© Copyright by Harigovind Venkatraj Ramasamy, 2002

A GROUP MEMBERSHIP PROTOCOL FOR AN INTRUSION-TOLERANT GROUP COMMUNICATION SYSTEM

BY

HARIGOVIND VENKATRAJ RAMASAMY

B.ENGR., Anna University, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

To Amma and Naina.

Acknowledgments

I owe my sincere gratitude to my advisor, Prof. William H. Sanders, for guiding me through the research presented in this thesis, for his wise advice and support, and for teaching me so much. I thank Dr. Michel Cukier for so many insightful discussions and useful input regarding my research work.

My fellow ITUA project mates at the University of Illinois, Prashant Pandey and James Lyons, have been excellent people to work with. I thank them for our stimulating discussions on research and other interesting things in life.

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 their reviews of my work and for their contribution to the ITUA project.

I owe a big thanks to all the other PERFORM group members: Ryan, Kaustubh, Sudha, Tod, Mouna, Dave, Salem and Jenny for making this such a great research group to work in. In particular, I would like to thank Jenny for making this thesis readable, and for her help at all times.

My Amma and Naina (Tamil words for mom and dad, respectively) have been my greatest source of support, encouragement, love, and blessings. Words cannot express my gratitude and love for them. I thank my brother, Ashok Purushotham, for his encouragement and love and for helping me maintain good cheer at all times.

I am grateful to Dr. Jay Lala and the Defense Advanced Research Projects Agency for funding this research under contract F30602-00-C-0172.

Table of Contents

Chapter 1 Introduction **1**
1.1 Intrusion Tolerance by Unpredictability and Adaptation 3
1.1.1 The ITUA Environment 3
1.1.2 The ITUA Intrusion Model 4
1.1.3 ITUA Architecture Overview 5
1.2 Previous GCS Research 8
1.2.1 Ensemble 9
1.2.2 Rampart 9
1.2.3 Practical Byzantine Fault Tolerance 9
1.2.4 SecureRing 10
1.3 Research Contributions and Thesis Organization 10

Chapter 2 The Group Membership Protocol in the ITUA GCS **12**
2.1 System Model and Assumptions 13
2.2 Group Membership Protocol 14
2.2.1 Group Formation 15
2.2.2 View Installation to Remove a Single Corrupt Member 16
2.2.3 View Installation when Multiple Faults Occur 18
2.2.4 Detailed Protocol Description 20
2.3 Group Membership Protocol Properties 26

Chapter 3 Implementation Details **29**
3.1 The Ensemble Framework 30
3.2 ITUA GCS Protocol Stack 32
3.3 Infrastructure Implementation 34
3.4 Cryptography Details 34
3.5 Group Membership Protocol Implementation 36

Chapter 4 Performance Measurement **38**
4.1 Experimental Setup 38
4.2 Results for Group Membership 38

Chapter 5 Conclusions and Future Work **49**
5.1 Conclusions 49
5.2 Future Work 50

References **51**

List of Tables

3.1 The cryptographic support functions added to the infrastructure 35

4.1 Faults injected during a view installation to make it a transitional view . . . 41

List of Figures

1.1 ITUA architecture 6
1.2 ITUA gateway 7

2.1 Finite state automaton for view installation 17
2.2 The *suspect_message_handler* function 21
2.3 The *found_new_fault* function 22
2.4 The *received_newview* function 23
2.5 The *received_Commit* function 25
2.6 The *cast_NMC* function 25

3.1 Main components of the Ensemble layering model 30
3.2 Modified C-Ensemble stack 32

4.1 View installation times for single faults 40
4.2 Comparing gmp-reliable-total, gmp-reliable-total-dummy_crypt, & C-Ensemble stacks 42
4.3 Comparing gmp-reliable-total & gmp-reliable-total-dummy_crypt stacks for different faults 44
4.4 Comparing the view installation times for single and double faults 45
4.5 Variation of view installation time with load (single crash fault) 46
4.6 Effect of using time-outs for fault detection in transitional views 47

List of Abbreviations

GCS Group Communication System

GMP Group Membership Protocol

ITUA Intrusion Tolerance by Unpredictability and Adaptation

CORBA Common Object Request Broker Architecture

IIOP Internet Inter-Orb Protocol

DII Dynamic Invocation Interface

UIUC University of Illinois at Urbana-Champaign

UCSB University of California at Santa Barbara

LAN Local Area Network

WAN Wide Area Network

Chapter 1

Introduction

Computers, and computer networks that connect them, have become widely prevalent, in large part because of the Internet revolution. Every aspect of society, be it energy, transportation, business, defense, telecommunications, or manufacturing, has become increasingly dependent on large-scale, highly distributed systems built on large computer networks to gather, process, and distribute information. Some of these systems deal with military command and control, communication infrastructure, aviation control, and electric power control grids, and therefore are critical to national security. There has been increasing concern that a malicious attack on these systems could have catastrophic effects on national security and put many lives at risk. Hence, it has become important to ensure that these systems can *survive* such attacks. Ellison et al. [EFL+99] define *survivability* as the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents.

Computer security traditionally refers to a binary term, which identifies the state of a system as being either safe or compromised. It focuses on making the system as hard as possible to penetrate. In spite of the best security practices and preventive security measures, it is virtually impossible to guarantee that a system is invulnerable to attacks.

Survivability helps a system that has been compromised by an attack to recover, while continuing to provide its essential services. One way of achieving survivability is to equip systems with both *intrusion tolerance* and security mechanisms. An intrusion-tolerant system has mechanisms that allow it to continue to operate even when significant portions of it have been compromised and may be in the control of an intelligent adversary.

Traditional fault tolerance focuses on keeping the system operational in spite of *benign* faults, such as fail stopping or omission of some steps. It includes fault detection, diagnosis, masking, confinement, compensation, and recovery from faults. One common approach to fault tolerance is to use backup systems to withstand failure of the primary system. Intrusion tolerance also uses this type of *redundancy* to tolerate compromise of a subsystem.

However, if the backup systems are identical to the primary system, then they share the same vulnerabilities, and an attacker may exploit those vulnerabilities to penetrate both the primary and backup systems. Hence, redundancy alone does not guarantee intrusion tolerance. Redundancy needs to be augmented with ways to ensure that different subsystems are all not vulnerable to the same attack strategies. This could be done, for example, by following significantly different implementation methods for the primary and backup systems. Once an adversary has compromised parts of a system, he can make those parts behave in an arbitrary manner. Hence, the affected parts can no longer be trusted. That makes it necessary to employ authentication of all messages exchanged between the subsystems. For that purpose, intrusion tolerance draws concepts such as encryption, digital signatures, secret sharing, and authentication from the field of cryptography. Thus, intrusion tolerance combines ideas from fault-tolerant computing and networking, cryptography, and computer security.

Intrusion tolerance can be provided at different levels in the system: at the application level, the middleware level, the operating system level, or the hardware level. One promising approach is to provide intrusion tolerance at the middleware level, providing intrusion-tolerance services (such as remote method invocations) to distributed applications. Ongoing projects that aim to do this include MAFTIA [VNC00], ITDOS [SMN+02], and ITUA [CLP+01], among others. The MAFTIA project aims to develop a framework that ensures the dependability of distributed applications in the context of a broad range of faults and attacks. The project plans to achieve this goal by using dependable middleware, large-scale intrusion detection systems, dependable trusted third parties, and distributed authorization mechanisms as the building blocks of the framework. Both the ITDOS and the ITUA projects focus on building an adaptive, intrusion-tolerant middleware that can help CORBA[1] applications to tolerate malicious attacks.

The Intrusion Tolerance by Unpredictability and Adaptation (ITUA) project [PWL00, CLP+01] is a joint venture by BBN Technologies, the University of Illinois, Boeing and the University of Maryland. The ITUA project combines advanced redundancy management techniques (specifically countering faults resulting from a partially successful attack) with techniques that produce unpredictable (to the attacker) and variable responses to complicate the ability to preplan a coordinated attack and help achieve availability in the presence of malicious faults resulting from intrusions. In replicated systems, it becomes important to ensure the consistency of replicated information. This requires coordination of the replicas. The ITUA project takes a middleware approach to this problem by providing an intrusion-

[1] CORBA stands for Common Object Request Broker Architecture [Gro95], which provides a platform-independent way for applications to interact with each other.

tolerant middleware that provides certain key properties to help coordinate the replicas and achieve replica consistency.

1.1 Intrusion Tolerance by Unpredictability and Adaptation

We now describe the ITUA system by specifying the properties it aims to provide, the type of attacks it attempts to tolerate, the architecture of the system, and its associated environment. These details of ITUA are also presented in [Wea01].

1.1.1 The ITUA Environment

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

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

A host in a domain can have one or more application or system processes running on it. New processes can be started on the host, and existing processes may be killed. Hence, the set of processes running on the host is dynamic. Each process is in one of two states: *correct* or *corrupt*. A correct process behaves according to its specifications, while a corrupt process does not. A process can become corrupt as a result of its domain having become infiltrated. In order to provide applications with intrusion tolerance, we start multiple replicas of that application, which are distributed across multiple security domains. The replicas interact with each other and coordinate their actions to form a replication group. Once a replica becomes corrupt, it can never become correct again. To maintain the same level of intrusion

tolerance, we can remove the corrupt replica from the system, and start a new replica in a normal domain to take the place of the corrupt replica.

1.1.2 The ITUA Intrusion Model

The ITUA intrusion model describes the set of attacks that the ITUA architecture is interested in defending against. The model is a trade-off between including the set of all possible attacks (some of which certainly cannot be defended against) and including only simple attacks that the ITUA system can easily withstand. It is an attempt to include the most feasible attacks, while at the same time providing some defense against the worst possible attacks. We do not describe the intrusion model by listing all of the attacks. Instead, we identify certain abstract features of the attacks considered[2], and describe the intrusion model in terms of those abstract features.

The attacker's aim is to disrupt the normal functioning of the system. He can do so by corrupting processes. The attacker can continue to corrupt processes, until the attack has been repulsed or until the system's tolerance to corruption has been destroyed. An attack may be partially or completely successful. A partially successful attack, will at the very least, result in a degradation of the system's quality of service. A completely successful attack will result in the system's failure. A process can become corrupt only after the domain in which the process resides has been infiltrated. The attack primitive is to infiltrate a domain, which can be done by exploiting an operating system or application vulnerability, stealing a password, or introducing a virus, worm, or Trojan horse. Infiltrating a domain takes a non-negligble amount of time, and because of the heterogenity of domains, there is a limit on the number of domains that can be infiltrated simultaneously. That gives the defense some time to react to the attack. We call this the *staged* attack assumption, meaning that there is a non-negligble time between successive domain infiltrations. The ITUA model will not be able to defend against a situation in which all the domains housing the processes in a replication group are simultaneously corrupted. It also does not defend against a situation in which the attacker infiltrates the domains in stages, but the infiltrated domains show no observable signs of an intrusion until all the domains have been infiltrated. This is essentially the same as the first situation.

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

1.1.3 ITUA Architecture Overview

Figure 1.1 shows the basic ITUA architecture. The CORBA application specifies the intrusion tolerance requirements to the ITUA middleware, which then spawns the distributed application objects to satisfy those requirements. The existence of the ITUA middleware is totally transparent to these distributed CORBA application objects. From their perspective, they communicate with other distributed objects using remote method invocations, just as if the middleware were plain CORBA. In actuality, the inter-object interaction is intercepted and application-level behavior is altered by the Quality Objects framework within the ITUA middleware. This framework constitutes the in-band adaptation mechanisms for intrusion tolerance. It is complemented by out-of-band mechanisms, which involve intrusion response and recovery actions to manage and configure system resources independent of the inter-object communication. The ITUA architecture uses a decentralized infrastructure to implement such out-of-band intrusion-tolerance mechanisms. This decentralized infrastructure comprises special processes called *managers* and *subordinate managers* distributed across the security domains. We refer to the subordinate managers as *subordinates* in the description below.

