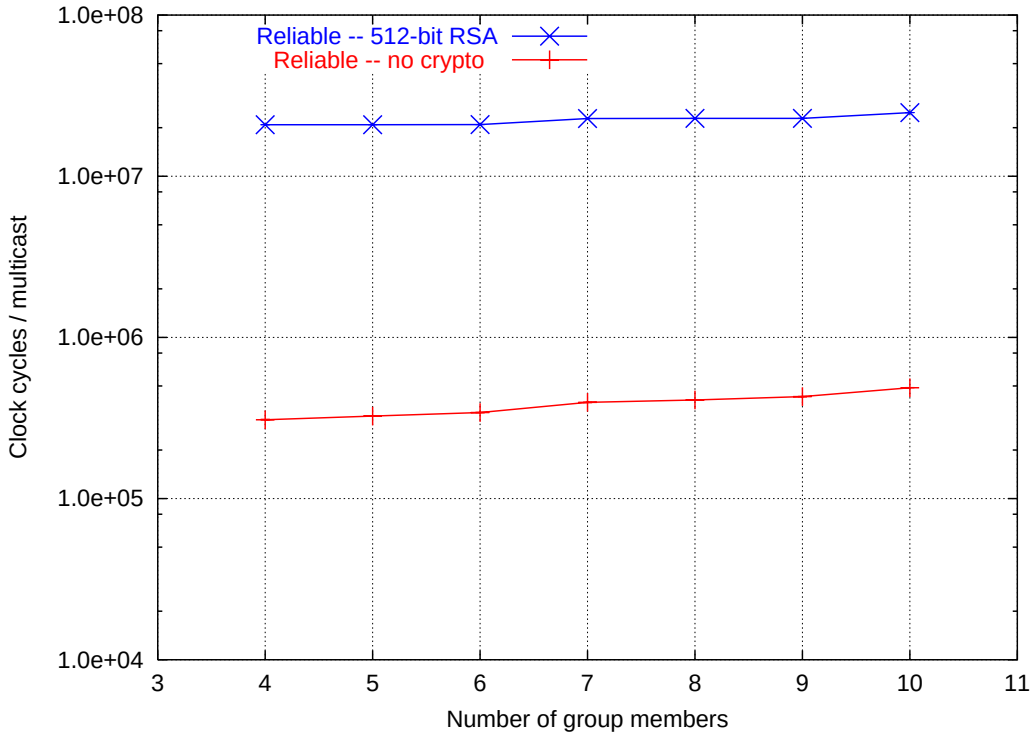


Figure 4.7: Application R: Cost of cryptography in *reliable*

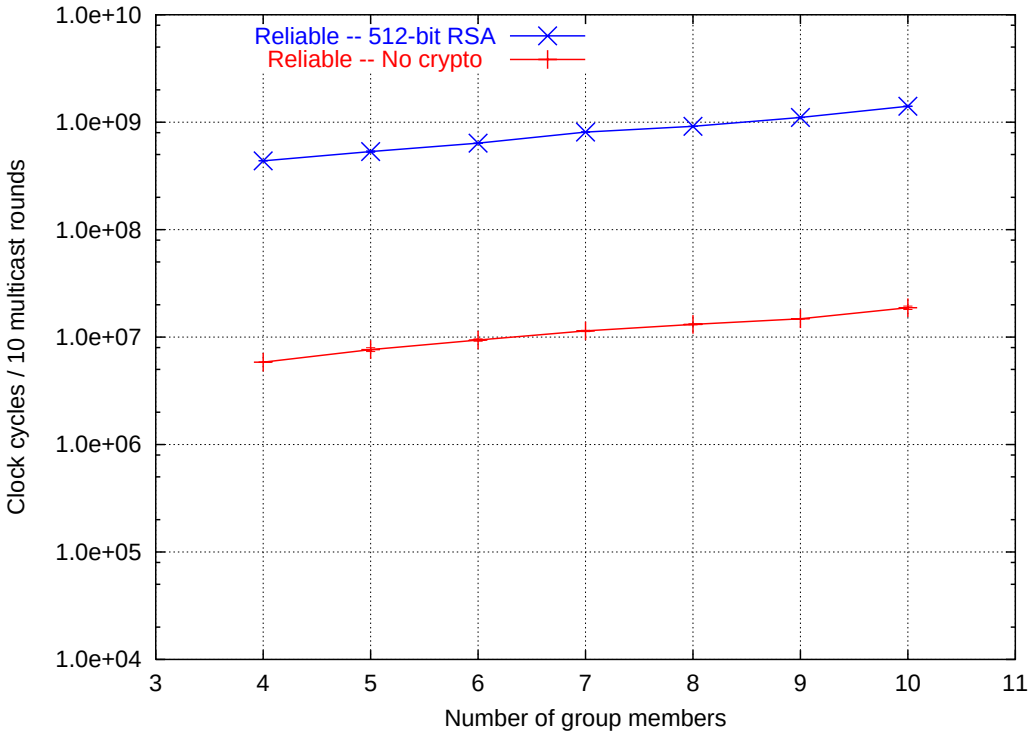


Figure 4.8: Application S: Cost of cryptography in *reliable*

Chapter 5

Conclusions and Future work

5.1 Conclusions

In this research, we have designed and implemented a total ordering protocol and a reliable multicast protocol for an intrusion-tolerant group communication system. This effort has led to a working prototype GCS that has been tested with multiple applications. A variety of design and implementation issues have been faced and dealt with. Various attack scenarios that would affect the messaging protocols have been considered. We tested the ability of the protocols to withstand such attacks by making subsets of process groups faulty and checking to make sure that the protocols detect and remove the corrupt processes. Performance measures have been made on the prototype intrusion-tolerant GCS to study the cost of using such systems.

Our performance measurements confirmed the notion that the capability to tolerate malicious faults does not come without cost. But it is also clear that the most significant factor in the high cost is public-key cryptography. This gives us hope that with specialized hardware and faster machines, the costs can be brought down far enough that a large class of applications can choose to have intrusion-tolerance capabilities.

As we built the core protocols of the GCS, it became clear that adding layers and providing new properties was easier once there were well-defined properties to build upon. Writing intrusion-tolerant distributed applications also becomes much simpler when the lower-level abstraction provides properties like reliable delivery and total ordering. One wider goal of our research group is to combine these ideas with replication and provide intrusion-tolerance properties transparently to applications. We feel that a intrusion-tolerant group communications system can be a very good building block for middleware solutions which address that issue.

5.2 Future Work

One important aspect of building distributed systems that has not been addressed so far in our work is that of formal proofs. We believe our protocols to be correct, but distributed protocols are notorious for subtle errors. One major hurdle in this direction is the difficulty of classifying the set of possible "bad things" that can happen in a system.

We would also like to make our reliable multicast protocol less expensive by tackling the cost of public-key cryptography. This could be done by exploring other methods of non-repudiably signing messages or by using a combination of public-key and symmetric cryptography.

Addition of dynamic timeouts to the reliable multicast and total ordering layers to improve efficiency and to detect congestion conditions would also require substantial work. One of the challenges of such extensions would be to ensure that the safety of the system is not affected by any changes made.

