© Copyright by James Patrick Lyons, 2003

A REPLICATION PROTOCOL FOR AN INTRUSION-TOLERANT SYSTEM DESIGN

BY

JAMES PATRICK LYONS

B.S., University of Pennsylvania, 2000

THESIS

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

Urbana, Illinois

To my Parents and Family

Acknowledgments

In writing this thesis many people have provided me with invaluable support in one form or another. I am forever indebted to you all. Specifically, I would like to thank my advisor Prof. William H. Sanders for guiding and encouraging me through this research and the beginnings of my graduate career. Dr. Michel Cukier has also helped me immensely with his many insightful suggestions about my work.

My fellow research students Prashant Pandey, Harigovind Ramasamy, Vishu Gupta, Sankalp Singh, and Fabrice Stevens 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 for many interesting discussions and for their contributions to the ITUA project. I would like to thank Tod Courtney for his tireless work in helping to debug implementations and build demos. Mouna Seri was instrumental in completing the implementation of this work. Her wonderful patience and hard work in helping me learn the AQuA framework and ACE, as well as in helping me implement and debug significant portions of the initial code, made this work possible.

I am grateful to the other members of my office community, Ryan Lefever, Kaustubh Joshi, Dave Daly, Graham Clark, Sudha Krishnamurthy, and Salem Derisavi for making the PERFORM research group of CSL at the University of Illinois a great place to spend my time. I am also very thankful to Jenny Applequist for her help at all times and, in particular, in editing this thesis.

I am grateful to the Defense Advanced Research Projects Agency for funding the ITUA research project. The research described in this thesis was funded by DARPA grant F30602-00-C-0172. In particular, I would like to thank Dr. Jaynarayan Lala, Program manager of OASIS, for his guidance and support of the ITUA project.

Finally, I would like to thank my close friends, both local and afar, for helping me stay focused and sane while working on this thesis. In particular my roommate and co-worker Ryan Lefever has been a great friend and helped me to stay on track at times when I might have given up. Maria Jimenez has been a wonderfully supportive and understanding friend throughout the process. Her patience and support during the many late nights and

stressful times was invaluable. Lastly, my parents have been integral in my reaching this goal successfully. To them, I am and forever will be, grateful for all they have done on my behalf.

Table of Contents

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Previous Related Research 3
1.2.1 ITTC: Intrusion Tolerance through Threshold Cryptography 3
1.2.2 Adaptive Quality of Service for Availability 4
1.2.3 Intrusion-Tolerant Distributed Object Systems 4
1.2.4 Immune 4
1.2.5 MAFTIA 5
1.2.6 ITUA 5
1.2.7 Group Communication Systems 5
1.3 Intrusion Tolerance by Unpredictable Adaptation 6
1.3.1 The ITUA Environment 6
1.3.2 The ITUA Intrusion Model 7
1.3.3 ITUA Architecture Overview 9
1.4 Research Contributions and Thesis Organization 12

Chapter 2 Sender-Based Majority Voting Replication Protocol **14**
2.1 Infrastructure Overview 16
2.2 Replication Protocol Assumptions 17
2.3 Protocol Description 18
2.3.1 Step 1 19
2.3.2 Step 2 24
2.3.3 Step 3 25
2.3.4 AQuA Comparison 26
2.4 Correctness 27
2.4.1 Liveness and Validity 28
2.4.2 Reliability and Total Order 30
2.5 Fault Reporting 31
2.6 Problem of Group Size 33

Chapter 3 Performance Results **36**
3.1 Experiment Setup 36
3.2 Results 37

Chapter 4 Conclusions and Future Work **42**
4.1 Conclusion 42
4.2 Future Work 43

References **45**

Chapter 1

Introduction

1.1 Motivation

The Internet has fundamentally changed the way that businesses and governments store and process information critical to their continued functioning. Both governments and industry have been increasing their reliance on large-scale distributed systems of computers to handle ever-increasing workloads and potentially sensitive information. Computer networks are now so pervasive that these critical systems have become part of large-scale networks, such as the Internet, thus increasing their accessibility and vulnerability. On the one hand, this has improved the kinds of services offered by governments and industries. However, it has also made it easier for malicious attacks to be conducted against these same agencies by hackers, other governments, terrorists, or competitors. The practice of building large-scale systems from commercial off-the-shelf components is also becoming increasingly common. These COTS components often have well-known bugs or security flaws that only make them more vulnerable to the same kinds of malicious attacks. This problem is exacerbated by the fact that many of these components cannot easily tolerate failures of subsystems due to faults or attacks. The great economic and strategic importance of such systems has led to increasing interest in ensuring their "survivability."

[EFL+99] defines *survivability* as a system's ability to fulfill its designed function, within reasonable time bounds, in the presence of attacks, failures, and/or accidents. Research in survivability focuses on attack tolerance, detection, recovery, and adaptation. Related fields, such as computer security, have focused on building systems in which access by unauthorized entities is denied. Despite many years of such effort, almost all existing systems have vulnerabilities that have been or could be used to compromise a system in one manner or another. Computer systems tend to be very fragile in that once parts of them start to fail as a result of malicious intent of an intelligent subversive agent, they can very quickly become

unstable at best, and nonfunctional at worst. The addition of "intrusion tolerance" to these systems is one way of addressing this potentially disastrous problem.

Intrusion tolerance can be defined as a system's ability to continue to perform its essential functions, perhaps in some degraded but correct way, despite significant fractions of it having been compromised and placed in the control of an intelligent and malicious adversary [DBF91]. Intrusion tolerance somewhat resembles traditional fault tolerance, in which the focus has often been on building systems out of redundant components in such a way that component failure can be tolerated by a system without causing system failure. Fault tolerance research assumes that failed components cannot be expected by other components to perform according to their specifications. Likewise, when a system component is corrupted by an adversary, the corrupted component should no longer be trusted by the rest of the system. For that reason, concepts from fault-tolerant computing apply in intrusion-tolerance research as well.

One reason intrusion tolerance is more difficult to achieve is that the simplifying assumptions that are often made in fault-tolerance research, such as classification of failures as *fail-stop*[1], are not applicable. Further, additional techniques in fault tolerance, such as primary-backup fail-over, typically require the presence of identical systems acting as backups in case of a failure. In such a system, however, chances are good that the primary and backup systems share identical vulnerabilities. That points to the first major difference between fault tolerance and intrusion tolerance. In fault tolerance research, the assumption that failures are independent can sometimes be hard to justify; in the case of network intrusions, it is certainly not true. Great care must be taken to ensure that techniques from fault tolerance that were built upon notions of independent failure are adapted so that they hold under the new conditions. Therefore, in order to build intrusion-tolerant systems, certain cryptographic concepts, such as encryption, digital signatures, authentication, and secret-sharing, are used as well. Intrusion tolerance is therefore the combination of several fields of current research, including fault tolerance, networking, traditional computer security, and cryptography.

One can attempt to provide intrusion tolerance at several levels of a system: within the applications themselves, as a middleware component that applications use, or at the hardware level itself. The middleware approach is a popular one, because it has the allure of being able to abstract the complexity of providing intrusion-tolerant services (such as remote method invocations) into a neat package that more than one application can use to build survivable systems.

[1] A fail-stop failure is a failure in which a component, upon failing, immediately ceases all interaction with other system components.

As mentioned above, in order to build intrusion-tolerant systems, it is also necessary to use certain cryptographic concepts, such as encryption, digital signatures, authentication, and secret-sharing. There are several projects currently exploring the use of fault tolerance, cryptography, and computer security concepts for achieving intrusion tolerance. The next section will outline current research directions and detail some related projects.

1.2 Previous Related Research

Intrusion tolerance research strives to provide some level of confidence that systems can continue to provide useful services to users despite the presence of malicious attackers. Intrusion-tolerant systems generally attempt to protect at least one out of the following three properties:

Confidentiality Data is shared only with authorized entities.

Integrity Data being stored or viewed in the system, and provided to or written by users, is correct and has not been altered or destroyed.

Availability Service desired by authorized entities is available when it is needed.

Intrusion-tolerant systems research projects differ in their views of precisely which properties are most important. Therefore, the projects' focuses are different depending on which properties need to be provided in the face of intrusions.

1.2.1 ITTC: Intrusion Tolerance through Threshold Cryptography

This Stanford-based project [WMB99] emphasizes confidentiality and integrity, and utilizes secret-sharing as a primary technique in achieving those goals. The researchers have provided a library of tools and developed applications that utilize cryptography to keep information in the system private. The researchers try to maintain the secrecy of the keys by never reconstructing them in any location within their system. Their tools are designed to allow others to develop applications using their system.

1.2.2 Adaptive Quality of Service for Availability

This project, undertaken at the University of Illinois and BBN [Ren01, CRS+98], provides a framework for allowing CORBA[2] applications to specify levels of fault tolerance at run time, and for providing those levels of fault tolerance to the application via a replicated object architecture. The team constructed a gateway based on the group communication system Ensemble [Hay98, Hay01], which was developed at Cornell University. They used Maestro [Vay98] (also developed at Cornell) on top of Ensemble to build a gateway for intercepting CORBA IIOP messages to fulfill the project's goal of providing a transparent replication framework for use with CORBA objects. This work was extended in the ITUA project to include support for intrusion tolerance, and is the topic of this thesis.

1.2.3 Intrusion-Tolerant Distributed Object Systems

The ITDOS project [MNS+01, DSB02] is another project that is trying to build a distributed object middleware for providing intrusion tolerance to CORBA applications in a heterogeneous environment. The intrusion tolerance is accomplished by replication of objects that use CORBA. The ITDOS team is developing a modified CORBA ORB for use in their system. They based their system on group communication fundamentals and use a slightly modified version of the Byzantine Fault Tolerant multicast protocol developed at MIT by [CL99].

1.2.4 Immune

This effort [NKMMS99] is an approach for providing survivability to CORBA applications that is similar to other distributed object approaches. The team has used a secure reliable multicast group communication system [KMMS98] as the basis for a middleware layer that sits below a standard CORBA ORB (VisiBroker). Within that middleware layer, they manage the interaction among replicated objects in a way that is transparent to the application, using a combination of majority voting and active replication techniques. The active replication techniques used in Immune are fundamentally different from those presented in this thesis, in that no cryptography is used to ensure correctness at the replication protocol level. That means that more messages between groups are necessary. Currently, cryptographic operations are expensive to compute, but with the advent of specialized hardware, faster processors, and more efficient algorithms, the savings in message complexity may be

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

significant. The Immune system also has no obvious support for the state transfer that is needed to ensure the consistency of new replicas that may need to join the system in order to maintain desired levels of replication.

1.2.5 MAFTIA

This European collaboration [VNC00] is an effort to design a large-scale framework for both intrusion tolerance and prevention, and supports a wide variety of messaging paradigms, including invocations (request-reply), dissemination (push-pull), and transactions for multiple operation encapsulation. The design incorporates the idea of a *Trusted Timely Computing Base* [VCF00], which is a small portion of the system that is completely trusted by the rest of the system and whose correctness can be more easily verified than that of an entire large complicated system. The TTCB can be relied on to provide correct responses to a small set of queries within reasonable time bounds. The MAFTIA team used a well-specified hybrid fault model to design their system and proposed their model for use with other systems that might be developed in the future.

1.2.6 ITUA

The Intrusion Tolerance by Unpredictable Adaptation (ITUA) [PWL00, CLP+01] project, which is being undertaken as a joint project of BBN Technologies, the University of Illinois, Boeing, and the University of Maryland, is an 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 adaptation mechanisms that produce responses that are unpredictable to attackers, in an effort to confound their ability to execute scripted attacks against the system. This approach requires a fairly sophisticated replication architecture in order to maintain replica consistency. The ITUA system model, environment, and architecture are described in detail in Section 1.3.