Each host runs a manager or a subordinate. For each domain, there is one host that runs a manager. Every other host in that domain runs a subordinate. The subordinates in a domain and the manager in that domain form a process group called the *subordinate group*. The managers in all domains form a process group called the *manager group*. The subordinate performs two main functions: security advising and replication management. In its security advising role, the subordinate uses local sensor-actuator loops to collect information about potential intrusions and anamalous events, to perform quick *knee-jerk* reactions to such events, and to inform the domain manager of such event occurrences. The scope of these knee-jerk reactions is usually local, i.e., they involve resources on the subordinate's host. However, some cases (e.g., changing the security posture of a replication group) may involve cooperation of the managers across domains, and participation of multiple hosts across domains. In its replication management role, the subordinate makes decisions regarding the starting and killing of replicas. It will also communicate with other subordinates to reach agreement about which host a new replica should be started on, whether new replicas should be started to increase the intrusion tolerance of a replication group, and what security posture a replication group should change to, after an observed anomaly. A domain manager has all the host-specific responsibilities of a subordinate. In addition, it interacts with other domain managers to report anomalous events observed by itself or subordinates in its domain and to take decisions that can affect hosts across all the domains, like changing

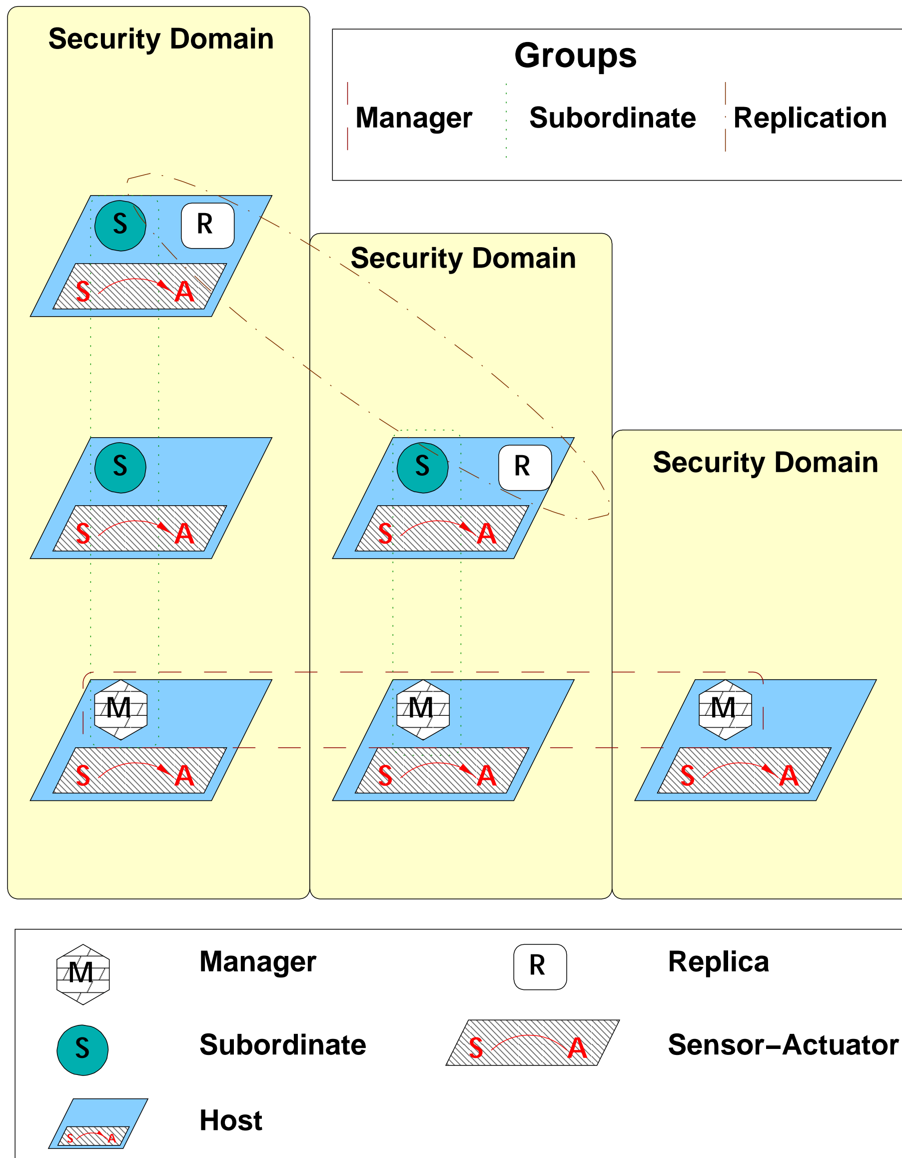


Figure 1.1: ITUA architecture

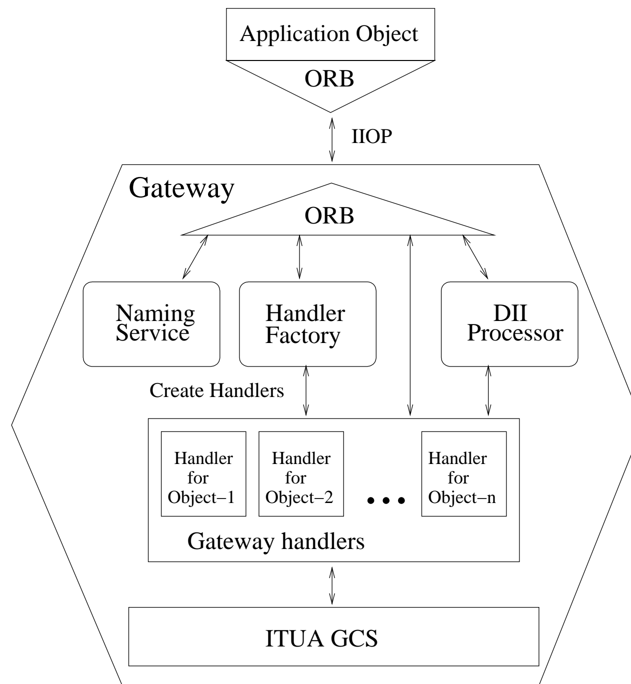


Figure 1.2: ITUA gateway

the security posture of a replication group.

The ITUA middleware, in addition to implementing the above-mentioned management functions that provide out-of-band intrusion tolerance, also has a component called the *gateway* that makes the remote method invocations of the distributed objects intrusion-tolerant. The various modules of the gateway are shown in Figure 1.2.

The gateway has intrusion-tolerant replication protocols and an intrusion-tolerant group communication system to make the object remote method invocations intrusion-tolerant. The manager group, subordinate group, and replication group are all basically process groups consisting of processes communicating with each other. Such process groups all require that the underlying system have some basic properties, like the ability to maintain consistent group membership and to provide reliable and totally ordered message delivery. Group communication systems (GCSs) are a convenient low-level abstraction for providing such properties to process groups. To provide these properties to the process groups in the ITUA architecture, we are developing a new intrusion-tolerant group communication system, called the *ITUA GCS*, as part of the gateway.

The gateway translates between the object-level messages at the CORBA application object and the process-level messages multicast by the ITUA GCS. The application object

generates IIOP[3] messages to communicate with other distributed application objects. These IIOP messages are intercepted by the gateway's dynamic invocation interface (DII) processor and sent to the handler for that object. The handler will then transparently communicate the messages to the recipient process groups. The handler also takes care that only a single response reaches the calling object. The choice of the handler to use for a particular object depends on the intrusion-tolerance requirements specified by the object, and those requirements determine how a single response reaches the calling object based on the multiple responses received from the replication group. Thus, from the calling object's perspective, communication with other distributed objects is the same as if plain CORBA (as opposed to the ITUA middleware) were being used. The gateway also provides an infrastructure for implementing various replication and voting schemes, and for detecting and reporting faults at the replication-group level to the respective subordinates/managers.

1.2 Previous GCS Research

Group communication is a well-known paradigm for providing consistency among the processes in a set. Several group communication systems have been built in the past, and they differ in the properties they provide to system builders. In this section, we focus on those group communication systems that are fault-tolerant or intrusion-tolerant. An early fault-tolerant group communication system was the ISIS system [Bir93, BvR94] from Cornell University. It led to other efforts for developing fault-tolerant group communication systems, like Horus [vRBM96] and Ensemble [Hay01] at Cornell University, Totem [MMSA+96] and SecureRing [KMMS98] at UCSB, and Transis [DM96] at the Hebrew University. There have been other projects that used GCSs to support dependable and reliable distributed computing. For example, the AQuA project [Ren01, CRS+98] at UIUC uses the Ensemble group communication system to build a middleware that can provide applications with the desired level of dependability in the presence of crash failures and value faults. The Eternal System [MMSN99] at UCSB uses the Totem group communication system to extend CORBA with object replication and fault tolerance. We now describe some group communication systems that are related to our research.

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

1.2.1 Ensemble

Ensemble [Hay98, Hay01] is a flexible group communication system that consists of a set of microprotocols. Each microprotocol provides a set of properties that may be required for a particular application. The system builder can choose the appropriate subset of the microprotocols and form a protocol stack to satisfy the application's requirements. Since Ensemble is a flexible, layered stack, it is easy to add new layers and provide new properties. Ensemble has many security features [RBD01], but it can tolerate only crash failures. One drawback is that Ensemble is implemented in the ML language, making it inaccessible to researchers who are unwilling to learn that language. A C implementation of the important microprotocols of Ensemble has been created by Mark Hayden, who was one of the original developers of Ensemble. That implementation is called *C-Ensemble*. We inserted new intrusion-tolerant microprotocols for providing group membership and reliable, ordered message delivery into the C-Ensemble framework.

1.2.2 Rampart

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

1.2.3 Practical Byzantine Fault Tolerance

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

for liveness. The protocol only provides group communication; it does not provide group membership services.

The protocol labels the server replicas with identifiers starting from zero. In a given view, one server replica (whose identifier is the view number modulo the number of replicas) acts as the primary; other replicas act as backups. A new view is installed when a two-thirds majority of the backups agree to mark the primary replica as faulty.

1.2.4 SecureRing

The SecureRing group communication system [KMMS98] provides reliable, ordered message delivery and group membership services despite Byzantine faults. The key protocols in the GCS are a message delivery protocol and a membership protocol. The membership protocol organizes the group members into a logical ring. Message multicast in the ring is controlled using a token; the process that currently holds the token can multicast a message. An unreliable Byzantine fault detector is used to report faulty processes to the membership protocol, which then reconfigures the system by forming a new ring consisting of apparently correct processors. The message delivery protocol ensures message delivery in a consistent total order to all members of the group. SecureRing's developers claim that their use of message digests in a signed token to allow a single digital signature to cover multiple messages makes their protocol more efficient than protocols in which all messages need to be signed.

1.3 Research Contributions and Thesis Organization

The ITUA GCS is an important component of the ITUA middleware. This thesis describes the research work that resulted in the design, development, and informal validation of a group membership protocol for the ITUA GCS. This group membership protocol provides consistent group membership to process groups in the presence of malicious faults resulting from intrusions. Chapter 2 describes the properties guaranteed by this group membership protocol, the assumptions under which the protocol functions, a detailed algorithmic description of the protocol, and an informal proof that the protocol satisfies the stated properties.

The group membership protocol has been implemented as a layer in the ITUA GCS, which is a layered protocol stack. Chapter 3 describes the protocol implementation details, the framework in which this layer was implemented, and the changes made to this framework in order to add the protocol.

Chapter 4 provides performance results for the group membership protocol. The protocol was instrumented to provide detailed information about the cost incurred during fault-free

operation and while tolerating both single and multiple correlated intrusions. To better understand what causes the overhead in each case, we instrumented the protocols in a way that allows us to isolate the overhead due to cryptography for the group membership protocol in the case of multiple simultaneous faults. These measurements allow us to know what portion of the overhead comes from the algorithms themselves, and what portion comes from cryptographic operations. The results are useful in that they provide new insights into the cost of providing intrusion-tolerant group communication, and suggest ways that this cost could be reduced in the future.

Chapter 5 outlines the conclusions we arrived at after implementing the group membership protocol and making performance measurements. We point out how to decrease the cost of tolerating malicious faults and provide ideas for extending this work.

Chapter 2

The Group Membership Protocol in the ITUA GCS

As discussed in the previous chapter, the ITUA architecture has different process groups, including a manager group, subordinate groups, and replication groups, to achieve the goal of providing intrusion tolerance to CORBA applications. All of the process groups share some common concerns, like ensuring that all correct processes in the group see the same group membership and receive the same set of multicast messages in the same order. These concerns reflect common problems in many application domains. Group communication systems have been developed to address these problems. A group communication system (GCS) provides a process group abstraction to a set of processes, and guarantees some important properties to the constituent processes. The type of guarantees provided by a GCS should fit the application's requirements. The GCS to be used in ITUA should provide consistent group membership to the correct processes in all process groups in the presence of malicious faults; corrupt processes need to be detected and removed from a process group, and new processes must be allowed to join the group.