Another interesting question related to the total-ordering protocol is that of sequence number generating functions. Dynamic generating functions which change during are an interesting subject for future work.

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 Delivery Ordering Options, 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.

[CL99a] Miguel Castro and Barbara Liskov. A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. MIT/LCS/TM 590, MIT Laboratory of Computer Science, 1999.

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

[CLP+01] Michel Cukier, James Lyons, Prashant Pandey, Hari Govind 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 B–64 – B–65, Göteborg, Sweden, July 2001.

[CRS+98] Michel Cukier, Jennifer Ren, Chetan Sabnis, David Henke, Jessica Pistole, William H. Sanders, David E. Bakken, M. E. Berman, David A. Karr, and Richard E. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, October 1998.

[CZ85] David Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

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

[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*, 3, Nov.–Dec. 1999.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[Gro95] Object Management Group. *The Common Object Request Broker*, July 1995.

[Gut01] Peter Gutmann. *Cryptlib Security Toolkit*, April 2001.

[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.

[LBS+98] J.P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D.A. Karr, R. Vanegas, and K.R. Anderson. QoS aspect languages and their runtime integration. In *Proc. Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, Pittsburgh, PA, May 1998. Springer Verlag.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MMSA+96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, 1996.

[MMSN99] Louise Moser, P. Michael Melliar-Smith, and Priya Narasimhan. A fault tolerance framework for CORBA. In *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, pages 150–157, Madison, WI, June 1999.

[MNS+01] Brian Matt, Brian Niebuhr, David Sames, Gregg Tally, Brent Whitmore, and David Bakken. Intrusion tolerant CORBA architectural design. Technical Report 01-007, NAI Labs, April 2001.

[NIS95] The Secure Hash Algorithm (SHA1). NIST FIPS PUB 180-1, National Institute of Standards and Technology, April 1995.

[PWL00] Partha Pal, Franklin Webber, and Joseph Loyall. Intrusion tolerant sytems. In *Proceedings of the IEEE Information Survivability Workshop (ISW-2000)*, Boston, MA, October 2000.

[Ram] Hari Govind V Ramasamy. A group membership protocol for an intrusion-tolerant group communication system. Unpublished.

[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] 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.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[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.

[WMB99] Thomas J. Wu, Michael Malkin, and Dan Boneh. Building intrusion tolerant applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, 1999.