1.2.7 Group Communication Systems

One of the major research areas that form the basis for much of the work being done in intrusion tolerance research is group communication system 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. Today there are several other systems based on the concept of

process groups, 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 group communication to develop higher-level abstractions to support reliable distributed computing. Some projects that are particularly relevant to the research described in this thesis are outlined below.

The ITUA and ITDOS projects use replication on top of group communication systems to achieve survivability. Replication of server objects is a technique used commonly in fault tolerance to achieve redundancy. In intrusion tolerance research, replication can help to increase the availability of the system, if the replicas are running on different components of the distributed system. However, replication can be harmful to the goal of confidentiality if information to be protected is known to each replica.

1.3 Intrusion Tolerance by Unpredictable Adaptation

In the following sections, we will discuss the details of the ITUA project (of which this thesis is a part), including the assumptions made with regards to the system being made intrusion-tolerant, as well as the types of attacks that ITUA is engineered to tolerate. Many of these details are also available in [Wea01].

1.3.1 The ITUA Environment

The collection of nodes comprising the system to be made intrusion-tolerant is partitioned into a set of disjoint "security domains." A *security domain* provides some boundary that attackers have trouble circumventing. A security domain may be composed of either a single node or many nodes, depending on the system design. In ITUA, each host is usually its own security domain. An implied requirement of this configuration is that hosts do not share administrative privileges. A LAN with firewalls protecting it from other networks is an example of a multiple node domain. Security domains strive to maintain as little mutual trust as possible among them, and divisions between them should be protected by commonplace computer security mechanisms such as firewalls, authentication, and access control. Utilization of different operating systems, firewall implementations, and other networking services throughout the system can help to increase diversity, and hence help to ensure that different domains are not subject to the same set of security flaws.

Domains are always in one of two states: *uninfiltrated* or *infiltrated*. A domain is *uninfiltrated* when all processes running within the domain follow their design specification and all components in the domain have their correct privilege levels. A domain is *infiltrated* when some process in it stops following its design specification, or some component gains a privilege level allowing it to execute functions that it is not normally allowed to execute. Initially, all domains are uninfiltrated; an attack on a security domain is successful when the domain's state changes from uninfiltrated to infiltrated. It is assumed that once the attacker is able to infiltrate a domain, the resources of that domain are fully in the control of the attacker and that there is no way to regain control of the domain and return its state to uninfiltrated. For example, a domain that is a secure file server becomes infiltrated when an attacker gains the ability to write files that do not belong to him/her, whether by gaining root access or exploiting an operating system flaw.

The set of processes running in a domain is always subject to change. New processes may be started at any time and existing processes may be stopped. Processes may be started and stopped in response to detected anomalies or in the event that new services require new processes. A process can be in one of two states, either *proper* or *corrupt*. A *proper* process is one that is executing as expected. A *corrupt* process is a process that is not proper. A process may become corrupted when the domain in which it is executing becomes infiltrated. The key to providing intrusion tolerance within this framework is the replication of objects across security domains. Objects which are to be made intrusion tolerant are replicated on different security domains so that groups of replicas will not simultaneously become corrupted in the event that one domain is infiltrated. The set of these replicas for a single distributed object is referred to as a *replication group*.

1.3.2 The ITUA Intrusion Model

The *intrusion model* defines the collection of attacks that a system being defended by ITUA was designed to tolerate. An attack is *considered* if it is contained in this collection. The attacks considered by the ITUA intrusion model are therefore attacks for which ITUA has been specifically engineered, and that ITUA is best able to tolerate. The intrusion model attempts to balance the need for describing as many attacks as can be conceived (thus making successful tolerance prohibitively difficult) and describing only trivially tolerable attacks. It attempts to consider only the most likely kinds of attacks without leaving the worst attacks unaddressed. It defines, in an abstract way, subtle and important features of attacks.

An attacker's goal is to alter the functioning of the system so as to compromise either the system availability or its correctness. He/she can attempt do so by corrupting processes

at any level. He/she will continue to corrupt processes until the attack fails or the attacker reaches his/her goals. An attack might not be totally successful; if the attacker can only achieve some successes against the system, the attacks are considered to be *partially successful*. In a partially successful attack, the system continues to function, but in a measurably degraded manner. A totally successful attack will render the system unable to function correctly.

The basic attack action in the ITUA model is the infiltration of an uninfiltrated security domain. The attacker may perform the infiltration in any of a variety of ways, including by stealing a password, successfully installing a virus, or exploiting some system or protocol vulnerability. The model assumes that given sufficient time, an attacker will always succeed in infiltrating a given security domain. However, the model also assumes that the time required to do so is significant and that the process for doing so is not necessarily obvious. The fact that security domains do not share privilege means that successful infiltration of a single security domain does not make it easier to infiltrate another. The fact that a detected domain intrusion could trigger a system-wide reconfiguration makes the infiltration of additional security domains more difficult.

Because an attacker is assumed to have free reign over the resources and processes of an infiltrated domain, none of the processes running in an intruded domain are trusted, and all of them are considered corrupt. Any cryptographic information contained in the domain, such as the private key of the hosts, is no longer trusted, as it could be used by the attacker in future attempts to subvert the system as a whole. Process corruption can take many forms; it may be as simple as killing the process, or as complex as making subtle modifications to the executing binaries in an attempt to exploit holes in the implementation of other components of the system. If process corruption is detected, the domain is assumed to be infiltrated, and none of the processes within the domain are trusted, even if they have displayed no signs of corruption themselves.

Because the infiltration of domains takes a significant period of time and success at one domain does not provide the attacker with advantages in attacking further domains, the model assumes a notion of a *staged attack*. Infiltration of domains occurs in discrete steps or stages in a staged attack, in which domains are infiltrated one at a time. The maximum rate of the domain infiltration is limited due to the time it takes to infiltrate a single domain. When a domain is infiltrated, there is a chance that the infiltration will be detected. The chance is not fixed, but varies according to the intruder's actions. ITUA does not provide protection if all domains are simultaneously infiltrated. Such a catastrophic infiltration would mean that all processes in the system were under the control of the attacker and that no processes would be left to perform any adaptation. Therefore, ITUA also does not provide

protection if all domains are infiltrated without any of the infiltrations being detected until the last domain is infiltrated. The architecture does provide protection if less than a third of any process group is corrupted at any given time, and takes steps to ensure that the number of infiltrated processes does not exceed a third of the total membership.

The model's use of security domains assures that attackers must infiltrate domains before any process within them may be corrupted. Practical experience shows that if security measures, such as firewalls, have been configured correctly, infiltration of a security domain does take time. The model also makes the assumption that it is not possible for an attacker to instantly perceive which security domain members of a particular replication group are running in. This assumption is necessary to make the staged attack notion reasonable. ITUA makes some assumptions about the cryptographic routines that it employs. Digital signatures are assumed to be unforgeable, and private keys are not stolen from hosts in uninfiltrated domains.

1.3.3 ITUA Architecture Overview

The overall system architecture of ITUA is given in Figure 1.1. The middleware accepts requests from applications to start distributed object replicas that have particular intrusion tolerance needs. These object replicas are transparently replicated, and the middleware controls the number of replicas, the placement of the replicas within security domains, and the state consistency of replicas. The ITUA architecture supports both in-band and out-of-band adaptations. Inter-object CORBA communications are intercepted, and the applications response is potentially modified in so-called *in-band adaptation*. In-band adaptation mechanisms are overseen by the QuO [LBS+98] adaptive middleware. *Out-of-band adaptation* involves actions that require reconfiguration of system resources and management of certain infrastructure mechanisms, such as firewalls, regardless of the inter-object communications. In order to manage this out-of-band adaptation, the architecture utilizes aptly named *managers* to form a decentralized management system.

A manager process runs on every host in the ITUA system. Usually, each host is its own security domain. The *manager group* is the group of all managers in the system. Configuration decisions are made by the managers on the basis of the domain-specific information available to them and information received from other managers about their own configuration changes. Managers are responsible for two main functions: *security advising* and *replication management*. While performing those roles, the managers' behavior is always local, i.e., it involves resources associated with the security domain in which the manager runs and the host(s) within it. However, some decisions made by the managers can have

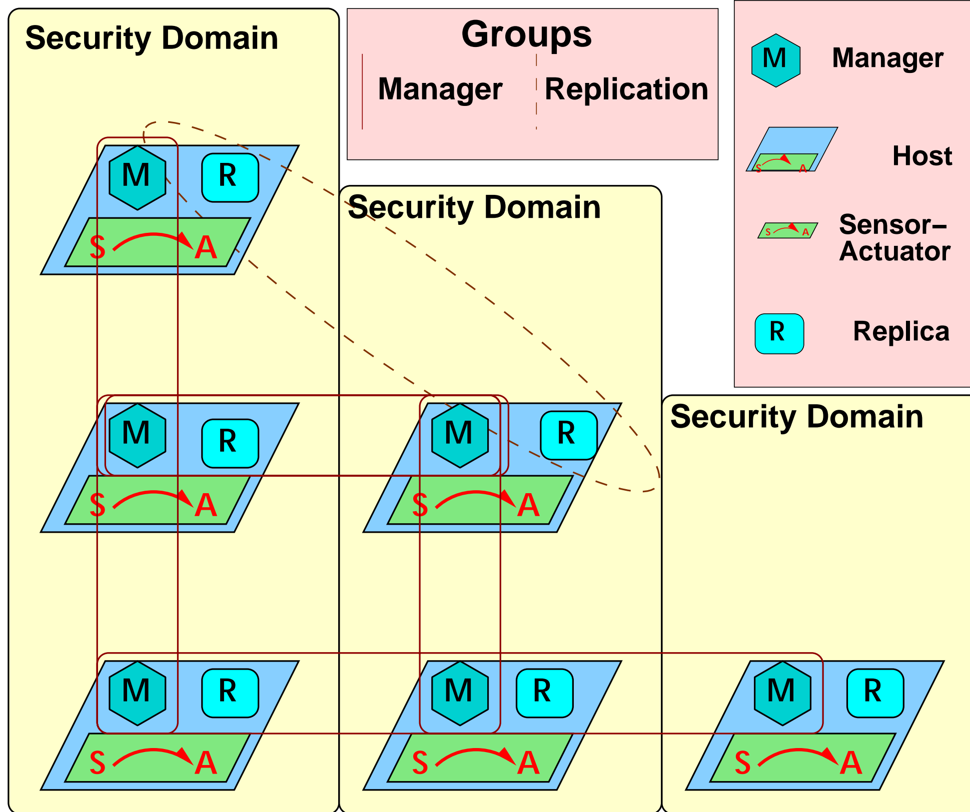


Figure 1.1: ITUA architecture

an effect on replicas running outside the security domain. For instance, if the tolerance level of a group is changed that decision affects all replicas in a group, which are spread across multiple security domains. Managers must therefore also cooperate in order to make such changes effectively. A manager's role as a security advisor makes use of multiple local *sensor-actuator* loops to monitor the system for intrusions and other potentially unanticipated events, to perform quick "knee-jerk" reactions to some specific events when they are observed, and to provide other managers with information regarding what is being observed within their domain.