The ITUA GCS must also provide reliable multicast guarantees to the processes in a process group. For example, a manager may multicast a message reporting a detected intrusion in the manager group. It is necessary to ensure that all managers receive this message. In general, multicast messages must be delivered to all correct processes in spite of faulty behavior by some corrupt processes. As indicated in Section 1.1.2, processes can become corrupt without detection. Such a corrupt process can attempt, for example, to multicast a message to only a subset of the group. It may also try, for example, to multicast two messages with the same sequence number, but with different contents, to two mutually exclusive subsets of the group. It is clear that in the presence of malicious faults, members of a group cannot blindly trust the contents of a message they get from the network. This

problem suggested that an intrusion-tolerant reliable multicast primitive should be provided in the ITUA GCS.

The ITUA middleware provides intrusion tolerance to CORBA applications by forming replication groups of the application object. Each of these application replicas maintains state information. When the replication group is first formed, all the application replicas are initialized to the same state. This state will change as the replicas process information. The replication protocols in the ITUA architecture use the state machine approach [Sch90], in which the final state after a set of messages has been processed depends on the order in which the messages were received by the replica. Since we want all the replicas in a replication group to be in the same state, it is necessary to ensure that messages are delivered to all the replicas in a replication group in the same order. A good way of providing this total-ordering property is through the GCS used by ITUA.

Thus, the ITUA GCS should have protocols that guarantee consistent group membership and ordered, reliable message delivery. The protocols for reliable multicast and total ordering are described in [Pan01]. This thesis focuses on the group membership protocol that has been developed for ITUA.

The rest of this chapter describes the intrusion-tolerant group membership protocol in detail. We also informally prove the correctness of the protocol by showing that it guarantees the properties it claims to provide. Before focussing on the protocol, we first describe the system model used by the protocol.

2.1 System Model and Assumptions

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

The protocols are concerned with one set of processes that wish to be in a group. The group membership protocol installs a series of views, V_0, V_1, ..., each of which is a set of process identifiers of processes that are members of the view. The processes in a single view

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

We use message digests and digital signatures based on a public key cryptosystem. Each process possesses a private key, public key pair. The private key of a process is known only to the process. The process uses its private key to sign messages digitally. Each process is able to obtain the public keys of other processes to verify signed messages. The protocols also use message digest functions, such as MD5, in which an arbitrary-length message m is mapped to a fixed-length output $d(m)$. This function takes the message envelope as the data.

Each process is either *correct* or *corrupt*. A *correct* process conforms to the protocol specification. A *corrupt* process can exhibit arbitrary behavior. The process group can continue to provide correct service if there are no more than $f = \lfloor (|V| - 1)/3 \rfloor$ corrupt processes. We assume that all processes are computationally bound. This means that a corrupt process cannot find two messages m and m' such that $m \neq m'$ and $d(m) = d(m')$; it cannot produce a valid signature of a correct process, or compute the message summarized by a digest from the digest. We assume that private keys cannot be stolen from correct processes.

2.2 Group Membership Protocol

The intrusion-tolerant group membership protocol of the ITUA GCS ensures that all correct processes maintain consistent information about the current membership of the group in spite of intrusions. It is responsible for maintaining group membership information, removing processes from the group, and joining new processes into the group. In providing this function, the group membership protocol relies on the reliable multicast protocol described in [Pan01] to deliver the messages it sends to maintain group membership.

We incorporated checks into several layers of the modified C-Ensemble protocol stack to detect deviations from the protocol specifications. Any detections of deviations are reported as *suspicions* to the group membership protocol, which then multicasts these suspicions

[1] The responsibilities will be explained in the rest of this chapter.

to the group. The group membership protocol provides an interface *suspect(process-rank i, reason R)* for this purpose. The suspicions may be mistaken and can differ between processes. If a process p_k has received enough[2] suspicions regarding a process p_i, then the group membership protocol at p_k acts to remove p_i from the group; it does so by initiating a view installation procedure, which is a series of steps at the end of which p_i will be removed from the group. Like the protocol in [Rei94b], our protocol follows a three-phase commit strategy for the view installation. However, while a view installation in [Rei94b] removes a *single* corrupt process or adds a *single* correct process to the group, each view installation that we describe here removes *all* corrupt members from the group.

2.2.1 Group Formation

When a process is created, it is given the identifier of the group it needs to join and the intrusion-tolerance requirements that the group should satisfy. The intrusion-tolerance requirements will be specified in terms of the number of malicious processes to tolerate. The cardinality of the group should be more than three times that number, before the group can start its application-level services. The process is also given a list of certified public keys. These constitute the public keys of processes that are allowed to be in that group. In the ITUA architecture, when a process group is to be created, the ITUA managers go through a consensus round, and decide at which hosts the constituent processes should be started. The managers will allow a new process to be started on a host in any security domain not suspected to have been infiltrated. Then the manager/subordinate in each of those hosts will create a public key, private key pair for the process that is to be started on that host. The manager/subordinate will also take care of getting the public key certified by a certification authority. The managers then agree on the public keys of the processes that can be part of the group. While responding to an attack, it is possible to add new processes to existing process groups to improve the intrusion tolerance, but new process groups will not be started. Hence, it is reasonable to assume that during the time of group formation, the constituent processes are correct.

When a process is created, it forms a singleton group. It will then unreliably broadcast a *gossip* message indicating its public key and the process group it wants to join. One of the constituent processes will be started with a singleton group identifer, that is the same as the identifier of the intended process group. This process will be the nominated leader process. When this leader process receives the gossip message, it will verify the message's validity by checking whether the public key is in the list of authorized public keys, and will form a new

[2]Later in this section, we will explain how many processes are *enough*.

view that includes the process that broadcast the gossip message. Subsequently, before the group formation is complete, if a valid gossip message is received by a non-leader process, it will forward the message to the leader, which will then form a new view that includes the process that broadcast the new gossip message. If the number of processes in the group has reached the number necessary to provide the specified intrusion tolerance, the process group will start providing its application-level services. After that point, processes in the group may become suspected for deviations from the protocol specification; this may result in their removal from the group.

To maintain the desired level of intrusion tolerance, it may be necessary to add new processes to the group to compensate for the removed processes. That would require an update to the list of certified public keys that each group member has. In the ITUA middleware, any decision to add a new process to an already existing group would involve agreement between the managers as to the domain and the host in that domain in which the new process should be created. The subordinate on that host would create the key pairs for the process, get the public key certified by the certification authority, and give this certified public key to its manager. The manager would then inform the other managers about the new process's public key, the group that the new process is intended to join, along with proof of consensus that this new process can join the intended group; the other managers would pass this information on to their subordinates. At a host on which some member of the process group is running, the subordinate will add the public key of the new process to the list of valid public keys for that group member. The new process will then unreliably broadcast a gossip message, which will be processed by the group members in the same way as described above.

2.2.2 View Installation to Remove a Single Corrupt Member

This section describes how the group membership protocol removes a single corrupt member from the group. Here, we exclude the possibility that additional processes will exhibit faulty behavior during the view installation procedure. However, we do tolerate that scenario. A description of how we do so is in Section 2.2.3.

We describe how a corrupt member is removed from the group by representing the group membership protocol as a set of communicating finite state machines, each as shown in Figure 2.1. Upon initialization, all state machines are in the NORMAL state. Any member of the group may invoke the suspect function; correct processes will do it if they detect that another member has deviated from its specified behavior, while corrupt processes may do it at any time. When the suspect function is invoked, the group membership protocol of the process that suspects another process will broadcast a signed *Suspect* message to the group

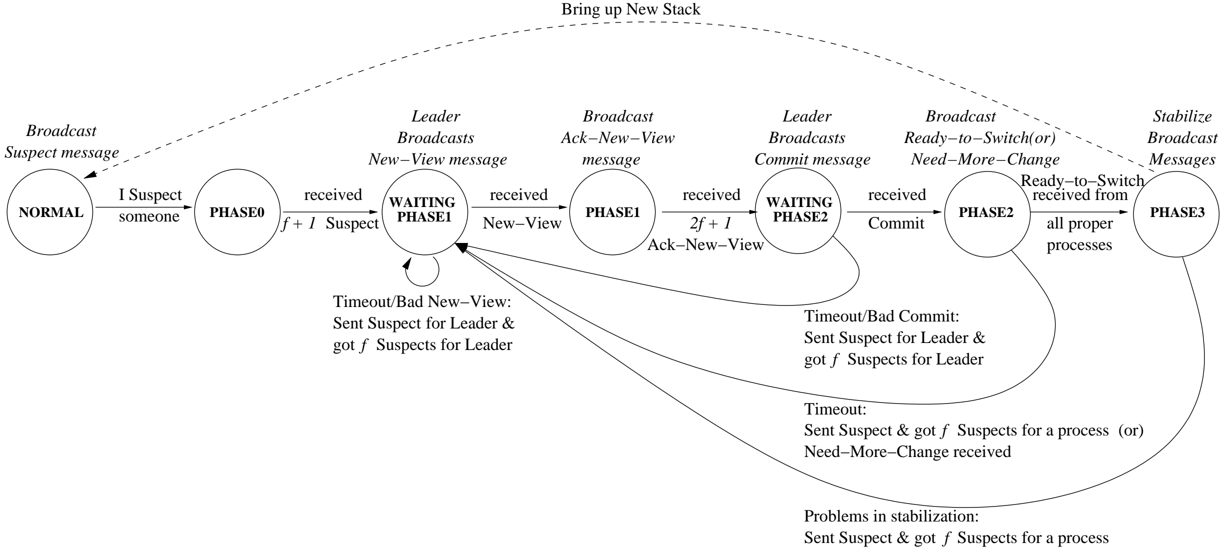


Figure 2.1: Finite state automaton for view installation

and change its state to PHASE0.

When a non-leader process in the group has seen $f + 1$ *Suspect* messages for a member, it changes its state to WAITING-PHASE1. The process also starts a timer, and expects the leader of the group to take action before the timer expires. If the $f+1$ *Suspect* messages were for the leader, the deputy is expected to become leader and take action. When the leader sees $f + 1$ *Suspects*, it broadcasts a signed *New-View* message. A valid *New-View* message contains the list of endpoints for the next view, which excludes the corrupt member. It also contains justification for exclusion of the corrupt member from the next view, namely the $f + 1$ *Suspect* messages.

When a valid *New-View* message is received, a correct process changes its state to PHASE1 and broadcasts a signed *Ack-New-View* message. If a process p_k acknowledges a *New-View* message from p_b, then it will not acknowledge any more *New-View* messages from processes of lower rank than p_b in that view. Upon receiving $2f+1$ *Ack-New-View* messages for a *New-View* message, a correct non-leader process changes its state to WAITING-PHASE2. It also starts a timer, and expects the leader to take action before the timer expires. Upon seeing $2f + 1$ *Ack-New-View* messages for the *New-View* message it sent, the leader (if correct) broadcasts a signed *Commit* message. A valid *Commit* message contains the same view specified in the *New-View* message and includes the $2f+1$ *Ack-New-Views* as proof that the majority of the correct processes have acknowledged its *New-View* message.

When a valid *Commit* message is received, a correct process changes its state to PHASE2.

It then broadcasts a signed *Ready-to-Switch* message. It also starts a timer, and expects a *Ready-to-Switch* message from all members of the new view before the timer expires. When *Ready-to-Switch* messages have been received from all members of the new view, a correct process changes its state to PHASE3. The members of the current view that are also members of the next view then begin to form a consensus on which messages have been delivered by each of them up to this point. This is the *message stabilization phase*; it is needed to ensure that all correct processes deliver the same set of messages broadcast in the current view. In this final phase, each process multicasts a signed message with an array indicating the highest-sequence-numbered multicast from each member that it has reliably delivered. Once a process has received these multicasts from all other correct processes, it enters the second round, in which it multicasts a message containing all signed copies from the previous round. When a process has received the second round of multicasts, for each entry of the array, it expects that the process that claims to have the highest value of that entry will multicast all messages that the process with the lowest value (of that entry) claims not to have received. Each of these rounds has time-outs associated with it, and members that obstruct progress are reported as suspects to the group membership protocol. After all application messages broadcast in the current view have stabilized, each correct process that is a member of the new view installs a new protocol stack, which is initialized to the NORMAL state. Each of the three phases of the view installation has timers to ensure liveness. If a timer expires before the corresponding action expected from a process is observed, then a *Suspect* message is sent for the process.

2.2.3 View Installation when Multiple Faults Occur

An enhanced algorithm is used to remove the earlier assumption that no additional faults occur during a view installation. An additional fault has occurred during a view installation if $f+1$ *Suspect* messages have been received for a member that is not among those processes being removed by the current view installation.