All objects in the ITUA system are distributed across security domains by managers. The managers make these placement decisions in a distributed manner. One of a manager's primary functions in its replication management role is the starting or stopping of particular replicas on certain hosts. The manager, in its security advisor function, may determine that a host is under attack, and respond by modifying, the tolerance level of some groups that have replicas within that domain. Such decisions need to be coordinated with other managers, as the reconfiguration of the tolerance level of a group affects replicas in multiple

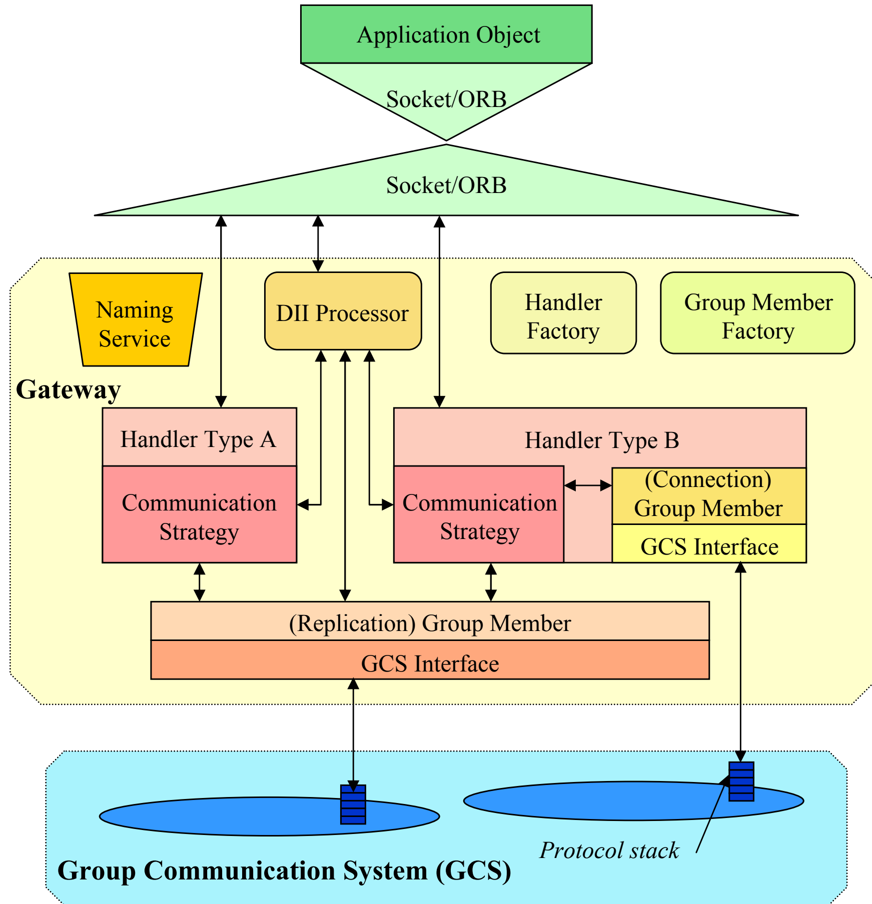


Figure 1.2: ITUA gateway

domains. Local actions by the managers can trigger the beginning of an agreement process to provide such adaptation.

ITUA also provides a framework for handling CORBA remote method invocations on distributed objects via the *ITUA gateway*. The gateway handles all aspects of the replicated objects' communication and is built upon an intrusion-tolerant group communication system in order to provide intrusion tolerance to the remote method invocations. The design of the ITUA gateway is shown in Figure 1.2. The gateway design originated in adaptive fault tolerance work by [Ren01]. The gateway has been adapted to allow multiple GCS systems to be used as the underlying layer provided that a new GCS Interface implementation is provided. Additionally, the gateway was extended so that it has a socket interface as well as a CORBA interface. The choice of which interface to use is part of the system design and depends on the needs of the application.

The ITUA gateway serves as a proxy for process-level communications. It provides a

framework for implementing different replication algorithms and reporting faults or anomalous behavior to the manager or subordinate on the host. The gateway can intercept standard IIOP[3] messages generated at each CORBA replica. The gateway manages all the communications to ensure that recipients of the messages receive each of them exactly once. The gateway intercepts and transmits these IIOP messages using the Dynamic Invocation Interface (DII) processor. A naming service is provided to allow the gateway to assign names to CORBA object ID strings, or IOR[4]. The gateway is designed to allow different handler objects to implement different replication protocols, and to allow the different handlers to be plugged into the gateway to change the tolerance and performance properties of the gateway. The handler implements a method for taking the requests or responses from a replication group and communicating them correctly to another replication group. Different handlers may have different intrusion-tolerance and performance characteristics, and therefore may need to be chosen with the objects to be replicated in mind. The gateway also allows for the easy study of different replication protocols by providing a well-defined framework for implementing them within this architecture.

The ITUA architecture contains several process groups, such as replication groups, connection groups, and the manager group. The groups have several functions in common, including maintenance of group membership information, point-to-point message delivery, and reliable message delivery. The fact that the common functions exist naturally suggests that a lower-level facility to provide the functions should be created. To provide this low-level facility, a new intrusion-tolerant group communication system has been specially developed for ITUA [RPL+02, Pan01, Ram01]. The group communication system is a fundamental component of the gateway (Figure 1.2) and is used to form the manager group, as well as the replication and connection groups.

1.4 Research Contributions and Thesis Organization

As outlined in the ITUA architecture overview, an intrusion-tolerant replication protocol is an important architectural component of ITUA. The research described in this thesis includes:

- The design of an active replication majority voting protocol for state machine replicas.
- The implementation and testing of the protocol within the ITUA gateway framework.

[3]IIOP is the Internet Inter-ORB Protocol, which specifies message format and relevant transmission syntax to allow independently developed ORBs to interoperate.

[4]IOR stands for "IIOP Object Reference."

- Performance and evaluation of the protocol using test CORBA applications.
- Presentation of extensions and possible future work.

The protocol ensures that requests and replies are processed by group members in the same order exactly once even if replication groups belong to multiple connection groups, and that no faulty or corrupt minority of group members is capable of subverting this property or indefinitely stalling the system.

Chapter 2 describes the functioning of the replication protocol and the properties that it provides to applications, and contains detailed algorithmic descriptions of the routines that were implemented and correctness arguments.

Important performance characteristics of the intrusion-tolerant replication protocol have been measured. These performance measurements give an idea of the performance overhead involved in using the intrusion-tolerant gateway to build applications that are intrusion-tolerant. More importantly, many techniques used in this replication protocol have also been used in other attempts at creating fault-tolerant systems using replication. Therefore, the performance measurements can be used to gain an insight into the most computationally expensive parts of an intrusion-tolerant gateway and to motivate research on optimizing them or looking for alternatives. The results of the performance measurements are described in Chapter 3.

Chapter 4 outlines the conclusions that have arisen from implementation of the protocol and analysis of the performance measures. It also provides thoughts on how this work could be extended in the future.

Chapter 2

Sender-Based Majority Voting Replication Protocol

In the past, replication has been accomplished via several different methods. One popular method is based on the notion of replicated state machines. In order to perform state machine replication, one must ensure that replicas start with the same initial conditions and maintain a consistent state while executing [Sch90]. The first requirement of such replication techniques is that the applications themselves be deterministic; that is, given identical inputs and starting states, the applications' outputs are identical. We assume the applications using this protocol are deterministic and therefore attempt to preserve the following two properties, in order to realize such replication:

- All replicated processes start with the same initial state.
- A replicated process processes all the same requests and replies in the same order.

To ensure initial state consistency, the ITUA system performs a state transfer to new replicas when they join the replication groups to which they belong. It currently does so by temporarily halting the processing of new messages to ensure that all state machines have processed the same messages. The application objects are then asked to collect state which will be provided to incoming replicas [Gup03]. The state is collected via the implementation of certain methods within the application that are called by the gateway. The gateway then collects its own state and sends this state and the application state to the new replicas, which install it through another specially implemented application method.

Replication protocols are the messaging protocols used to ensure that all replicas in a group process requests and replies in the same order. The complexity of these protocols arises from the fact that individual members of the group may belong to multiple groups. Therefore, each member may perceive a different order of delivery of messages from different groups. In the end, each member must process each message in the same order, so the

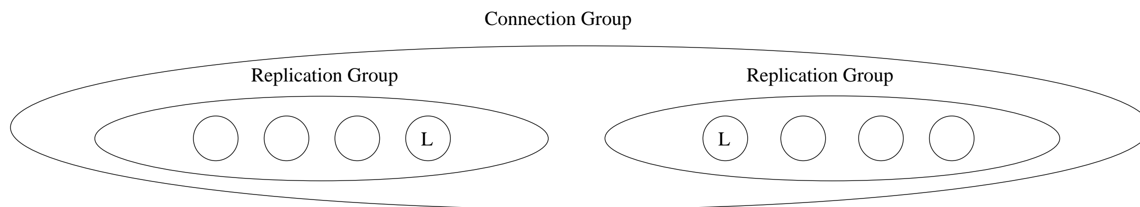


Figure 2.1: A sample of a connection group with 8 members

protocol must be able to arrive at a consistent ordering of messages at each member despite any initially perceived ordering. Ordering properties of group communication services are provided to messages sent within process groups, but not across them. The protocol presented in this section addresses the situation in which two separate replication groups need to communicate a remote method invocation and a reply to that invocation. Throughout the following discussion, *replication group* [RCS01] will be used to refer to a group of replicas that all belong to the same group in a group communication system, such as the ITUA GCS described in [RPL$^+$02, Pan01, Ram01]. Each replication group maintains a group leader. The leader has some additional responsibilities but by itself cannot incorrectly influence any other group members, or members of other groups, to perform operations. Any replica in a replication group is capable of executing the leader's role, and when a leader fails and is removed from the group, a new leader is designated automatically. Leaders are chosen as the process with the lowest replica ID in a group. A *replica* is a process that meets the deterministic requirements specified above, and is the process that is being made intrusion-tolerant via the system. A *connection group* [RCS01], as illustrated in Figure 2.1, is a group consisting of all the replicas of two communicating replication groups. The group communication system allows applications using them to generate suspicions and pass them down into the group membership protocol for processing and potential removal. This feature will be used throughout the course of this protocol to notify the GCS when faulty members have been detected at the gateway level. If enough members of the group suspect the same member, removal will be done via the group membership facilities of the group communication system.[1]

To help the reader understand the protocol more easily, we first describe the infrastructure components it uses, in order to make the subsequent protocol description clearer.

[1]The ITUA GCS requires $f+1$ suspicions to be generated in a group to trigger the beginning of the removal processes of a member, where the number of members is $3f+1$.

2.1 Infrastructure Overview

In the following sections, the two groups that make up a connection group are referred to as either "sending" or "receiving" replication groups. A *sending replication group* is the group that originates the request. A *receiving replication group* is one that receives the request and originates the reply to that request. The protocol presented in this chapter uses several message buffers to accomplish its goal of providing consistently ordered message processing to replicas. These buffers are:

Majority Buffer: A buffer used in the beginning of the protocol by non-leader gateways of the sending replication group to store messages that have reached a majority but have not yet been sent to the receiving replication group. It allows non-leaders to take over in the event of leader failure.

Majority Delay Buffer: Used to buffer messages sent from the leader before a replica in the sending replication group reaches a majority locally, and is ready to receive and process the messages.

Multicast Delay Buffer: Used to buffer messages sent in the last step of the protocol from the leader of the receiving replication group to the other members of the receiving replication group before those members are ready to receive such messages.

Total Order Buffer: Used by replicas in the receiving replication group to store messages that have arrived but have not yet been delivered to the applications. The leader then picks messages out of this buffer in the order in which they arrived to establish a total order. Other replicas maintain this buffer locally so that in case of a leader failure, any replica would be able to replace it.

The exact time when each of these buffers is used during the protocol is described in detail in Section 2.3. Here, we provide only a brief description of their use. The buffers are implemented as lists of gateway messages. They are searched linearly as needed for messages as the protocol is executing.

In addition to buffers, a few timers are also used to ensure that crashed or stalled gateways are detected and removed. A brief description of how these timers are used is given here to introduce the reader to the terminology. The details of when they are set and stopped will be discussed in detail in Section 2.3. The timers are the:

Vote Collection Timer: Set upon receipt of the first valid vote in the sending replication group; represents the length of time the vote will remain active. If the timer expires

before a majority is reached, the votes are discarded, and the vote is deemed to have failed to reach a majority.

Garbage Collection Timer: Used by gateways in the sending replication group to determine how long to keep a vote that has reached a majority in its buffers. After this timer expires, votes are discarded to prevent buffers from growing too large. Votes are stored after a majority is reached to allow the checking of outstanding votes that arrive after a majority has been reached. The checking process ensures that the late arriving votes agree even though the majority value has already been determined. The timer ensures that the gateways do not need to buffer the messages forever in the event a vote never arrives. Votes that arrive after this timer has expired are considered late messages, and are subject to suspicions and error reporting.

Multicast Latency Timer: Used by non-leader gateways to detect a leader failure. It is started when a majority is reached in the sending replication group and terminated when the connection group multicast is received. If it expires before the message is received, the group members suspect the leader, and the group member with the next member ID assumes the role of the leader and multicasts the message.

Rebroadcast Timer: This timer is used by non-leader gateways to ensure the timely rebroadcast of messages in the receiving replication group in the third step of the protocol. If this timer expires before the message is received, the leader is suspected, via a process similar to that followed after the Multicast Latency Timer expires.

2.2 Replication Protocol Assumptions

The replication protocol makes several assumptions about the underlying group communication mechanisms it uses:

- **Reliable Multicast** Multicast messages are delivered by all correct members of the group to the replication protocol exactly once.

- **Message Integrity** A group member that receives a message is assured that all other recipients of the message received the same message contents.

- **Message Authenticity** A message that appears to have been sent by member m was indeed sent by that member.

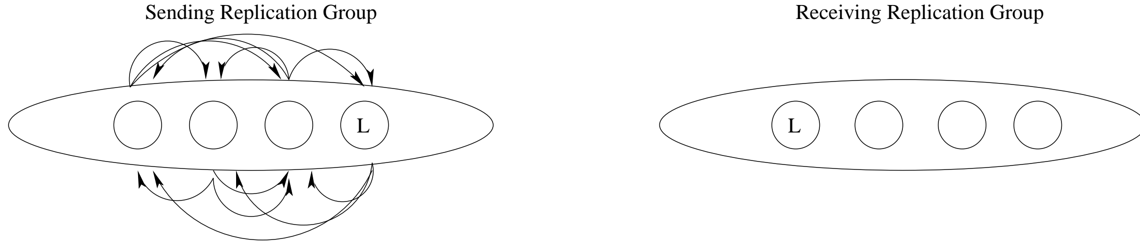


Figure 2.2: All replicas multicast to all others in the first part of Step 1

- **Totally Ordered Multicast** Any two messages received by a group member are received in the same order by all correct members in the group.
- **Reliable Group Membership** If a message m is received with group membership v, then all correct members of the group see the same v for that message m.

These assumptions can be fulfilled through the use of a suitable group communication system. A group communication system designed to tolerate the worst kinds of failures is assumed by the gateway protocol. The work described in this thesis used the intrusion-tolerant group communication system (ITUA GCS) developed at the University of Illinois [RPL+02]. The assumed properties can be provided by the GCS only if less than a third of the group is corrupted. For the system model considered in this research, that limit is actually a lower bound and is not a limitation of the GCS implementation [GM98]. Hence, it is also assumed that less than a third of any group in the system is corrupt at any given time.

In the next section, the protocol itself will be described in detail.

2.3 Protocol Description

The protocol can be described as a three-stage process. The first step (Figure 2.2) achieves a consensus in the sending replication group, by multicasting requests within it. As these requests are collected, the voting process is executed until enough votes have been received at each member to allow continuation to the second step. The second step (Figure 2.3) communicates the agreed-upon request to the receiving replication group; the leader sends a multicast within the connection group, which is made up of the two replication groups. Before the request is processed, the digital signatures collected in the first step are sent with the multicast so that they can be verified by members of the receiving group in order to ensure that a majority was indeed reached. When there may be more than one sending group, the third phase is necessary to totally order requests and replies in the receiving

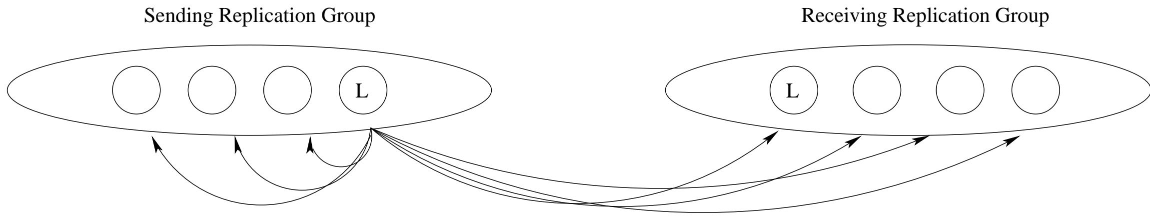


Figure 2.3: Leader multicasts in connection group in second part of Step 1

Figure 2.4: Step 2: Leader re-multicasts to others in replication group

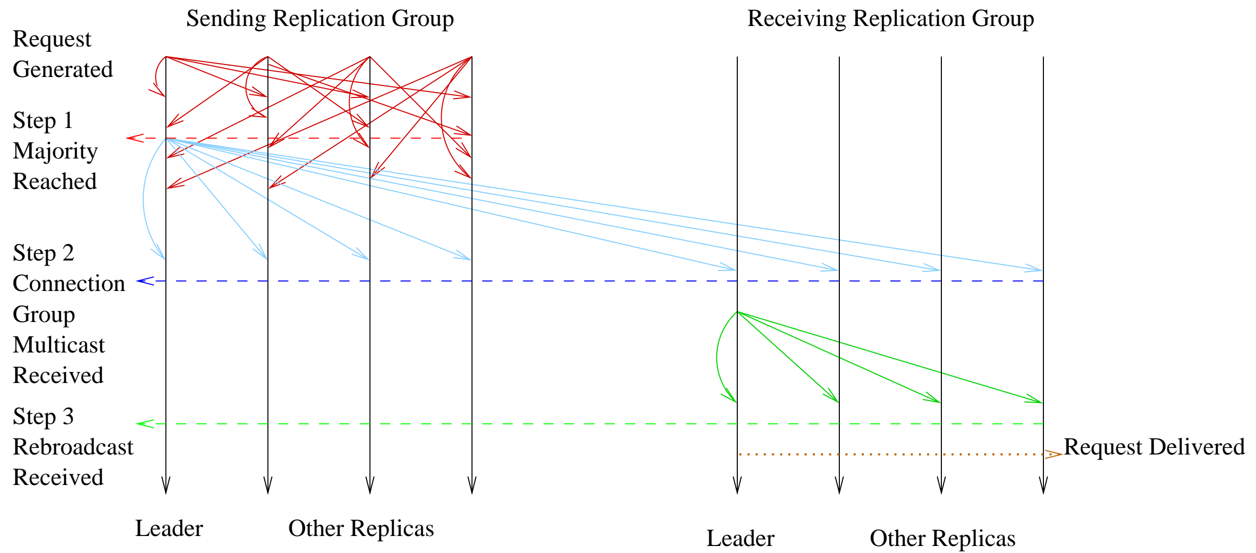
replication group, and is accomplished using the familiar sequencer approach (see Figure 2.4) [CM84].

Figure 2.5(a) gives an overall picture of the execution of the protocol. This figure assumes that a majority size of 2 is required before the request is sent out in the connection group, which is composed of all the replicas shown in the figure. The figure also assumes a particular ordering of events that illustrates the case where the majority delay and multicast delay buffers are not required. Figure 2.5(b) shows an execution path in which both majority delay and multicast delay buffers are required.

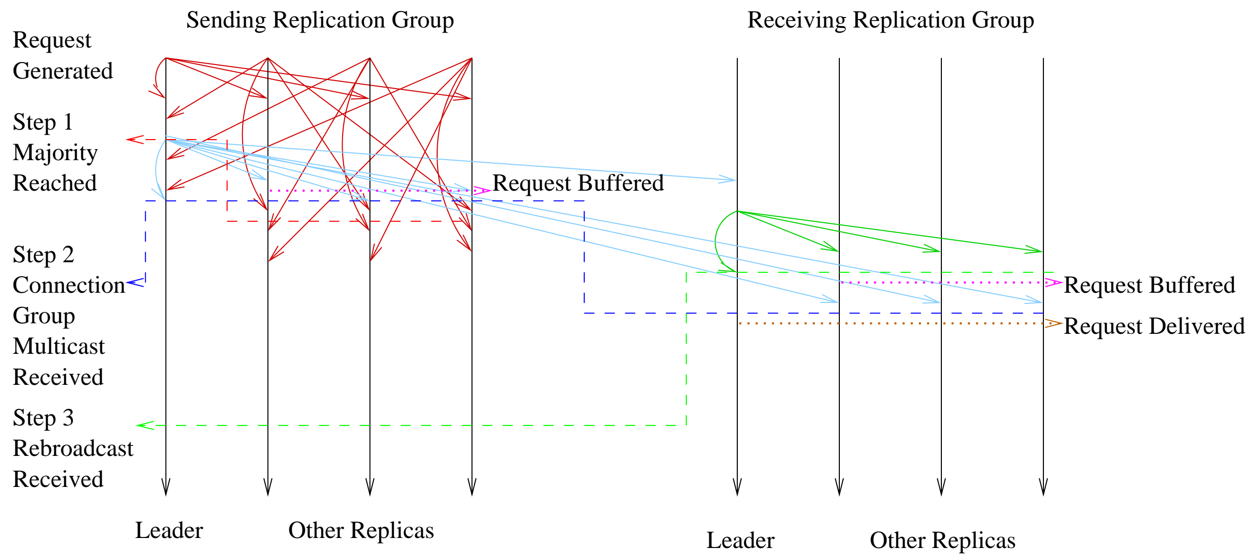
For the purposes of explanation, the three steps in the protocol process will be discussed one at a time in the following sections.

2.3.1 Step 1

An object in the sender group receives a remote object request via a CORBA IIOP message to the gateway. The gateway then assigns to this message the next available sequence number to identify the request. The gateway then dispatches the message to the gateway handler by calling Algorithm 1. Prior to being sent out in the replication group, each message is tagged 1) with the opcode FORWARD_REQUEST to indicate that it is in the first communication step, and 2) with the host's name (ID) to identify which replica sent the request. Additionally, a digest (such as MD5) of the message, along with the newly tagged data, is then computed and cryptographically signed using a public key signature algorithm (such as RSA). The message, along with the digital signature, is then multicast in the sending replication group. The first communication step continues as the gateway



(a) Normal protocol operation with no buffering required



(b) Normal protocol operation with buffering required

Figure 2.5: Illustration of protocol operation under differing message delivery orders

Algorithm 1: Pseudocode for msg arrival in Step 1

```
FromCorba(Message m)
(1)   m.opcode ← FORWARD_REQUEST
(2)   m.sender ← hostname m.signature ←ComputeSig(m)
(3)   ReplicationGroup.Multicast(m)
```

Algorithm 2: Pseudocode for msg arrival in Step 1

```
FromReplCastStep1(Message m)
(1)   if VerifySignature(m)
(2)       m.opcode ← CONNECTION_GROUP_REQUEST
(3)       StoreInListOfVotes(m)
(4)       CheckMajority()
(5)   else
(6)       ErrorReport(BAD_SIGNATURE, m.sender)
(7)       return
```

Algorithm 3: Stores votes in Step 1

```
StoreInListOfVotes(Message m)
(1)   foreach i in ListOfMessages
(2)       if i.equals(m)
(3)           i.voteCountInc(m)
(4)           i.AppendSignature(m.signature)
(5)           return
(6)   if m.sequenceNo > LastAddedSequenceNo
(7)       ListOfMessages.Add(m)
(8)       VoteCollectionTimer.Start(m)
(9)       LastAddedSequenceNo ← m.sequenceNo
(10)  else
(11)      ErrorReport(DUPLICATE_VOTE, m.sender)
```

collects votes sent from other replicas of the sending replication group.