Consider a case in which an additional fault occurred during the view installation when a correct process p_k had not yet reached the state PHASE2. When p_k receives a *Commit* message from the leader, it changes its state to PHASE2, as described in the previous subsection. However, this time, it broadcasts a signed *Need-More-Change* message (instead of a *Ready-to-Switch* message, as described before). The *Need-More-Change* message indicates that the proposed new view specified in the last *New-View* message does not exclude all known corrupt members. A valid *Need-More-Change* message points out the other corrupt members that need to be excluded, and provides justification in the form of the $f+1$

Suspect messages received for each of these other corrupt members. After sending this *Need-More-Change* message, p_k starts a timer, and expects a *Ready-to-Switch* message or a *Need-More-Change* message from each of the members of the view proposed by the last *New-View* message, except the corrupt ones. After receiving those *Ready-to-Switch* or *Need-More-Change* messages, p_k changes its state to WAITING-PHASE1 (as shown in Figure 2.1). p_k then starts a timer and waits for another *New-View* message from the leader, excluding at least one more known corrupt member from the next view than it did in the last *New-View* message. The last *New-View* message received did not result in the installation of a new protocol stack. That *New-View* message and the corresponding *Commit* message (if it was broadcast) are part of what we call a *transitional view*. If the additional fault was at the leader, then a *Commit* message from the corrupt leader may never be received. If it is not received, then the deputy takes over as the new leader, changes its state to WAITING-PHASE1, and broadcasts a *New-View* message. This is shown in Figure 2.1 by the reverse transition from the state WAITING-PHASE2 to the state WAITING-PHASE1. Other correct processes change their states to WAITING-PHASE1, start timers, and wait for the *New-View* message from the new leader.

If the additional fault occurs when p_k is in state PHASE2 or PHASE3 (i.e., after it has responded with a *Ready-to-Switch* message), then it reverts back to the state WAITING-PHASE1 (as shown in Figure 2.1). Then, if p_k is a leader, it broadcasts a *New-View* message that excludes at least one more known corrupt member from the next view than the last *New-View* message did; if p_k is a non-leader, it starts a timer and expects to receive a *New-View* message from the leader before the expiration of the timer.

The timer value associated with this WAITING-PHASE1 (reached from either WAITING-PHASE2, PHASE2, PHASE3, or WAITING-PHASE1 itself) at the non-leader processes will be higher than the value associated with WAITING-PHASE1 if there is no transitional view. After p_k moves to WAITING-PHASE1, the view installation follows the procedure outlined in 2.2.2, with a correct process changing state to PHASE1 upon receipt of the new *New-View* message, and so on. Should $f + 1$ *Suspect*s for another process, p_j, be received after this latest *New-View* message is received, then that *New-View* message will become part of another transitional view, and a *Need-More-Change* message will be broadcast as the response to the *Commit* message in this transitional view. That would trigger the broadcast of another *New-View* message, excluding p_j and the processes excluded by the last *New-View* message. This cycle of transitional views, in which a process keeps changing its state back to WAITING-PHASE1, would continue until a *New-View* message finally excludes all known corrupt members and all three phases of the view installation are completed, thus bringing up a new protocol stack.

If there is just one *New-View* message broadcast between two successive concrete protocol stack creations, then there was no transitional view involved. If more than one *New-View* message was broadcast between two successive protocol stack creations, then all except the last of the *New-View* messages correspond to transitional views.

2.2.4 Detailed Protocol Description

As described previously, any protocol in the group communication protocol stack can invoke the *suspect(process_rank i, reason R)* function provided by the group membership protocol, if it has observed that the peer protocol in process p_i has deviated from the protocol specifications. If the group membership protocol (GMP) is in state NORMAL, it will broadcast a *Suspect* message to the group and change its state to PHASE0. This message has two parts:

- A data structure containing the view-id and the rank of the suspected member.
- A signed version of the above data structure, called *suspect_notice*. The suspect_notice sent by process p_k for a process it suspects, p_i, is denoted by $suspect_notice_k(i)$.

The GMP maintains a *notice_matrix* data structure. The cell $notice_matrix_{k,i}$ in this matrix contains $suspect_notice_k(i)$. When the GMP receives a *Suspect* message m broadcast from process p_k, it invokes the *suspect_message_handler(m, k)* function. The pseudo-code for this function is shown in Figure 2.2. The $verify_suspect$ function verifies whether the signature in the *suspect_notice* is valid. It also checks whether the rank of the suspected process, i, is between 0 and $|V_x|-1$. If the *suspect_notice* is the $f+1^{th}$ distinct *suspect_notice* received for p_i, then the predicate $faulty(i)$ holds. This starts the first phase of the view installation.

The $found_new_fault$ function (Figure 2.3) is invoked whenever the *faulty(i)* predicate becomes true for some process p_i. If the GMP state is NORMAL or PHASE0, the $found_new_fault$ function switches the GMP state to WAITING-PHASE1. At the leader, this function creates a *New-View* message containing the following:

- The list of processes, p_i, such that *faulty(i)* holds.
- For each such p_i, the proof that *faulty(i)* is true. This proof is the ith column in the *notice_matrix*, which should contain at least $f + 1$ valid *suspect_notices* for p_i.
- The proposed new view to be installed next, which excludes all such p_i.

At a process other than the leader, the $found_new_fault$ function starts a timer, T_1. The process expects a *New-View* message from p_{my_leader} before T_1 expires[3]. If T_1 expires before

[3] p_{my_leader} at any process indicates who that process thinks is the leader.

suspect_message_handler (Suspect_message m, proc_rank k) in view V_x

```
1: i ← get_suspect_rank(m)
2: y ← get_view(m)
3: notice ← get_suspect_notice(m)
4: if (x ≠ y) or (suspect_rank = my_rank) then
5:    Stop
6: end if
7: if verify_suspect(m) fails then
8:    Invoke the function suspect(k, Bad-Suspect)
9:    Stop
10: end if
11: if notice_matrix_{k,i} is empty then
12:    Store notice in notice_matrix_{k,i}
13:    if this is the f+1^th distinct suspect_notice received for process p_i then
14:       faulty(i) ← TRUE
15:       my_leader ← smallest j (j ≥ 0) such that faulty(j) is still false
16:       found_new_fault(i)
17:    end if
18: end if
```

Figure 2.2: The *suspect_message_handler* function

the *New-View* message was received, then the process invokes *suspect(my_leader, New-View Time-out)*. If it receives the *New-View* message m from p_{my_leader} before the timer expiry, it invokes the *received_newview(m)* function. The pseudo-code for this function is shown in Figure 2.4.

The *received_newview* function checks the validity of the *New-View* message received. It updates the list of processes, p_i, for which the *faulty(i)* predicate holds. It also updates the entries in the *notice_matrix* for those processes. The GMP maintains a data structure called *View-List*. This is a list for storing the proposed next view indicated by the *New-View* messages received. For each *New-View* message, the *View-List* also contains an array of *Ack-New-View*s received (called *Ack-array*), an array of *Ready-to-Switch* messages received (called *RTS-array*), and an array of *Need-More-Change* messages received (called *NMC-array*). Each of these three arrays contains $|V_x|$ cells, which get updated when the corresponding message is received. After the *New-View* message has been verified to be valid, the GMP switches its state to PHASE1 and adds the proposed next view to the *View-List* data structure. It then broadcasts an *Ack-New-View* message, which contains the signed

found_new_fault (Suspect_message m, process_rank k) in view V_x

```
1: old-state ← GMP-state
2: if old-state = NORMAL or old-state = PHASE0 or old-state = PHASE2
   or old-state = PHASE3 then
3:    GMP-state ← WAITING-PHASE1
4:    if my_leader = my_rank then
5:       Broadcast a New-View message
6:    else
7:       Start/Restart timer T1
8:    end if
9: else if old-state = WAITING-PHASE1 then
10:   if this new fault has caused me to become the new leader then
11:      Broadcast a New-View message
12:   else
13:      Increment the time-out value for T1
14:   end if
15: else if old-state = PHASE1 or old-state = WAITING-PHASE2 then
16:   if this new fault changed my_leader then
17:      GMP-state ← WAITING-PHASE1
18:      if I am the new leader then
19:         Broadcast a New-View message
20:      else
21:         Restart the timer T1
22:      end if
23:   else if old-state = WAITING-PHASE2 then
24:      Increase the timeout value for T2
25:   end if
26: end if
```

Figure 2.3: The *found_new_fault* function

received_newview (NewView_message m) in view V_x

1: *being_removed* ← set of processes in V_x to be excluded from the V_{x+1}, as indicated by m
2: **for** each p_i in the set *being_removed* **do**
3:   **if** $i = my_rank$ **then**
4:     Stop
5:   **end if**
6:   **if** m does not have the array of *suspect_notices* for p_i containing at least $f + 1$ valid *suspect_notices* **then**
7:     Invoke the function *suspect(my_leader, Bad New-View)*
8:     Stop
9:   **else**
10:     **if** $faulty(i)$ is not already TRUE **then**
11:       $faulty(i)$ ← TRUE
12:       Update the ith column in the *notice_matrix* with the *suspect_notices* in m for p_i
13:     **end if**
14:   **end if**
15: **end for**
16: *my_leader* ← smallest j ($j \geq 0$) such that $faulty(j)$ is still false
17: **if** $origin(m) = my_leader$ **then**
18:   GMP-state ← PHASE1
19:   Add the proposed next view indicated by m to the *View-List* data structure
20:   Broadcast an *Ack-New-View* message containing the signed version of the proposed next view
21: **end if**

Figure 2.4: The *received_newview* function

version of the proposed next view.

When the GMP receives an *Ack-New-View* message, it verifies the validity of the message and then stores it in the *Ack-array* of the corresponding new view in the *View-List* data structure. When $2f+1$ *Ack-New-View* messages for a proposed next view have been received, the second phase of the view installation begins.

At the leader, the GMP forms a *Commit* message containing the proposed next view and justification for installing the next view. The justification is the *Ack-array*, which contains at least $2f + 1$ *Ack-New-View*s. It shows that a majority of the correct members agree that it is fine to exclude the faulty members indicated by the *New-View* message. The GMP now

switches from state PHASE1, which it must be in at this point, to WAITING-PHASE2.

At processes other than the leader, if the GMP is in PHASE1 when it has received *Ack-New-Views* from $2f+1$ processes, it switches to WAITING-PHASE2 and then starts a timer, T_2. It expects a *Commit* message from p_{my_leader} before T_2 expires. If T_2 expires before the *Commit* message is received, then the GMP invokes *suspect(my_leader, Commit Time-out)*. If the GMP receives a *Commit* message m from p_{my_leader} before timer expiry, it invokes the *received_commit(m)* function. The pseudo-code for this function is shown in Figure 2.5. After checking the validity of the received Commit message, the GMP switches state to PHASE2 and broadcasts a *Ready-to-Switch* message containing the proposed next view, if the proposed next view excludes all p_i for which the $faulty(i)$ predicate holds. The GMP also starts a timer, T_3. All members of the proposed next view should send either a valid *Ready-to-Switch* or *Need-More-Change* message before the timer T_3 expires. The function *suspect(j, Phase2 Time-out)* would be invoked for all p_j that did not send a *Ready-to-Switch* or *Need-More-Change* message before the timer T_3 expired.

If the proposed next view contained in the Commit message does not exclude all known faulty members, then the GMP invokes the $cast_NMC()$ function. The pseudo-code for this function is shown in Figure 2.6.

If the GMP receives a *Ready-to-Switch* message from each member of the proposed next view, it is a confirmation that all of those members have agreed to be part of the next view. The GMP changes its state to PHASE3. The GMP now enters the third and final phase of view installation, in which all broadcast messages are stabilized. This phase involves the collaboration of the GMP with the reliable delivery protocol. The GMP gives the reliable layer a list of processes that are being removed. The members of the current view that are also members of the next view begin a consensus round to decide which messages have been delivered by each of them at this point. To do so, each such member reliably multicasts a signed message with an array that indicates the highest-sequence-numbered message from each member that has been reliably delivered at this member. After that, there is a second round in which every member multicasts a message containing all the signed copies from the previous round. At the end of the second round, each member is able to form a data structure containing three arrays, *max_num_cast*, *has_max_num_cast*, and *stability_vector*.

1. For $0 \leq j < |V_x|$, $max_num_cast[j]$ is the highest-sequence-numbered multicast from p_j received by some group member.

2. For $0 \leq j < |V_x|$, $has_max_num_cast[j]$ indicates which group member has received that highest-sequence-numbered multicast from p_j.

received_commit (Commit_message m) in view V_x

1: **if** $origin(m) \neq my_leader$ **then**
2: Stop
3: **end if**
4: **if** the proposed next view doesn't include me **then**
5: Stop
6: **end if**
7: **if** the justification field of m doesn't contain $2f + 1$ *Ack-New-Views* **then**
8: Invoke the function *suspect(my_leader, Bad Commit)*
9: Stop
10: **end if**
11: GMP-state ← PHASE2
12: **if** the proposed next view excludes all p_i such that $faulty(i)$ is TRUE **then**
13: Broadcast a *Ready-to-Switch* message containing the proposed next view
14: Start a timer T_3
15: **else**
16: $cast_NMC(m)$
17: **end if**

Figure 2.5: The *received_Commit* function

$cast_NMC$ $(Commit_message\ m)$ *in view* V_x

1: $need_to_remove \leftarrow \{\}$
2: **for** each p_i, such that $faulty(i)$ is TRUE but the next view proposed by m does not exclude p_i **do**
3: add i to the list *need_to_remove*
4: Include the ith column of the *notice_matrix* containing $f + 1$ *suspect_notices* for p_i (This serves as the proof that *faulty(i)* holds and hence p_i also needs to be excluded from the next view)
5: **end for**
6: Broadcast a *Need-More-Change* message containing the *need_to_remove* list with the proofs attached

Figure 2.6: The *cast_NMC* function

3. For $0 \leq j < |V_x|$, $stability_vector[j]$ indicates the highest-sequence-numbered multicast from p_j that has been received by all members of the next view.

It is the responsibility of the process $p_{has_max_num_cast[j]}$ to re-broadcast all messages from sequence numbers $stability_vector[j]$ to $max_num_cast[j]$ received from process p_j. Each of the above rounds has time-outs associated with it, and members that obstruct progress are reported as suspects to the GMP, which will then broadcast *Suspect* messages for those members. If the faulty predicate becomes true for a member in this phase, the GMP reverts back to the first phase, where, if it is a leader, it broadcasts a *New-View* message excluding the previous faulty members and the newly discovered faulty member; if it is a non-leader, it restarts timer T_1 and expects to receive a *New-View* message from p_{my_leader} before timer expiry.

2.3 Group Membership Protocol Properties

The group membership protocol described in the previous section has the properties described below. These properties are similar to those provided by SecureRing[KMMS98] and Rampart[Rei94b]. A group membership protocol message m broadcast during the installation of view V_x is denoted by $m(V_x)$. A *Suspect* message sent out for a process r is denoted by *Suspect(r)*. Here, we are concerned only with those views V_i such that *New-View*(V_i) resulted in the installation of a new protocol stack, i.e., *New-View*(V_i) was not part of a transitional view.

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

Suppose that a corrupt leader wants to install two different views, V_x and V_x', at correct processes p_i and p_j. We show by contradiction that this is not possible. For the leader to install a view V_x, it must have presented a *Commit* message with $2f + 1$ *Ack-New-View*(V_x) as proof to p_i. Similarly, it must have presented another *Commit* message with $2f + 1$ *Ack-New-View*($V_{x'}$) as proof to p_j. Since there are two sets of $2f + 1$ processes and the total number of members is $3f + 1$, these two sets must intersect in at least $f + 1$ members. Therefore, these $f + 1$ processes must have sent both *Ack-New-View*($V_{x'}$) and *Ack-New-View*(V_x). However, that is not possible, since we have assumed that only f members are corrupt, and a correct process sends only one *Ack-New-View* message in a particular view to the leader. Hence, by contradiction, $V_x = V_x'$.

Self-inclusion *If a correct process p installs a view V_x, then V_x includes p.*

A process p will discard a *Commit* message from the leader, if the message excludes p from the proposed next view. That means that p will install a view V_x only if p is a member of V_x. Hence the self-inclusion property is satisfied.

Validity *If a correct process p installs a view V_x, then all correct members of V_x will eventually install V_x.*

The group membership protocol is such that installation of V_x is complete at a correct process p only after p has received *Ready-to-Switch(V_x)* message broadcasts from all processes in V_x, and all messages broadcast in V_{x-1} are known to have been reliably delivered at all processes in V_x. Since any broadcast message will be received reliably by all the correct members in the group, they will also eventually receive those *Ready-to-Switch(V_x)* broadcast messages and hence install V_x. This will lead to completion of installation of view V_x at all correct members of V_x.

Integrity *If view V_x includes p but V_{x+1} excludes p, then p was suspected by at least one correct member of V_x.*

Suppose that p is in $V_x - V_{x+1}$. Then, at least $f + 1$ members of V_x broadcast the *Suspect(p)* messages indicating that p should be removed. Since there are at most f corrupt members of V_x, and since correct processes broadcast *Suspect* messages in accordance with the protocol specifications, it follows that some correct member of V_x suspected that p was corrupt.

Liveness *If there is a correct process p that is a member of view V_x and is not suspected by $\lceil(2|V_x|+1)/3\rceil$ correct members of V_x, and there is a process q that is suspected by $\lfloor(|V_x|-1)/3\rfloor$ correct members of V_x, then q is eventually removed.*

The liveness property provided by the protocol is stronger than the one provided in [Rei94b], which says that under the above conditions a new view will eventually be installed, but does not imply that q will eventually be removed. We now present an informal proof that the protocol satisfies this property.

The group membership protocol satisfies the *eventual suspicion* property. The various timers in the three-phase view installation protocol ensure it. By the eventual suspicion property, if for some correct process p in V_x and some corrupt process r in V_x, if $f + 1$ *Suspect(q)* messages have been received for all $q \in V_x$ lower-ranked than r, and V_{x+1} has not been installed for long enough, then *Suspect(r)* is eventually broadcast by p.

Suppose that there is a correct process p in V_x such that $2f + 1$ correct members of V_x do

not suspect p. Then it is not possible for a member that is higher-ranked than p to become a leader by showing $f + 1$ *Suspect(p)* messages. Thus, if p is the leader and broadcasts a *New-View* message, or if p acts to replace the current leader (with rank less than p), then each correct member of V_x will reply to p, and p will complete the protocol and install a new view.

Suppose p is the leader. A correct process that suspects a process p_b will broadcast *Suspect(p_b)* messages until it or p_b is removed. Since p is a correct process, and all correct processes eventually receive broadcast messages, if $f + 1$ correct processes ever suspect a process p_b, then p will eventually receive the *Suspect(p_b)* broadcasts. Hence p will soon broadcast a view message, *New-View(V_y)*, with p_b excluded from V_y. And, by the argument above, V_y will be successfully installed.

Now suppose there is a process p_b for which some subset P of V_x, ($|P| = f+1$), have sent *Suspect* messages. The timers in the view installation protocol ensure that if V_y ($y > x$, and V_y does not contain p_b) is not installed for long enough, then eventually each q in V_x, such that $rank(q) < rank(p)$, will be suspected by at least $f + 1$ members of V_x. These *Suspect* messages will eventually be received at all correct members of V_x. Thus, if p is not the leader, then p eventually receives $f + 1$ *Suspect* messages for each process lower-ranked than itself. According to the argument in the previous paragraphs, that will cause p to become the leader and install a view V_z excluding all such q and p_b.

In this chapter, we described how our intrusion-tolerant group membership protocol acts to form a process group, add new members to the group, and remove corrupt members from the group. We also detailed the properties it provides and proved informally that the protocol indeed provides the stated properties.

Chapter 3

Implementation Details

In this chapter, we describe how the abstract group membership protocol given in Section 2.2 was incorporated into the C-Ensemble framework to provide intrusion-tolerant group membership. [Pan01] describes how we also added intrusion-tolerant reliable delivery and total ordering protocols to the C-Ensemble framework. They are important steps in our effort to build an intrusion-tolerant group communication system (*ITUA GCS*) from the crash-tolerant C-Ensemble.

Since C-Ensemble tolerated only crash failures and no malicious faults, processes within a group trusted each other. For example, if any group member suspected that another member had crashed, and multicast a suspicion to the rest of the group, the group would reconfigure to remove the suspected member. Such *implicit trust* among group members cannot be applied to the toleration of malicious faults. An attacker could compromise some group member, and make that member send spurious suspicions for a correct group member. If group members trusted each other, then by compromising a single group member, the attacker could subvert the whole group.

It was quite obvious that such implicit trust among group members had to be removed, if we wanted to be able to tolerate arbitrary faults. In order to remove the implicit trust from the C-Ensemble microprotocols that relate to group membership and message delivery, we made significant modifications to them. In this chapter, we describe modifications related to the group membership protocol. Changes that were made in order to incorporate intrusion-tolerant message delivery protocols into C-Ensemble are described in [Pan01].

The organization of this chapter is as follows. Section 3.1 will familiarize the reader with the layering architecture of Ensemble in order to provide a context for the rest of the chapter. Section 3.2 describes the layers in the prototype GCS that provide intrusion-tolerant group membership and message delivery. Section 3.3 gives details about the support infrastructure for the C-Ensemble protocol stack, and the changes we made to remove the implicit trust

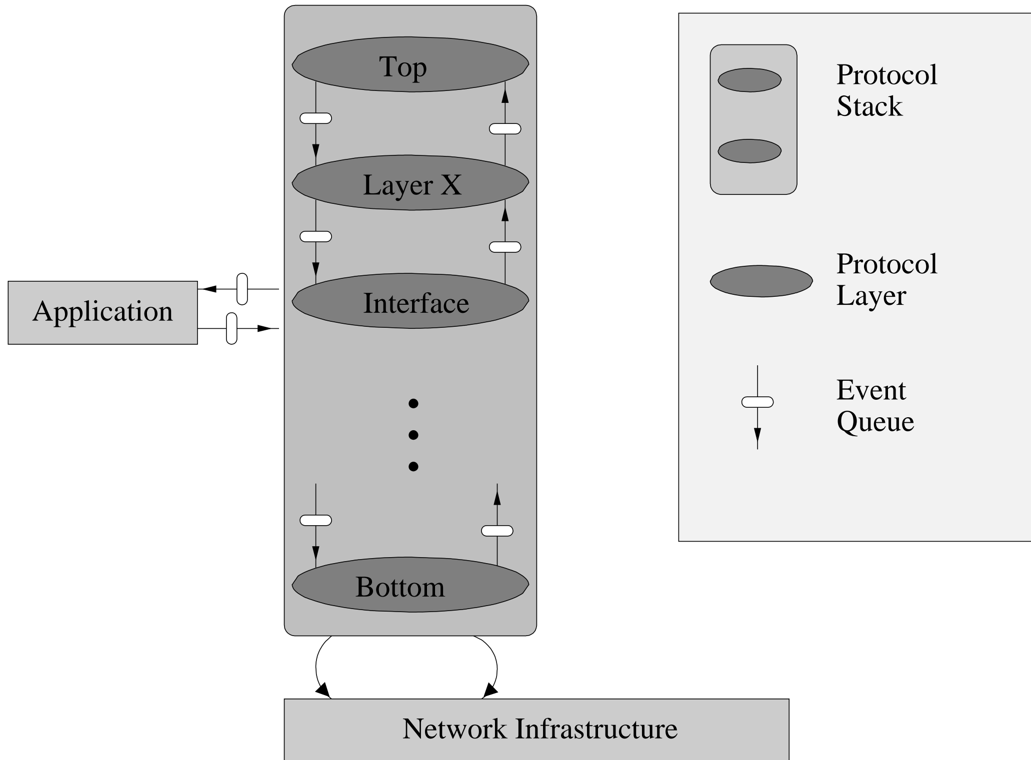


Figure 3.1: Main components of the Ensemble layering model

from the parts of the infrastructure relevant to group membership. Section 3.4 describes how we added cryptographic support to the infrastructure. Section 3.5 tells how we implemented the intrusion-tolerant group membership protocol in the C-Ensemble framework.

3.1 The Ensemble Framework

The Ensemble group communication system [Hay98] is basically a protocol stack composed of layers that work together to implement high-level protocols. It is shown in Figure 3.1. There will be one such protocol stack for each group member. The Ensemble model of a process group is a collection of protocol stacks that have the same linear composition of layers and communicate with each other through messages. Different combinations of layers will yield different protocol stacks that provide different sets of properties. Based on information about the guarantees required by the application, a suitable combination of layers can be chosen to provide those guarantees.

In Ensemble, a process is a unit of state and computation provided by the operating

system. A process may consist of one or more *endpoints*, each of which is associated with a unique identifier. Endpoints are finer-grained divisions of processes, and correspond to the individual members in a group. For simplicity, in this thesis, we view a process as comprising only one endpoint[1]; hence, we use the terms *process* and *endpoint* interchangeably.