The first action upon receipt of a vote (as outlined in Algorithm 2) is to verify the origin of the message. This can be done by simply verifying the digital signature and host ID information contained in the message. Upon receiving the first multicast message, a gateway starts the Vote Collection Timer (see Algorithm 3). If this timer expires before a majority is reached, the vote is aborted (see Algorithm 4).

Messages are then compared, bit-by-bit, to ensure that they represent the same request. If the comparison succeeds (see Algorithm 3), the digitally signed digest that accompanies the request is stored and appended to a proof, which will be sent out with the request when the message is multicast in the next step. If the message fails the verification process,

Algorithm 4: Pseudocode executes when Timer Expires on message m
VoteCollectionTimerExpires(Message m)
(1) $members \leftarrow ReplicationGroup$.getMembership()
(2) **if not** $m.majorityReached$
(3) ErrorReport(WRONG_VOTE, $m.senders$)
(4) $ListOfMessages$.Remove(m)

Algorithm 5: Stops timer when majority reached in Step 1
CheckMajority()
(1) **foreach** i in $ListOfMessages$
(2) **if** $i.voteCount > majoritySize$
(3) $VoteCollectionTimer$.Stop(m)
(4) MajorityReached(i)

an error is reported to the manager and a suspicion is generated in the GCS; this action may result in the removal of the sending group member. At this stage, it is possible to suspect a member on the basis of having received a message with an improper signature, even though the signature at this level is incorrect (and therefore tells the gateway nothing of the sender's identity by itself), because one of the assumptions made of the GCS is that it will authenticate the identity of the sender before it delivers the sender's message to the replication protocol. If the message verification succeeds, then the message's opcode is changed to one representing the second step in the process (FORWARD_REQUEST to CONNECTION_GROUP_REQUEST). The majority voting method (Algorithm 5) is then invoked to check if any votes have reached a majority yet. Majority voting occurs at this phase to ensure that a majority of the replicas have reached the same point and wish to invoke the same remote request on replicas in the receiving group. If a replica is the leader of its replication group, it is responsible for multicasting the request in the connection group (shown in Figure 2.3, and Algorithm 6) immediately upon reaching a majority.

If the replica is not the leader, there are two additional possibilities. The first is that the leader has not yet sent the connection group multicast, or more precisely that the non-leader replica has not yet received it. In that case, the message the replica expects to see is stored in the *majority buffer*. When a message is placed in the *majority buffer* the Multicast Latency Timer is started (see Algorithm 6). If this timer expires before the gateway receives the expected connection group multicast, the leader is suspected (see Algorithm 7).

Replicas of the sending replication group also start the Garbage Collection Timer (Algorithm 6) at this point. This timer will expire when the gateway does not expect to receive any more votes from any members (see Algorithm 8). Since the vote is sent in the connec-

Algorithm 6: Start Timers and send message at end of Step 1
MAJORITYREACHED(Message m)
(1) $GarbageCollectionTimer$.START(m)
(2) $m.majorityReached \leftarrow$ TRUE
(3) **if** ISLEADER()
(4) $ConnectionGroup$.MULTICAST(m)
(5) **else**
(6) **if** $MajorityDelayBuffer$.FIND(m)
(7) $MajorityDelayBuffer$.REMOVE(m)
(8) **else**
(9) $MajorityBuffer$.STORE(m)
(10) $MulticastLatencyTimer$.START(m)

Algorithm 7: Pseudocode executes when Timer Expires on message m
MULTICASTLATENCYTIMEREXPIRES(Message m)
(1) $Leader \leftarrow ReplicationGroup$.GETLEADER()
(2) GENERATESUSPICION(Leader)
(3) ERRORREPORT(LEADER_CAST_FAILURE, Leader)

tion group immediately after majority is reached, any remaining votes should arrive after the connection group multicast has been sent out. Those votes are compared to the majority value that was sent. Values that differ from the value already sent lead to suspicions. Votes that are missing when the timer expires indicate a failure to participate in a correct vote, and therefore produce suspicions for those members that did not vote. When the connection group multicast is finally received, the algorithm proceeds to the next step.

The second possibility is that the leader has already reached a majority and sent the connection group multicast before the non-leader replica reached a majority. If that happened, the request sent by the leader would have been buffered by the replica and stored in the *majority delay buffer*. To distinguish between the two cases, when the replica reaches this stage it checks the *majority delay* buffer to determine whether the request has already been

Algorithm 8: Pseudocode executes when Timer Expires on message m
GARBAGECOLLECTIONTIMEREXPIRES(Message m)
(1) **foreach** i in $members$
(2) **if not** $m.senders$.CONTAINS(i)
(3) ERRORREPORT(NO_VOTE, i)
(4) **if** $ListOfMessages$.FIND(m)$.voteCount < majoritySize$
(5) ERRORREPORT(WRONG_VOTE, $m.sender$)
(6) $ListOfMessages$.REMOVE(m)

Algorithm 9: Pseudocode for Msg arrival in Step 2
FROMCONNECTIONCASTSTEP2(Message m)
(1) **if** (VERIFYPROOF(m))
(2) **if** $m.source == myReplGroup$
(3) $MulticastLatencyTimer$.STOP(m) **if** REACHEDMAJORITY(m)
(4) $MajorityBuffer$.REMOVE(m)
(5) **else**
(6) $MajorityDelayBuffer$.STORE(m)
(7) **else**
(8) $m.opcode \leftarrow$ REPLICATION_GROUP_REQUEST
(9) **if** ISLEADER()
(10) $ReplicationGroup$.MULTICAST(m)
(11) **else if** $MulticastDelayBuffer$.FIND(m)
(12) $MulticastDelayBuffer$.REMOVE(m)
(13) DELIVERTOAPP(m)
(14) **else**
(15) $TotalOrderBuffer$.ADD(m)
(16) $RebroadcastTimer$.START(m)
(17) **else**
(18) ERRORREPORT(WRONG_PROOF, $m.sender$)

Algorithm 10: Pseudocode executes when Timer Expires on message m
REBROADCASTTIMEREXPIRES(Message m)
(1) $Leader \leftarrow ReplicationGroup$.GETLEADER()
(2) GENERATESUSPICION(Leader)
(3) ERRORREPORT(LEADER_CAST_FAILURE, Leader)

buffered. Storing messages in these buffers ensures that they will not be lost if the leader fails.

2.3.2 Step 2

The second communication step begins when the members of the connection group receive the message multicast by the leader of the sending replication group at the end of Step 1 (See Algorithm 9). Replicas of the receiving connection group start the Rebroadcast Timer upon receipt of the connection group message. This timer expires after the time allowed for the leader of the receiving replication group to rebroadcast the message in the replication group has passed (See Algorithm 10). If the leader fails to perform the rebroadcast in a timely manner, it is suspected by the other members of the group.

As outlined in Algorithm 9, the protocol proceeds as follows. Since each member of the

connection group receives the connection group multicast, the replica must first determine if the message came from a replica in the same replication group as itself, or a different one. It does so by checking the message headers. The digital signatures that make up the proof that accompanies the message are verified to ensure that a majority of the sending replication group members indeed generated the request. If the signatures that make up the proof fail to check out, or there is an insufficient number of them, an application-level suspicion is generated in the connection group, and the managers are notified. If the replica is in the sending replication group, the request is either stored in the *majority delay buffer* (if the replica has not yet reached a majority on that request) or removed from the *majority buffer* (because there is no longer a need to buffer it). If the replica is in the receiver group, the opcode is changed to reflect the fact that the message is now proceeding to the last step in the process. (In the implementation, that is reflected the transition from CONNECTION_GROUP_REQUEST to REPLICATION_GROUP_REQUEST.) If the replica is the leader of the receiving replication group, it acts like a sequencer by multicasting the message to the members of the receiving replication group, as shown in Figure 2.4.

If the replica is not a leader, it checks whether the *multicast delay buffer* contains the request. (A copy of the message would have been stored in the *multicast delay buffer* if the leader had received the connection group multicast message and re-multicast it in the replication group before the replica had received the message multicast in the connection group.) If the *multicast delay buffer* contains the message, the message is removed from the buffer and delivered. Otherwise the request is added to the *total order buffer*. The *total order buffer* permits message recovery if the leader crashes, or is intruded, before it has multicast a message in the replication group. If the leader should fail, the new leader can recover messages by processing them from the its local copy of the *total order buffer*.

2.3.3 Step 3

The third step begins when the message sent by the leader in step 2 is received by the receiving replication group (See Algorithm 11).

First, the sequence number is checked against the last delivered sequence number to make sure that the message has not already been delivered in the receiving replication group. If it has, the message is simply ignored. Otherwise, the message is delivered to the application. The last delivered variable is set to the sequence number of the just-delivered message. If the replica is not the leader, the *total order buffer* is then checked. If the request is in the *total order buffer*, it is removed, since the message has now been delivered to all replication group members. If the buffer does not contain the request, the request is added to the *multicast*

Algorithm 11: Pseudocode for Msg arrival in Step 3

```
FromReplCastStep3(Message m)
(1)   if (VerifyProof(m))
(2)      RebroadcastTimer.Stop(m)
(3)      if (m.sequenceNo > LastDeliveredSeqNo)
(4)         if (isLeader())
(5)            DeliverToApp(m)
(6)         else if (TotalOrderBuffer.Find(m))
(7)            TotalOrderBuffer.Remove(m)
(8)            LastDeliveredSeqNo ← m.sequenceNo
(9)            DeliverToApp(m)
(10)        else
(11)           MulticastDelayBuffer.Add(m)
(12)     else
(13)        ErrorReport(WRONG_PROOF, m.sender)
```

delay buffer.

The reply is accomplished via a precisely symmetric process (steps 4-6).

2.3.4 AQuA Comparison

The "Majority Voting" handler, which was designed as part of the AQuA system [RCS01], strongly resembles the protocol presented in this chapter. It is clear that the protocols share a great deal in their overall layout. The message buffering implementations are very similar. In contrast, however, the receiving replication group in the new protocol requires group membership information pertaining to the sending replication group in order to correctly establish the allowed "majority size" dynamically. In the new protocol, the concept of a majority is somewhat different from that in [Ren01]. In the sender-based majority voting replication protocol, a majority is defined by $f + 1$ votes, where f is the number of faulty replicas being tolerated in a group of $3f+1$ members. The major changes are the addition of message digests and digital signatures to the message payloads and their verification when members receive these messages, as well as the use of proofs to ensure the correctness of requests. The use of cryptography creates an additional difference from the voting protocol described in [Ren01], in that the replicas in both replication groups will need to have access to a public key infrastructure that allows the verification of messages sent to members of one replication group by members of the other replication group. The presence of such an infrastructure is also assumed by the GCS on top of which this gateway was constructed.

2.4 Correctness

The replication protocol described in the previous sections provides certain properties to applications using it despite the possibility that as many as f of the replicas act maliciously, where the number of replicas in a replication group is $n = 3f + 1$. The reader should be warned that although the algorithm might look deceptively like it would work if f of the replicas were faulty and $n = 2f + 1$ replicas were used, it would not. The GCS provides certain properties to the gateway handler that are assumed in the following correctness discussion. All of the properties can be provided by the GCS used here only if $n = 3f + 1$; and thus, by extension, this algorithm must assume the same bound on malicious processes. Otherwise the basic assumptions that the algorithm makes would become invalid.