Communication between group members (inter-process communication) is done through messages. The network serves as the medium for transmitting messages between processes and supports two primitive operations: *send* and *receive*. The send operation takes a destination address and a message as inputs and transmits the message to the destination group member. The destination group member will invoke the receive operation to get that message, thereby completing the message delivery. A message originates from some layer in the protocol stack. It then passes down the layers below, each of which may add additional information to the message in the form of headers. When the message emerges out of the bottommost layer in the stack, the network infrastructure concatenates a connection identifier to the message and transmits the message over the network. The connection identifier specifies the destination of the message, and helps the infrastructure at the destination pick up the message. The infrastructure at the destination strips the connection identifier and delivers the actual message to the protocol stack. The destination layer in the protocol stack is the peer layer of the layer in the source group member that originated the message. As the message passes up the protocol stack at the destination group member, the layers strip the headers added by their peer layers at the source, and may take actions based on the control information in the header. Thus, access to a header is localized to the layer that generated it.

Communication among the various layers of the protocol stack for a group member (intra-process communication) is done through events. Events are never transmitted through the network, but some events, like *cast* or *send* events, may result in the transmission of messages over the network. Events travel up and down the protocol stack through *event queues*. Each pair of layers is connected by these bi-directional queues. Events are placed on one end of a queue by one layer and removed from the other end by the adjacent layer in FIFO order. Events are the only way in which layers interact with their environments. A layer communicates, using events, with the layers above and below it in the protocol stack. For that purpose, each layer exports handlers for receiving events and messages from its adjacent layers, and, in turn, is given handlers for passing events and messages to its adjacent layers.

The application communicates with the protocol stack using application events, which include group join/leave and message send/receive events. Although most layering architec-

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

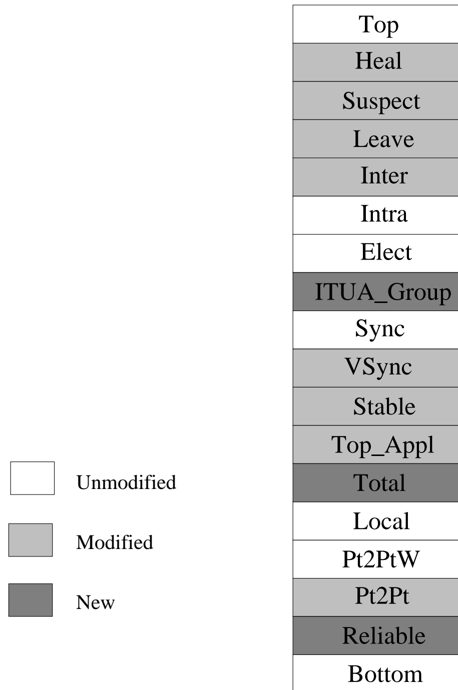


Figure 3.2: Modified C-Ensemble stack

tures place the application at the top of the stack, in Ensemble the application is considered part of one of the layers. That allows the application to appear lower in the protocol stack, eliminating the overhead of the layers above it when it sends messages.

There are several advantages to the layering architecture of Ensemble described above. Layers can be combined in a number of ways, providing rich sets of combinations of properties from which applications can select. Layering also makes it easy to change the system incrementally. Layers cannot access other layers' states or headers; that eliminates dependencies between layers on these data types. Thus, one layer's header or state can be changed without affecting other layers. It is also easy to replace a layer or to add a new layer that provides new properties. Because of the limited interactions between the layers, it is easy to model their behavior to validate their correctness, and evaluate the performance of individual protocols.

3.2 ITUA GCS Protocol Stack

Figure 3.2 shows the protocol stack for the prototype GCS that provides intrusion-tolerant group membership and message delivery. Three of the layers, *ITUA_Group*, *Total*, and

Reliable, were added by us to the original C-Ensemble. They form the core of the intrusion-tolerant group membership, total ordering, and reliable delivery protocols, respectively. The *Heal*, *Suspect*, *Leave*, *Inter*, *Vsync*, *Stable*, *Top_Appl*, and *Pt2pt* layers were modified (from the original C-Ensemble implementation) to remove the implicit trust among group members and to provide intrusion-tolerant group membership and message delivery. Section 3.5 describes the changes made to those layers. Other layers, like *Top*, *Intra*, *Elect*, *Local*, *Pt2PtW*, and *Bottom*, were used unmodified.

The *Top* layer and the *Bottom* layer, as the names indicate, are the topmost and bottommost layers in the stack. The *Bottom* layer communicates with the network infrastructure. The *Heal* layer originates the *Gossip* message that advertises the presence of a process, and is used by the process to help it join its intended group (see Section 2.2.1). The *Suspect* layer generates heartbeat messages to let the group know that the process is alive, and reports processes suspected of crashing to *ITUA_Group*. The *Inter* layer implements the functionality for forming groups and adding new processes to a group. The *ITUA_Group* layer implements the first two phases in the three-phase protocol to remove a group member. The third phase (message stabilization phase) is implemented by the *Vsync* and *Stable* layers. Thus, the *ITUA_Group* layer, along with the *Vsync*, *Stable*, *Heal*, *Suspect*, *Leave*, *Inter*, and *Intra* layers, provides the intrusion-tolerant group membership properties. The *Top_Appl* layer is the application's interface to the protocol stack.

The intrusion-tolerant group membership protocol installs a new view at a group member by bringing up a new protocol stack (shown in Fig 3.2) at that member, whenever some key attribute of the group (like the configuration of the group) changes. The new protocol stack will be configured through use of a record called the *ViewState* record. It contains all the information that a member needs to communicate within the group, such as the name of the group, the membership list, and the addresses of the processes. The group membership protocol guarantees that all the correct processes in a group have equivalent *ViewState* records, provided that no more than one-third of the members are corrupt. Each member also has a *local state* record, which stores information specific to that group member, such as its address and rank. Each layer in the protocol stack has access to the *ViewState* record and the *local state* record. Each layer in the stack also maintains its own state information, which is local to that layer and cannot be accessed or modified by other layers in the stack.

3.3 Infrastructure Implementation

Like C-Ensemble, our prototype GCS has a library that implements the support infrastructure for the protocol stack. The support infrastructure includes the network infrastructure, which transmits the message at the sender and delivers it to the receiver. The library provides data structures and functionality commonly needed by the layers, such as scheduling of the event queues between the layers, creation of events, and so forth. It is the infrastructure that composes the layers to form a new protocol stack during a new view installation. At the end of the three-phase protocol for view installation, the *Top* layer at each group member emits a *View* event containing the *ViewState* record for the new group. When this event emerges out of the bottom of the stack, the infrastructure uses the information in the *ViewState* record to select the appropriate layers for the new protocol stack, creates a new local state for each layer, connects the stack to the network, and initializes the layers with the *ViewState* and their respective local state records.

Layers have no direct access to the system clock or timers, and request alarms by generating *Timer* events and sending them down the protocol stack. When a timer event emerges out of the *Bottom* layer, the infrastructure gets the timeout value. After the specified time has expired, the infrastructure sends up a *Timer* event with the current time.

The infrastructure provides *marshalling* functions to linearize the data structures used to form the message into a sequence of bytes that can be transmitted over the network. The functions are used by the layers at the sender side. At the receiver end, the corresponding peer layers use the *unmarshalling* functions to convert the sequences of bytes back into the data structures that can be interpreted by the layers.

We reused the infrastructure library of C-Ensemble, making some modifications so that we could use the signed versions of the raw C-Ensemble data structures. We also had to create new events, and data structures to support the new events, for our intrusion-tolerant group membership protocol.

3.4 Cryptography Details

In order to remove the implicit trust among group members, we had to make modifications so that the messages received over the network could be authenticated using digital signatures. That meant adding cryptographic support to the C-Ensemble infrastructure library. For that purpose, we used Peter Gutmann's Cryptlib [Gut01] as the core cryptographic library, and wrote wrapper functions around it. Table 3.1 describes some of the wrapper functions.

All GMP messages are multicast using the reliable multicast protocol, which uses the

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

Table 3.1: The cryptographic support functions added to the infrastructure

crypt_hash_buf and *crypt_hash_buf_verify* wrapper functions to create and verify message digests. The *crypt_sign_buf* wrapper function is used to sign the message digests; the *crypt_sign_buf_verify* wrapper function is used to verify these signatures. To sign messages that are not reliably multicast, the *marsh_crypt_sign* wrapper function is used. At the receiving end, these messages are verified using the *marsh_crypt_verify* wrapper function.

The *ITUA_GMP* layer makes use of the *crypt_sign_buf* and *crypt_sign_buf_verify* wrapper functions. They are used to sign and verify *suspect_notice*s and the proposed new view in *Ack-New-View* messages, as described in Section 2.2.4. The signed *suspect_notice*s are used as justification in a *New-View* message. The signed versions of the proposed new view from the *Ack-New-View* messages are used as justification in a *Commit* message. (The details are described in Section 2.2.4.)

In Section 2.2.1, we saw that the group membership protocol needs access to the public key information of other group members during group formation and while a member is being added to the group. For that purpose, the *ViewState* and local state records had to be modified (from the C-Ensemble implementation) to include cryptographic information.

The local state now has these additional fields, which carry cryptographic information:

- *private_keyset:* reference to a file that stores the private key of the group member.
- *passwd*: password for accessing the key stored in *private_keyset*.
- *public_keyset:* reference to a file that stores public keys of all possible members of the group. Each key is associated with a *key_id*, which is used to access a particular key.

- *key_id_list:* list of *key_ids* needed to access the *public_keyset*.
- *my_key_id: key_id* of the group member. Such keys are exchanged by members during group formation and when a member joins a group.

The above fields are initialized with the appropriate values at the time of process creation. In the ITUA architecture, the manager/subordinate that creates the process will provide the values.

The *ViewState* record now contains a field *key_ids*, which stores the key IDs of the processes present in the current view in order of rank. The field is updated whenever the membership of the group changes.

3.5 Group Membership Protocol Implementation

The *ITUA_GMP* layer implements the data structures and algorithms for the core of the three-phase view installation protocol (detailed in Section 2.2.4) for removing corrupt members from the group. We had to make several changes to other layers of C-Ensemble in order to support our protocols for group formation and removal of corrupt members. Those changes were as follows.

C-Ensemble detects process crashes by noticing missing heartbeat messages. When a process crash is detected, C-Ensemble sends a *dn(suspect)* event that directly triggers the events for installing a new view. We don't want that to happen in our GCS. Instead, before installing the new protocol stack, our GCS should perform a three-phase protocol during which the members agree that the suspected process should be removed from the group. Therefore, we have added a new event, *itua_suspect*, to the infrastructure. The *Suspect* layer that detects the missing heartbeat messages has been modified to send a *dn(itua_suspect)* event instead of the original *dn(suspect)* event. The *dn(itua_suspect)* event will then be bounced up to the *ITUA_GMP* layer, which will then broadcast a *suspect_notice* for the suspected process. That triggers the three-phase view installation protocol.

We had to modify the *Heal* layer to include the process's public key and *key_id* in the information being unreliably broadcast (*Gossip* message) when a process wants to join a group. The *Inter* layer was modified to accept join requests only from processes whose public keys are on the list of certified public keys that correspond to the processes that are allowed to be part of the group.

We modified the *Vsync*, *Top_Appl*, and *Stable* layers to ensure that messages multicast in the current view are delivered at all correct processes before the group switches to the

next view. In doing so, we had to account for the possibility that new corruptions may be discovered during the message stabilization phase. During view change, the members exchange information about the multicasts they have received from each other. We added data structures that indicate, for each group member, the process that has the highest-sequence-numbered multicast. We also added timeouts that prevent the stalling of the message stabilization phase. In C-Ensemble, if a group member claimed to have the highest-sequence-numbered multicast from another member, its claim was trusted, and other group members that did not receive that multicast waited until the claiming member re-broadcast the message. If the claiming member was malicious, it could stall the view installation process by not rebroadcasting the message. In our implementation, we removed the trust. If a process holding the highest-sequence-numbered multicast from a group member does not rebroadcast unstabilized messages from that group member, the *Top_Appl* layers at other group members will generate *up(stabilize_suspect)* events for the process trying to stall the protocol. That will cause the *ITUA_Group* layer at those members to multicast *Suspect* messages for that corrupt process. On receiving enough such *Suspect* messages, the *ITUA_Group* layer at the leader will multicast a *NewView* message that excludes the previous faulty members and the newly discovered faulty member.