As discussed before, the ITUA GCS provides applications such as the gateway with several properties:

Reliable Multicast Multicast messages are delivered by all correct members of the group to the replication protocol exactly once.

Message Integrity A group member that receives a message is assured that all other recipients of the message received the same message contents.

Message Authenticity A message that appears to have been sent by member m was indeed sent by that member.

Totally Ordered Multicast Any two messages received by a group member are received in the same order by all correct members in the group.

Reliable Group Membership If a message m is received with group membership v, then all correct members of the group see the same v for that message m.

It is important to note that total order is necessary to maintain consistent state in state machine replication architectures. The remaining properties are necessary to ensure the correctness of the replication algorithm. These properties are therefore assumed in the discussion below.

This replication protocol also provides properties to the applications that use it. They include:

Liveness No f or less members of a replication group may stall the protocol indefinitely, where $f = \lfloor (n-1)/3 \rfloor$, and n is the number of members in the group.

Validity No f or less members of a replication group may cause a request or reply to be processed by the other replication group in a connection group where $f = \lfloor (n-1)/3 \rfloor$, and n is the number of members in the group.

Reliability A remote method invocation made by a correct process will be delivered to all correct processes in the receiving connection group.

Total Order Remote method invocations are processed at all replicas in a receiving replication group in the same order.

Informal arguments for these properties appear in the following discussion.

2.4.1 Liveness and Validity

Liveness is ensured by our protocol through the use of timeout mechanisms to detect crashed or failed replicas. Validity is ensured through use of public key cryptography digital signature algorithms. To illustrate how liveness and validity are maintained, we will look individually at each communication step detailed in Section 2.3 to ensure that at every point in the protocol, the protocol continues to provide these properties.

In the beginning of the protocol, requests (or replies) are generated by each replica independently and votes are collected. At that point, each replica can do one of three things: (1) send a request (or reply) that agrees with the other replicas, (2) send a request (or reply) that differs from the other replicas, or (3) send no request (or reply) at all. It should be noted that requests or replies that arrive late (that is, the messages were delayed intentionally or accidentally) are considered here as special instances of case 3, because those messages will not be processed. Therefore, from the algorithm's perspective, the votes were never sent.

Case 1: This is the trivial case in which the replica does exactly what it is supposed to do, so that the protocol's behavior is unaffected.

Case 2: In this case the value generated by the replica will not be agreed upon, since less than a third of the replicas may generate faulty values. Therefore, not only will suspicions be generated at this step when the majority is reached and other votes are found to differ from the majority value, but the correct majority value will be passed on to following steps.

Case 3: In this case, the Vote Collection Timer is used to determine the time by which a replica must send a request. Since all members of the group are replicas, any valid request must be sent by a majority of members. If no majority is reached before the Vote Collection Timer expires, then the request must be a false one, and any replicas voting for such a false request will be suspected and removed by the others. If a majority is reached, and, after

some additional time measured by the Garbage Collection Timer, no vote has been received from a particular replica, that replica is suspected and removed for failing to generate a correct request. If, after the majority has been reached, a value arrives that differs from the majority value that has been sent out, the situation is treated as a special instance of Case 2.

In any of the cases, the protocol will either proceed to step 2 or decide that no correct request was generated. That means that Liveness and Validity are preserved at step 1, since correct requests will succeed and invalid ones will be suppressed.

Only when a majority of the members of the group agree on a request does the multicast in the connection group occur (Step 2). Three things can happen at this point: (1) the leader sends the message correctly, (2) the leader sends a message with a value different from the agreed-upon majority value in the connection group, or (3) the leader sends nothing at all.

Case 1: This is again the trivial case in which the leader does exactly what it is supposed to do. Therefore, the protocol's behavior is unaffected.

Case 2: The leader will not have sufficient proof and will not be able to forge the necessary signatures for its new message. In the sending replication group, the difference will be noticed immediately, and the leader will be removed from the group. In the receiving replication group, the proof of the message will be checked and will fail. The message will then be ignored. In this manner an incorrect request made by a replica that has been corrupted will not be processed by the receiving group.

Case 3: In this case, the other members of the group will notice the lack of action on the part of the leader as measured by the *Multicast Latency Timer*. If the timer expires before the leader sends the message in the connection group, the leader is thought to have been corrupted (or crashed). In that case the leader is suspected and removed via the group membership protocol of the GCS, and the replica with the next-lowest ID number assumes the role of the leader and sends out the request in the connection group. The new leader is capable of sending the request because it has the request in the *Majority Buffer*, where it remains on all replicas in the sending replication group until it has been received in a connection group multicast. In a truly asynchronous environment, that process may lead to the eviction of a correct but slow group leader. However, in practice, reasonable time bounds can be placed on message transmission times, so that the chance that a leader will be removed mistakenly will be remote. Further it should be noted that a corrupted member of the group cannot prevent the leader from sending its messages.

The proof-checking mechanism is used here to ensure Validity. Case 2 above is an especially critical one because it includes the case in which a single intruded leader tries to make the receiving connection group process a request that is different from the one agreed upon

by the sending replication group in step 1. The *Multicast Latency Timer* is used to ensure liveness. If a corrupted leader attempts to stall the protocol then it will be detected by the timer and removed. A correct process will assume the role of the leader, the request will be sent in the connection group, and the protocol will proceed to the last step.

Upon receipt of the connection multicast by the members of the receiving replication group, there are again only a fixed number of events that can occur: (1) the leader can rebroadcast the same message in the replication group, (2) the leader can rebroadcast a different message in the replication group (to attempt to change the value of the request/reply), or (3) the leader can send nothing at all.

Case 1: Again this is the trivial case in which the leader does exactly what it is supposed to do. The protocol's behavior is again unaffected.

Case 2: In this case again, the attached proof with multiple signatures prevents the message from being modified by the leader of the group. If the payload of the message does change, the signatures will not check out. If that happen, if the original connection cast message had correct signatures but the rebroadcast message did not, the leader of the receiving replication group will be suspected and removed.

Case 3: In this case, the other members of the group will detect a problem after the timeout measured by the *Rebroadcast Timer* has expired and the replication group rebroadcast has not been received. If the timer expires before the rebroadcast has been received, the leader is suspected to have failed in its duty to rebroadcast the message. The leader will be removed and a new leader will be established to rebroadcast the message, much as in case 3 of the previous step. The information necessary to being the leader is maintained on all replicas because of the use of the *Total Order Buffer*. As in case 3 of step 2, a message's failure to arrive could be the result of a slow replica. However, for the same reasons as before, it is reasonable to use a timeout here.

Validity in this step is again ensured via the same means as in step 2. The *Rebroadcast Timer* ensures that Liveness is preserved as well.

2.4.2 Reliability and Total Order

The next two properties are easier to explain because of the above discussion of the previous two. A request generated at a correct process will also be generated by at least $2f$ other processes (a total of $2f + 1$). This means that a majority is assured to have been generated at some point. Since the protocol is never halted, as discussed in the previous section, the requests are assured of reaching the sending replication group. The total order of requests is also a fairly straightforward property of the system. Given the use of a sequencer process in

the receiving replication group, all replicas will process requests in the order in which they are processed on the leader. That ensures that all replicas process the requests in the same order and thus that total order is preserved.

2.5 Fault Reporting

The previous discussions have described the nature of the protocol and how it works under normal operation. They also describe how the protocol tolerates the misbehavior of members in the system. Additionally, the gateway in which the handler runs is capable of notifying a management entity of any faulty behavior that it witnesses. It does so via a direct link between each gateway and a manager process.

Sections 2.3.1 – 2.4.2 briefly discussed the kinds of faults that are detected and reported. See those sections for details of how these fault detections fit within the context of the protocol. This section will discuss the various kinds of faults that are detected specifically and why their reporting is necessary despite the system's ability to tolerate them.

Briefly, the kinds of faults that are detected in the gateway are as follows:

- **No Vote:** Member fails to send a vote for a particular request or reply.
- **Wrong Vote:** Member sends a vote that differs from the value picked by the majority of members.
- **Duplicate Vote:** Member reuses a sequence number to send an additional vote for a request for which a vote has already been sent.
- **Invalid Signature:** The digital signature accompanying the vote does not verify correctly.
- **Invalid Proof:** The proof sent along with the request is either insufficient or incorrect.
- **Leader Cast Failure:** The leader of the replication group fails to send a message after it is supposed to in either the connection group or the receiving replication group.

The *No Vote* fault can mean one of two things. The machine running the host may have crashed or become corrupted in such a way as to stop sending messages, in which case removal should take place as soon as possible to return the group to a state in which all members are functioning properly. On the other hand, the machine in question may have fallen behind in its processing of requests and simply failed to keep up within the allowed timeout period. The timeout values are set to try to minimize the chances that that will

occur. In the event that it does and the machine is restarted successfully, the machine will be allowed back into the group using the same keys that it used before. The decision to allow or deny re-entry to the group is actually a group membership function and as such is dealt with by the managers and the group communication system and not by the handler directly.

The *Wrong Vote* fault represents the case in which a member of the group has sent a value that differs from the value that the group has decided upon as a majority. Such an event indicates that the machine has become corrupted to the point that its responses are no longer consistent with the specified protocol. When that happens the member is reported to the managers, and the managers should take care of removal. Such a failure is different from a *No Vote* in that a member convicted of such an offense is not allowed to rejoin the group using the same key. Only after a new key has been issued and the process restarted (potentially on another machine) is the replica allowed to rejoin. *Duplicate Vote* is handled in much the same way. If a replica has already received a vote from another replica or the vote has closed, the vote is discounted and the replica is reported. If a vote has closed, the replica has already been reported for a *No Vote* fault.

Duplicate Vote means that a message with correct signatures or proof arrives with sequence numbers that are old or already delivered. Such messages are ignored by the gateway, and the event is logged and reported.

Invalid Signature means that the signature accompanying the message does not match up with the message and public key of the sender. If that happens, it is assumed that communications with that replica have been compromised, and the replica will be reported immediately (and eventually removed, after others report it). We should point out that the replicas can be sure of the message's source, due to the properties provided to it by the group communication system.

An *Invalid Proof* is very similar to an invalid signature, because a proof is simply a collection of signatures supporting the message. The proof may contain too few signatures, or some signatures that do not verify correctly; in either case, the managers are notified, and the request is not executed. This ensures that only valid requests are executed in the receiving connection group.

Leader Cast Failure has occured if a leader of a replication group fails to perform its multicast operation. The leader may do so in two cases: when it sends a multicast in the connection group at the end of Step 1 (Section 2.3.1), and when the leader of the receiving replication group fails to send the multicast in the replication group (Section 2.3.2) within the allowed time bounds. A *Leader Cast Failure* is similar to a *No Vote* in that a timer is used by other members of the group to determine if the leader has failed. In order for the

protocol to make forward progress, the leader must be removed from the group, and one of the other replicas must assume the role of the leader.

The failure types presented above represent the kinds of failures that can be detected at the gateway handler level. Use of a group communication system, such as the ITUA-GCS, provides certain assurances that simplify the task of identifying some faults or intrusions. For example, *Invalid Signature* and *Invalid Proof* are easy to detect with certainty and reach quick consensus on, because all members of the groups in question receive the same, incorrect, copies of the messages and therefore reach the same conclusions. Timers are an essential part of the failure detection mechanisms and are also necessary to provide liveness guarantees, as well as being useful for performing garbage collection to maintain manageable buffer sizes.

2.6 Problem of Group Size

Throughout the above discussion, the notion of majority size was assumed to be obvious to the protocol. Indeed, if groups are of a fixed size, it is easy to determine the size of a group for the purposes of determining the majority size. However, when replicas are joining and leaving the groups, determination of the majority size is problematic, because the group size could change while a vote is in progress. Previous attempts to design majority voting algorithms have encountered similar problems [Ren01].

For example, consider two replication groups (A, B) in a single connection group (C). During normal operation, replicas in group B process requests from group A. One of the members of group A is corrupted. This corruption is detected and the member is removed. However, the replica was a member of two groups (A, C), both of which require removals, and both of which therefore require a notification of new membership. Such a notification is called a *view change*. However, the order in which the removal takes place is not known, nor is it known whether a view change notification will come before or after a vote has taken place. If replicas depend on the size of the connection group to determine the current majority size, a correct vote might not be processed. Due to the fact that nothing is assured about the relative delivery order of the replication and connection group view changes, situations like the following may occur.

A replica in group A is corrupted and removal begins. Group A generates a request r for group B and starts collecting votes for r. The replication group view change is received and the membership change means that the protocol is expecting one less vote in the sending replication group than it had been before the view change. Upon receiving this new, lower

threshold of votes, the leader sends r out in group C. The connection group view change message has not yet been sent in group C so the multicast from the leader of group A will arrive first. Therefore, the group size that group B maintains for group A is incorrect at the time it receives r. When r arrives, group B will perceive r to be short by one vote (one signature). That is interpreted as an improper proof, and the message is ignored.

Similarly, if the group size of one group increases by 1 (say in group A), the connection group view change is received first, and the view change increases the number of signatures that members of group B are expecting in the proof of r, once again the proof may fail to check out. Group A may generate a request r at the same moment the connection group view change is taking place, and send out the request in the belief that its own membership has not changed. (In fact, it appears to both groups A and B as if the other group's size has increased by 1.) The inappropriate request r again results in an incorrect number of signatures being sent with the message, and the request or reply is ignored.

The problem stems from the lack of a way for one replication group member to determine the size of another replication group even if both groups are in the same connection group. The group communication system has no way to distinguish connection groups from replication groups, and therefore is unable to make assurances about the relative ordering of view changes across them.

We handled the problem by assuming a fixed level of intrusion tolerance ahead of time. For example, to tolerate 1 possibly corrupt replica, the ITUA GCS requires that there be at least $(3f + 1) = 4$ replicas in the group. The replication protocol assumes a majority size of $f + 1$, and does not change that value during normal operation. That is, if group membership increased to 6 or 7 members, the handler would still be operating as if only 1 of its members could be corrupt at a time; thus, 2 votes would continue to be sufficient proof. That eliminates the problem that the basing of majority sizes on fluctuating group sizes presents to practical implementations of the protocol. The difference is that the level of intrusion tolerance or number of possibly corrupt members is specified ahead of time to the handler, and the system then runs with that parameter fixed. Ways to change this level of tolerance at runtime are currently being investigated.

A couple of different solutions to this problem are being investigated. One possible solution would be to modify the underlying group communication system so that connection group view changes always occur before replication group view changes. However, that sort of consistent ordering of view changes across groups would require significant amounts of modification. Another possibility would be to include the group size of the sending replication group as part of the request to be voted on. The solution is not quite that simple, though, because when votes are in progress, a view change may occur. That would change the group

size that needs to be included in the messages, and could lead to votes being considered different when they were not. To solve this issue, the replicas would have to return to the state just after the last completed vote before the view change was received and restart any pending votes again at step 1 to ensure correct agreement on group size.

Chapter 3

Performance Results

An important aspect of developing replication schemes, such as the one described in this thesis, is to determine how they perform in practice. In order to evaluate the performance of the SBMV algorithm described in Chapter 2, we instrumented the gateway handler code for timing experiment purposes. Section 3.1 describes the experiments that were conducted, and Section 3.2 shows the results of the experiments.

3.1 Experiment Setup

The experiments were conducted on a testbed of 1 GHz Intel Pentium III processors, each with 256MB of RAM connected via an isolated 100 Mbps LAN. The computers were each running Linux Version 7.2, and the gateway was run using Jacorb 1.4.1 as the ORB. The gateway code was compiled with gcc Version 2.96 and ACE Version 1.2.4. All cryptographic operations in the gateway and underlying group communications system were computed using the Cryptlib library [Gut01]. The machines were running a minimal load of supporting processes, such as xterms, and were otherwise unloaded.

In order to accomplish the timing measurements, we instrumented the gateway handler with additional code making use of the Intel x86 machine instruction, which makes it possible to read the CPU clock cycle counter. The timers then measured various intervals during execution, and the results were aggregated and converted to milliseconds for presentation here. All of the individual measurements taken were local intervals measured on either the client or a server. For example, to time a multicast, we started the timer when the multicast was sent, and stopped it when the multicast was delivered back to the handler. That is representative of the delay experienced in sending the message to other group members, however, it is not an exact measure of the time taken to send a message from one replica to another.

A test program called *Deet* [Rub00] was run to establish the amount of time it takes to execute a remote method invocation on objects replicated in the manner previously described. Deet works by making repetitive, sequential calls to a group of remote servers from a single client. The servers then perform a trivial calculation and return the result to the client. The calculations are quick to maximize the amount of time spent executing code in the gateway, and the group communication system, for each call initiated by the client.

One interval that the client gateway keeps track of is the elapsed time from when the request is first initiated until the reply to that request is delivered back to the application. It is called the *end-to-end delay*. It includes time spent in the ORB on the server, time spent sending broadcasts in the group communication system, and time spent on the server processing the request. The amount of time spent handling the request in the application, as mentioned above, was negligible for these experiments and is therefore ignored. Careful measurements were taken to account for the length of time spent executing in each component in order to determine the amount of overhead imposed by the replication protocol itself.

In order to more fully evaluate the performance of the algorithm under fault conditions, we also computed a few measures when faults were injected into the gateway using instrumented versions of the gateway handler with fault triggers compiled in.

3.2 Results

Normal Operation: The first set of experiments represents normal case operation. Deet was run with a varying-size group of servers and a single client. No faults were injected into the handler; 50 method invocations were sent, and each was measured individually in order to obtain the results presented below. The end-to-end delay was measured at the client locally as one aggregate timer. The C-Ensemble overhead was measured as the sum of the six delays measured, one for each multicast step in the round trip (three measures for the request and three more for the reply). The results of the first set of experiments, shown in Figure 3.1, show how the length of time to execute an RMI changes as the size of the server group is varied. Also shown is the estimated amount of time spent in handlers waiting for all the multicasts that the gateway sends to arrive. The time is called the *C-Ensemble Overhead*.

There are a couple of points on the graph worthy of mention. The first is that the vast majority of time spent in this algorithm is spent waiting for the group communication system to finish sending broadcasts. As seen in [RPL+02], the time to send a single totally ordered, reliable multicast is significant and increases significantly as the group sizes increase

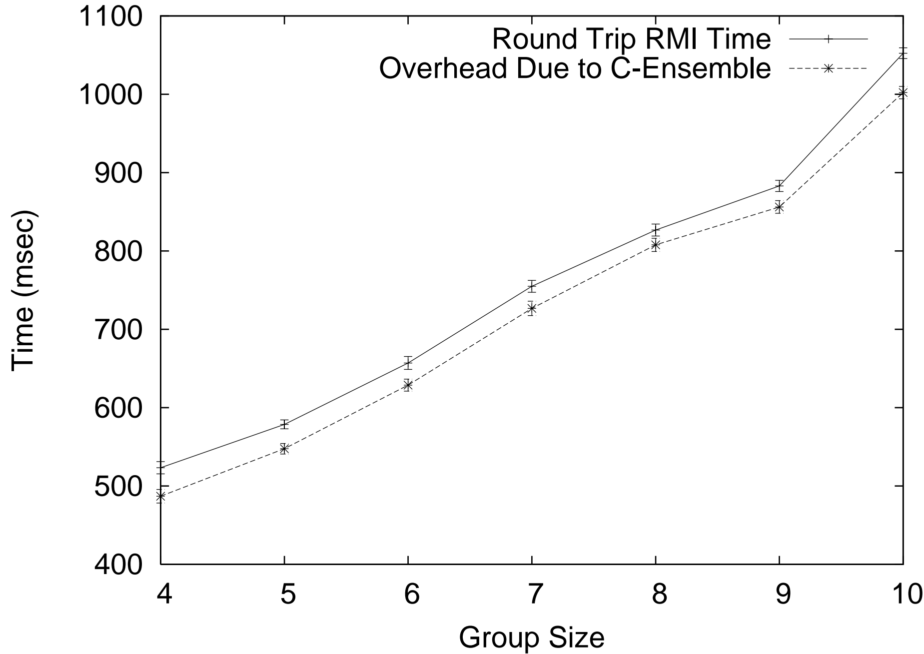


Figure 3.1: Graph shows the end-to-end delay per RMI in msec as a function of group size

Table 3.1: Data from fault-free execution

Group Size	Total RMI Delay (msec)	C-Ensemble Delay (msec)
4	523.3	486.9
5	578.6	547.5
6	657.0	628.8
7	754.9	726.7
8	826.7	807.8
9	882.9	856.2
10	1052.4	1002.1

to 7 and 10 replicas. This cost is compounded by the algorithm's need to send several such broadcasts throughout its operation. The GCS used tolerates an increasing number of faults as the group size increases above sizes 7 and 10. However, the increased tolerance comes at a large performance cost, which can be seen in the figure. Additionally, the total ordering protocol used in the GCS works well when all members of a group are sending messages in roughly equal proportions, but its performance degrades significantly when only one member of the group is sending messages. The degradation is primarily due to its use of a logical token ring ordering mechanism. A given replica may have to wait for $n-1$ null messages to be sent before a message it sent can be delivered. Each of those null messages, however, will be reliably delivered, which requires the computation of signatures and their verification

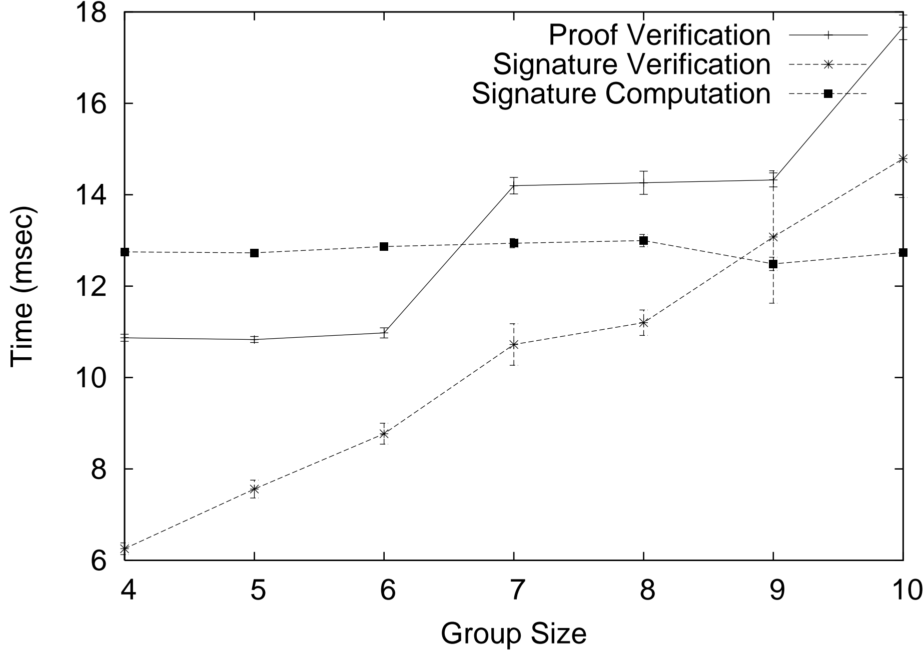


Figure 3.2: Time spent executing cryptographic operations as group size increases

in the GCS for delivery [Pan01]. About 2.2 ms of that gap is due to *ORB delivery delay*, which is the delay incurred at the server while a message to the application is being delivered and its reply is being generated and sent back to the handler. To alleviate congestion in the graph, we did not plot the delay, as 2.2 ms is not very significant compared to the scale of the other delays measured. The only other significant source of overhead, which is represented by the gap between lines in the graph in Figure 3.1, is time spent executing cryptographic operations.