Chapter 4

Performance Measurement

One of the important reasons for developing the microprotocols that provide group membership and reliable ordered message delivery in the ITUA GCS was to measure the cost of building intrusion-tolerance mechanisms at the group communication system level. To help us study the variation of message delivery times with varying group sizes, stack configurations, and layer parameters, we have written an application with process groups that communicate over the network. Our results are presented and analyzed in [Pan01]. We also injected malicious faults at some members, and measured the time it took for other, non-faulty members to recover from the faults and install new views removing the faulty members. This section describes in detail the different performance costs associated with intrusion-tolerant group membership in a GCS.

4.1 Experimental Setup

The tests were carried out on a testbed of ten 1GHz Pentium III computers with 256MB PC133 RAM. The computers were connected by a full-duplex 100 Mbps switched Ethernet network. The machines were otherwise unloaded, and, unless specified otherwise, a single process ran on each machine. The time measurements were taken in units of clock cycles, using an assembly-level instruction provided by the Pentium instruction set. These measurements were converted into milliseconds for presentation.

4.2 Results for Group Membership

This section presents results that show the cost of excluding a corrupt member or members from the group when faults occur. We injected faults at one or more group members. For the purpose of these experiments, a correct group member is one that has not had a fault injected

into it. A corrupt group member is one at which a fault has been injected. A group member has detected a fault when it has received $f+1$ *Suspect* messages for the corrupt member. The member that has detected the fault then starts the three-phase view installation protocol described in Section 2.2. At each correct member, we take time measurements when a fault has been detected and when a new view excluding all known corrupt members has been installed. The elapsed time is the time taken for the view installation. The average of these times across all correct group members is the presented time for view installation for that particular group size. We study the time taken for view installation when more than one group member was corrupted, when the injected faults are activated at different phases of transitional views, and when fault detection does or does not involve time-out mechanisms. We also estimate the overhead due to cryptography.

The scenario we use to quantify the cost of removing corrupt group members is as follows. Each process is started with the same group name and the same target group size, called $target_group_size$. The group membership protocol ensures that all processes join a single group. When the group size reaches $target_group_size$, each process starts multicasting messages to all of the members in the group. After a fixed number of rounds of message multicast, one or more members are injected with one of the following three types of faults:

1. *Crash*, causing the corrupted process to kill itself,
2. *Mutant message*, in which the corrupted process sends two messages with the same sequence number but with different contents, or
3. *Impede-total-ordering*, in which the total ordering layer in the corrupted process does not send the required *null* messages [Pan01] when application-level multicasts are stopped.

These faults are detected in different ways. For crash faults, if *heartbeat messages* have not been received from a group member for more than a specified length of time, the group member is suspected to have crashed. Mutant messages are identified by the reliable multicast protocol. For the impede-total-ordering faults, the total ordering protocol at correct members finds that neither application-level messages nor *null* messages have been received from a member for more than a specified length of time, and reports the suspected member to the group membership protocol.

Figure 4.1 shows the view installation times for various group sizes when one of the above faults is injected at the leader. The increase in view installation time is fairly linear from group sizes 4 to 6 and from 7 to 9. The increase is steeper as we go from group size 6 to 7 and from 9 to 10. This behavior can be explained as follows. At group sizes 7 and 10, the number

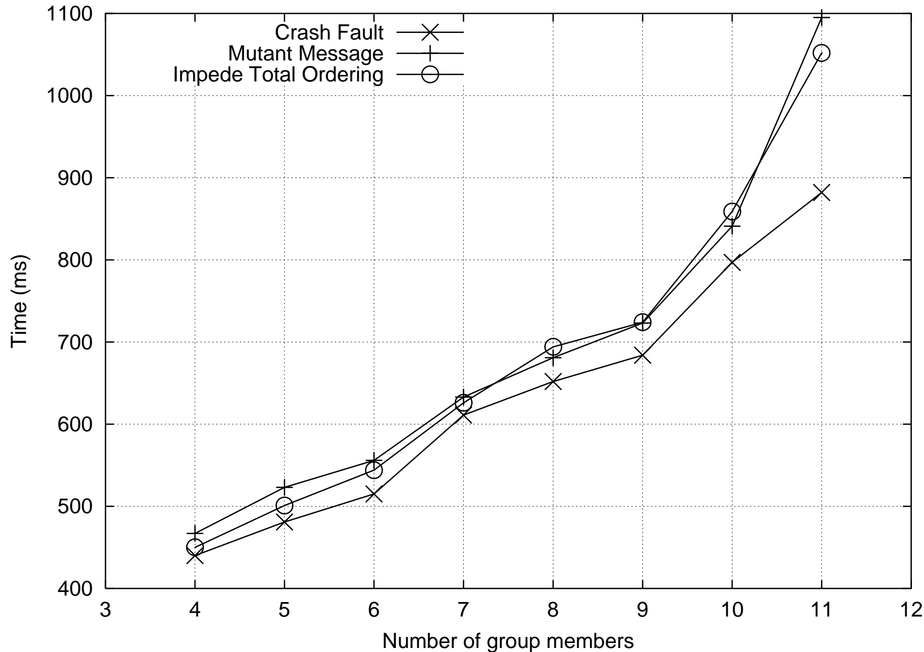


Figure 4.1: View installation times for single faults

of faults that can be tolerated, f, increases by one. Correspondingly, the number of *Suspect* messages $(f+1)$ that need to be shown as proof in a *New-View* message increases by 1; the number of *Ack-New-Views* being collected and the number of *Ack-New-Views* $(2f+1)$ that need to be shown as proof in a *Commit* message increase by 2. Thus, the steeper increase in view installation times at group sizes which are of the form $3f+1$ is due to the increase in the number of messages that need to be collected and processed before progression to the next step of the view installation.

A group whose size is greater than 6 processes can tolerate two simultaneous faults (we call such faults *double faults*). To quantify the cost of tolerating two faults, we inject a crash fault at one of the non-leader processes, and one of the faults in Table 4.1 at the leader process. When a view installation is initiated to exclude the crashed process from the group, a fault at the leader process is activated, so as to impede the view installation. As described in Section 2.4, the group membership protocol has time-outs and other mechanisms to ensure liveness. These mechanisms will detect the fault at the leader process. The additional fault at the leader will cause the first view installation to become a transitional view, and will trigger another round of view installation. Table 4.1 describes a list of faults that can occur during the view installation, and the stage in the view installation when they are activated.

A group whose size is greater than 9 processes can tolerate three simultaneous faults (we call such faults *triple faults*). For groups larger than 9, we inject a crash fault into one of

Table 4.1: Faults injected during a view installation to make it a transitional view

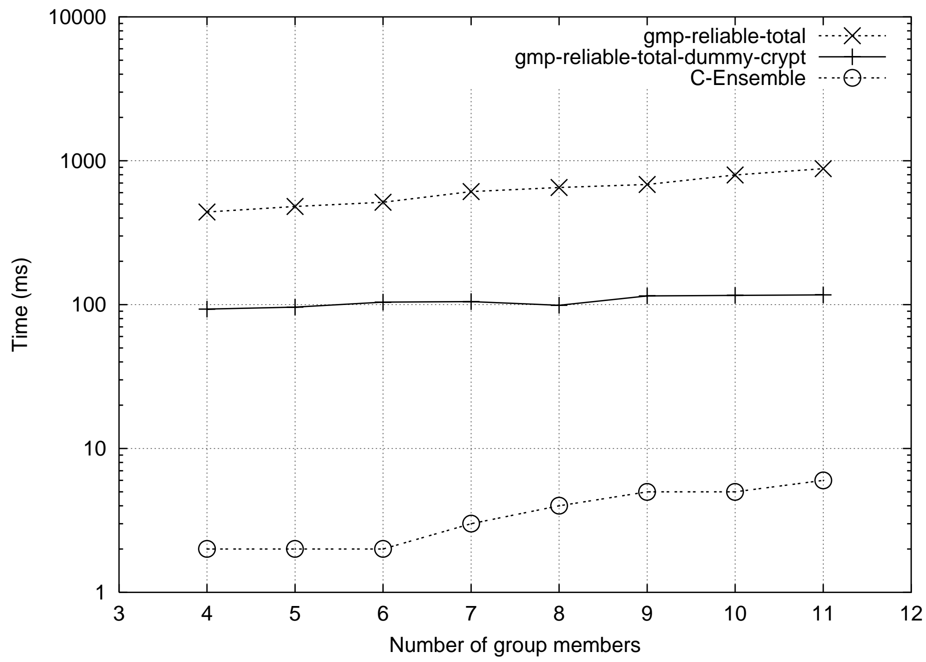
Fault	Fault Activation Point	Description
Bad New-View	start of PHASE1	Leader sends a *New-View* message with insufficient justification (less than $f+1$ signed suspects)
New-View Time-out	start of PHASE1	Leader does not send a *New-View* message after receiving $f+1$ suspects
Bad Commit	start of PHASE2	Leader sends a *Commit* message with insufficient justification (less than $2f+1$ signed *Ack-New-Views*)
Commit Time-out	start of PHASE2	Leader does not send a *Commit* message after receiving $2f+1$ *Ack-New-Views*
Ready-to-Switch Time-out	end of PHASE2	Process does not send a *Ready-to-Switch* after receiving a valid *Commit* excluding all known corrupt members
Impede Stabilization	PHASE3	Process claims to have received a high sequence number that was never multicast. When it does not rebroadcast that message for a sufficiently long time, other members start to suspect it.

the group members, inject one of the faults in Table 4.1 at the leader, and inject one of the faults in Table 4.1 at the deputy. When a view installation to remove the crashed member is initiated, the fault at the leader becomes activated, causing that view installation to become a transitional view. The deputy is supposed to start a new round of view installation. The fault at the deputy becomes activated during this round, resulting in another transitional view. A new protocol stack will be installed at the correct members in the group only when the deputy's deputy becomes the leader and starts the view installation procedure.

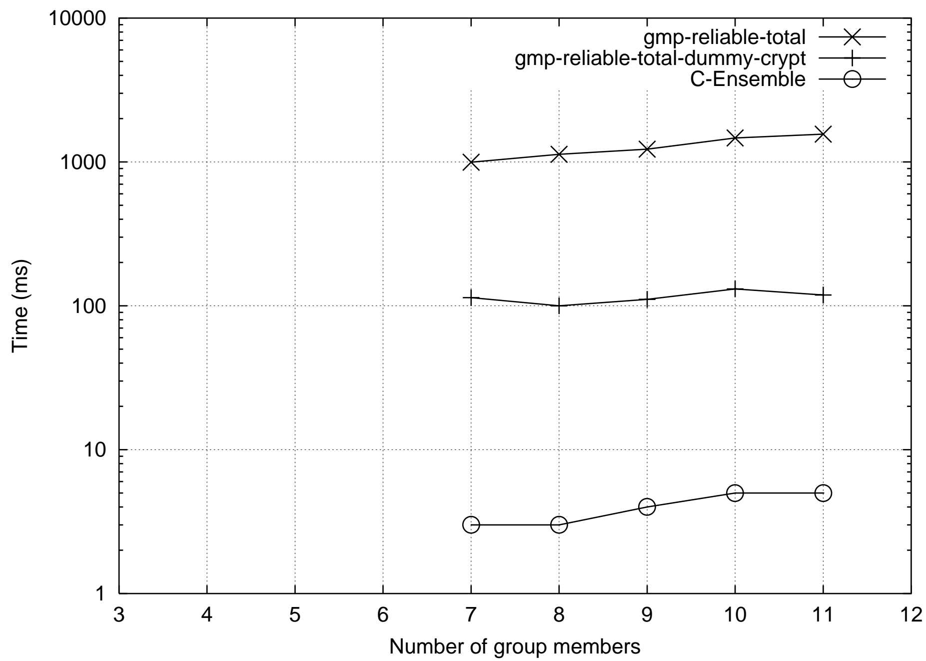
Figure 4.2 shows the view installation times for three protocol stacks:

gmp-reliable-total: This is the Ensemble stack with the new intrusion-tolerant microprotocols for providing group membership, reliable multicast, and total ordering. This stack uses normal cryptography.

gmp-reliable-total-dummy_crypt: This stack includes the same microprotocols as the first stack, but has a dummy version of the cryptographic library. This dummy cryptography library returns from function calls immediately without performing the expensive cryptographic routines needed to sign, verify, encrypt, or decrypt messages. Since the intrusion-tolerant microprotocols rely on public key cryptography for their



(a) View installation times for single crash faults



(b) View installation times for double crash faults

Figure 4.2: Comparing gmp-reliable-total, gmp-reliable-total-dummy_crypt, & C-Ensemble stacks

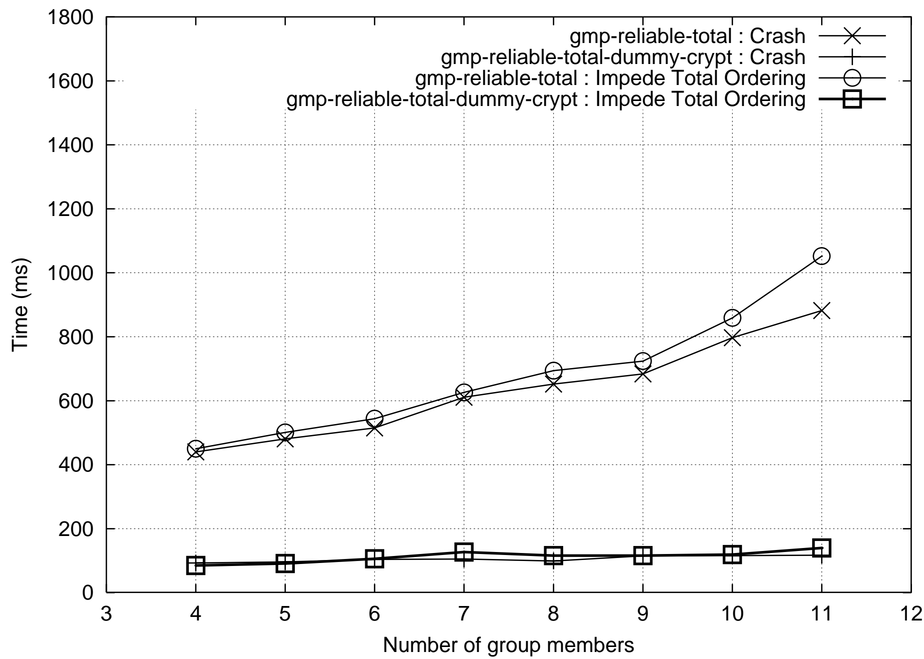
correctness, this stack does *not* provide intrusion tolerance.