Cryptography Costs: Cryptography accounts for a major fraction of the overhead imposed by the SBMV algorithm. In order to study just how large an impact the use of public key cryptography has on the gateway's performance, we also tracked the length of time spent calculating signatures and verifying them for each message during the course of the normal case operation experiments discussed previously. The results of the additional measures are presented in Figure 3.2. The graph shows that the cost of computing a signature is somewhat more computationally expensive than that of verifying a single signature. However, more time overall is spent in verifying proofs and signatures than in computing them. This suggests that a signature algorithm that allows the rapid verification of signatures would be even more useful in designing a protocol such as the one discussed in Chapter 2, even if the cost of computing such a signature is larger in comparison to the cost of verification. The

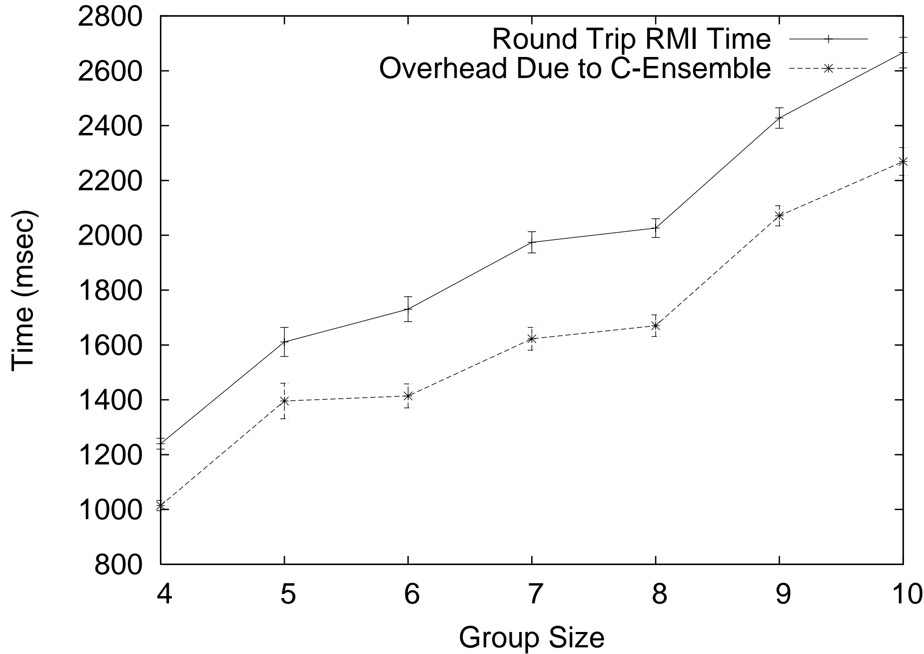


Figure 3.3: Time to perform RMI with a faulty leader as group size increases

steps seen in the graph for proof verification at group sizes 7 and 10 represent the group sizes at which an additional signature is required in order to provide the assurance of a correct majority vote for the group size. The signature verification plot is nearly linear since every signature that gets computed is verified at every node when it is sent out. This universal verification implies that the addition of one more member to the group requires that one more set of signatures be verified for each message.

Operation with a Corrupted Leader: The next case we studied was one in which the leader attempts to corrupt the message that is multicast in the connection group. We did the message corruption to simulate the case in which a server group's leader has been corrupted and is attempting to provide a false reply to the client. Such a modification to the result is detected by other members of the group via the proof-checking mechanism of the protocol. The client will simply ignore the reply when the proof fails to check out, and will await a correct reply from the server group. The members of the server replication group, upon receipt of the corrupted message, will generate suspicions that trigger a membership change in the group communication system. Upon completing that membership change, the new uncorrupted leader will send the correct message, along with a valid proof, in the connection group. The graph in Figure 3.3 shows that the time to execute an RMI for that is much greater than for the normal operation case. The increase makes sense because a membership

Table 3.2: Data from corrupted leader execution

Group Size	Total RMI Delay	C-Ensemble Delay
4	1239.9	1014.3
5	1611.0	1395.7
6	1730.6	1414.0
7	1974.4	1622.6
8	2026.2	1670.3
9	2427.9	2071.2
10	2666.4	2269.1

change is required in order for the reply to be received correctly at the client, which requires that an agreement be reached in the GCS. We did not directly measure how long membership changes take because it was difficult to locally measure the time when each server received the erroneous multicast message. However, one can infer the time from the graph by looking at the increase in size of the gap between the end-to-end delay and the C-Ensemble multicast overhead, as compared to the same gap in normal case operation data presented in Figure 3.1 and Table 3.2.

Chapter 4

Conclusions and Future Work

4.1 Conclusion

This thesis describes a protocol for replicating CORBA objects in an intrusion-tolerant manner. The protocol has been shown to be correct in the presence of malicious faults resulting from intrusions. We successfully implemented it within the existing ITUA gateway framework in order to build a working example to study. We then tested the protocol on top of working prototypes of intrusion-tolerant group communications systems developed as part of our research project [Ram01, Pan01]. We carried out performance measures to determine the cost of performing such replication.

The performance measures showed that the costs of performing intrusion-tolerant replication are heavily dependent on the underlying group communication system's performance. Although major costs are inevitable when providing intrusion tolerance, it is clear from the tests we conducted that a larger overall performance enhancement can be obtained from improving group communication system performance than from minimizing the overhead imposed in the gateway by the handlers themselves. Additionally, it can be concluded that when such expensive multicast primitives are used as part of a protocol design, communication comes at a great cost. Thus, protocols that use fewer rounds of communication may provide significantly better performance by avoiding the use of the group communication system altogether.

The use of cryptography in our protocol allows it to function by sending only one message per request in the larger connection group. As the results of [RPL+02] show, multicasting in larger groups can take significantly more time than multicasting in smaller ones. Therefore, while the cost of cryptography is not insignificant, the cost is somewhat offset by allowing the protocol to operate with less communication in the larger connection group.

The acceptability of those costs is dependent on the applications involved. It is obvious

from the results that one must be willing to sacrifice performance to gain levels of intrusion tolerance. The only way to determine whether the tradeoff is reasonable is to evaluate each application individually.

Our research is valuable to system designers in that it provides a clear notion of the cost of communication in a distributed environment such as the one used in our experiments. For example, it could encourage designers to build systems with very large computational loads in each RMI, thus rendering the overall cost of the intrusion-tolerant communications minimal in comparison. It also provides insights for future protocol designers by highlighting the need to minimize communication and optimize the performance of group communication systems.

4.2 Future Work

The correctness arguments provided in this thesis are informal. Formal validation techniques could be used to provide further assurance that the protocol is fully intrusion-tolerant. The formal validation of distributed systems is difficult and time-consuming, but the construction of large distributed software protocols is notoriously error-prone. Therefore, the use of formal techniques to ensure that system properties exist in implementations is an important step for future research.

Improvements in non-repudiable signature computation and verification would be very useful in systems such as ITUA. The protocol presented in this thesis, and the protocols in the group communication system developed in [Ram01, Pan01] make use of signed messages to accomplish their goals, and improvements in cryptography would affect the performance of all of those protocols.

Optimizations to the protocol as presented in this thesis are possible in special cases. For example, if a given pair of replication groups are communicating and each belongs to only one connection group, the third step of the protocol is unnecessary and could be skipped. Doing so would eliminate a (very costly) multicast. The current implementation would not allow us to skip the step, but it could be extended to let us investigate how such an optimization would change the overall performance. If the group communication system was capable of providing total orderings across groups, the third step would become unnecessary anyway. Investigating new group communication paradigms capable of providing orderings across groups would be useful research that could lead to the simplification of protocols such as the one presented here.

Work in providing consistent ordering of view change messages across groups would also

be both interesting and useful. The problem of group size as presented in Section 2.6 shows how the lack of such orderings is an unfortunate complication to a protocol such as ours.

Another area of further work that could be done would be the design of different protocols to perform intrusion-tolerant replication. Other protocols that potentially use fewer rounds of communication or less cryptography could be implemented, and compared to the current implementation.

Extending the system design assumptions to include operation in a WAN environment would also be a useful area for further investigation. Many practical systems tend to exist in wide geographic areas, so wide area networks are more likely to be the substrate on which such systems exist. The fact that there are military applications for intrusion tolerance research provides an additional motivation for conducting replication on wide area networks, since the type of physical damage that is an inherent danger to military systems would be better tolerated by spreading hardware among several geographically diverse locations.

References

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

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

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186, New Orleans, LA, February 1999.

[CLP+01] Michel Cukier, James Lyons, Prashant Pandey, Harigovind V. Ramasamy, William H. Sanders, Partha Pal, Franklin Webber, Richard Schantz, Joseph Loyall, Ronald Watro, Michael Atighetchi, and Jeanna Gossett. Intrusion tolerance approaches in ITUA. In *Supplement of the 2001 International Conference on Dependable Systems and Networks (Fast Abstracts)*, pages B–64 – B–65, Göteborg, Sweden, July 2001.

[CM84] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[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.

[DBF91] Yves Deswarte, Laurent Blain, and Jean-Charles Fabre. Intrusion tolerance in distributed computing systems. In *Proceedings of the IEEE Symposium on*

Research in Security and Privacy, pages 110–121, Oakland, California, May 1991.

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

[DSB02] et al David Sames and David Bakken. Developing a heterogeneous intrusion tolerant CORBA system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, Washington, DC, June 2002. To appear.

[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*, 3:55–63, Nov.–Dec. 1999.

[GM98] Juan A. Garay and Yoram Moses. Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SICOMP*, 27(1):247–290, 1998.

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

[Gup03] Vishu Gupta. Intrusion Tolerant State Transfer for Group Communication Systems. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[Gut01] Peter Gutmann. Cryptlib Security Toolkit. available at http://www.cs.auckland.ac.nz/ pgut001/cryptlib/, April 2001.

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

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

[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*, pages 303–318, Pittsburgh, PA, May 1998. Springer Verlag.

[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.

[NKMMS99] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 507–516, Austin, TX, May 1999.

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

[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.

[Ram01] Harigovind V. Ramasamy. A group membership protocol for an intrusion-tolerant group communication system. Master's thesis, University of Illinois at Urbana-Champaign, 2001.

[RCS01] Y. Ren, M. Cukier, and W. H. Sanders. An adaptive algorithm for tolerating value faults and crash failures. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):173–192, Feburary 2001.

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

[RPL+02] HariGovind V. Ramasamy, Prashant Pandey, James Lyons, Michel Cukier, and William H. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. In *Proceedings of the 2002 International*

Conference on Dependable Systems and Networks (DSN-2002), pages 229–238, Washington, DC, June 2002.

[Rub00] Paul Rubel. Passive Replication in the AQuA System. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[Sch90] Fred Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

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

[VCF00] P. Verissimo, A. Casimiro, and C. Fetzer. The Timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Distributed Systems and Networks*, pages 533–552, New York, USA, June 2000.

[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.