C-Ensemble: This is the original C version of Ensemble. It does not include any of the new microprotocols and is tolerant only to crash faults.

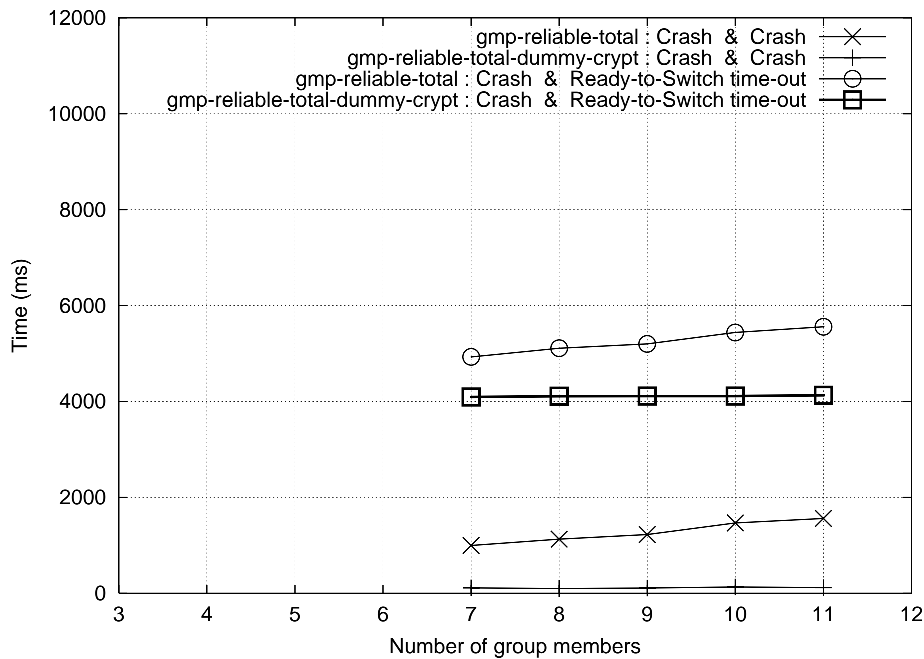
We compare the view installation times for the above three stacks in the presence of a single crash fault or multiple simultaneous crash faults, because the original C-Ensemble can handle only crash faults. Figure 4.2(a) shows the comparison for the single crash fault case. Figure 4.2(b) shows the comparison for the double faults case, in which two non-leader processes in the group are injected with crash faults. From Figures 4.2(a) and 4.2(b), we see that the time difference for view installation between **C-Ensemble** and the **gmp-reliable-total** stacks is two orders of magnitude, and becomes higher with increasing group sizes. This is because of the multiple rounds of message exchange and use of public key cryptography. The increase in the view installation time with increase in group size is very low in the **C-Ensemble** stack; it is on the order of a few milliseconds. In the **gmp-reliable-total-dummy_crypt** stack, the increase is on the order of a few tens of milliseconds. This increase is particularly pronounced in the **gmp-reliable-total** stack, especially at the group sizes at which the number of simultaneous faults that can be tolerated increases by one.

Figure 4.3 shows comparisons between the **gmp-reliable-total** and **gmp-reliable-total-dummy_crypt** stacks, with additional fault types. From Figure 4.3(b), we see that for a given group size, the gap between the values for the two stacks is approximately the same under different combinations of double faults. This is because, for a particular combination of double faults, the number of messages exchanged (communication cost) is the same for both of these stacks. The same observation applies to the single fault case, shown in Figure 4.3(a). The increase in cost with increasing group size is more pronounced for the **gmp-reliable-total** stack than the **gmp-reliable-total-dummy_crypt** stack. (That was also pointed out for Figure 4.2 in the previous paragraph.) That indicates that the cost associated with cryptography increases more steadily than communication cost as group size increases. The steeper increase in the view installation time when moving from group size 6 to 7, and from 9 to 10, was pointed out in Figure 4.1. A similar increase can also be observed in Figure 4.3 for the **gmp-reliable-total** stack, but not as much for the **gmp-reliable-total-dummy_crypt**. This indicates that the steeper increase can be attributed mainly to the signing and verification of additional messages when the fault tolerance of a group increases.

Figure 4.4 compares the view installation times for different combinations of double faults. There are three clusters of curves in this figure. The cluster at the bottom is close to the curves for single faults. This cluster of curves corresponds to combinations of double faults,



(a) Single fault: With and without cryptography



(b) Double faults: With and without cryptography

Figure 4.3: Comparing gmp-reliable-total & gmp-reliable-total-dummy_crypt stacks for different faults

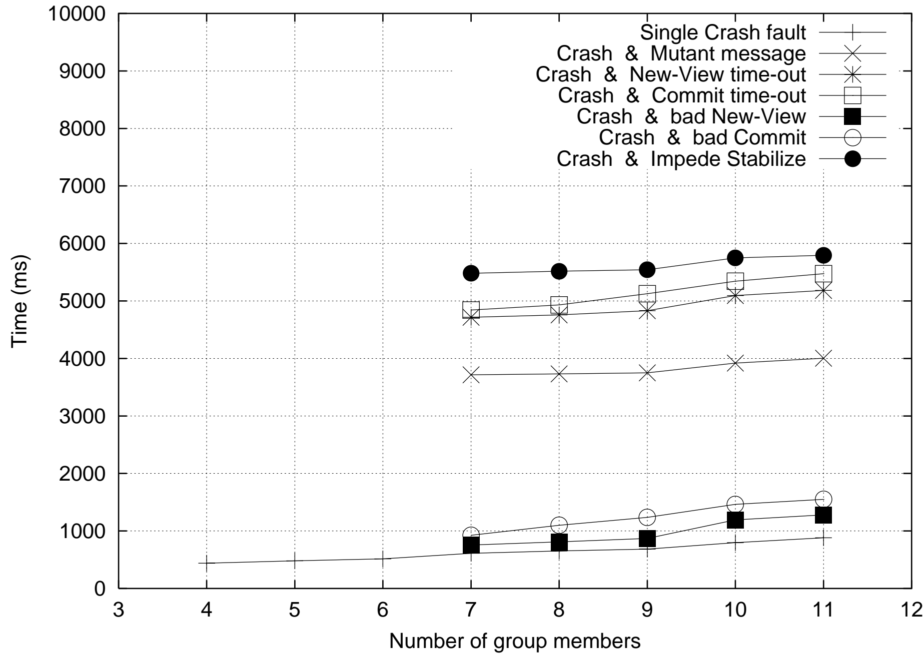


Figure 4.4: Comparing the view installation times for single and double faults

for which fault detection does not rely on any time-outs. The view installation times for these double faults are higher than those for single faults, because their values include the time to detect the additional (second) fault and the time for one transitional view. The difference in the values between one combination of double faults and another in this cluster is not large. Different combinations of double faults are obtained by activating the second faults at different phases of a transitional view. Since the time to complete a single phase in the view installation is small, changing the phase of the transitional view in which the second fault is activated does not have much impact on the view installation time. The other two clusters involve time-outs for fault detection. The time-out value employed is 4 seconds. However, the lone curve in the second cluster falls below 4 seconds, because it depicts the case in which the two simultaneous faults are a crash and a mutant message. The mutant message is detected first, and part of the time-out for the crash is subsumed in the transitional view installation intended to remove the mutant message fault. Curves in the third cluster are all higher than 4 seconds, because they involve a full time-out of 4 seconds needed to detect the second fault during the transitional view. Compare the curve for *Crash & New-View time-out* with the curve for *Crash & bad New-View*. In both cases, the second fault is injected in the first phase of the transitional view. The former involves a time-out of 4 seconds, while the latter does not. We observe that the difference between the times for these curves across various group sizes is around 4 seconds, which indicates that

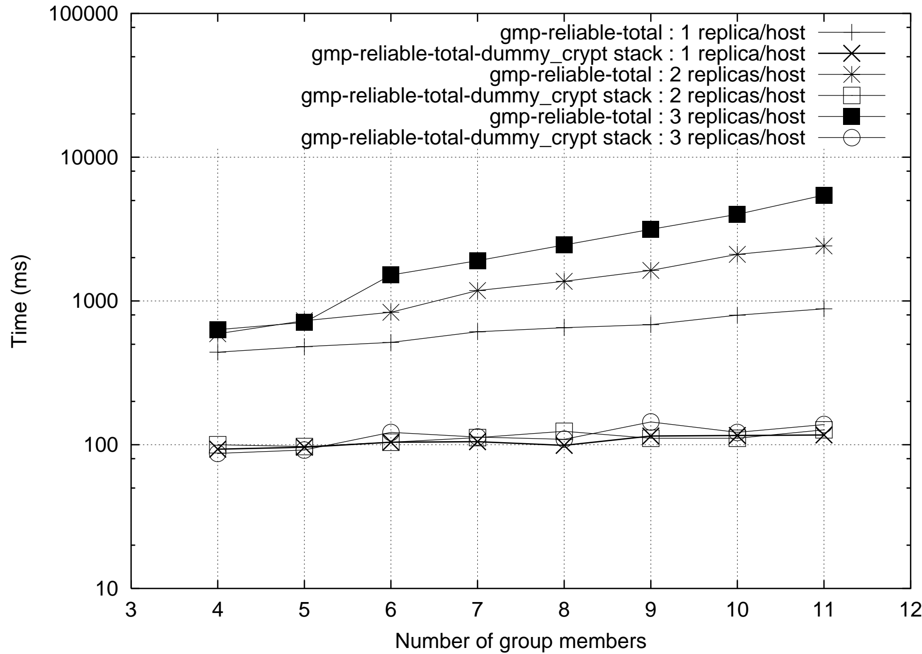


Figure 4.5: Variation of view installation time with load (single crash fault)

if not for the time-out employed, the cost for the view installation would be approximately the same in both cases.

To study the performance impact of having multiple group members on the same host, we placed more than one replica in an otherwise unloaded host. Figure 4.5 shows the performance results for the single crash fault case when we had one or two or three replicas per host for the **gmp-reliable-total-dummy_crypt** and **gmp-reliable-total** stacks. For the same group size, the communication costs associated with the view installation are the same in all three cases (one replica per host, two replicas per host, and three replicas per host). This can be observed from the three curves at the bottom of the graph, which show the view installation times for the **gmp-reliable-total-dummy_crypt** stack. The top three curves in the graph show the view installation times for the **gmp-reliable-total** stack with all cryptographic functions. The differences among the view installation times for these curves are mainly due to the differences in time needed to complete the cryptographic operations. The difference becomes larger as the number of cryptographic operations increases with increasing group size, thereby placing additional load on an already loaded processor. This highlights the fact that cryptographic overheads will be significantly greater when computing power is at a premium.

Figure 4.6 shows the effect on the view installation times of using time-outs for fault detection in the triple faults case. The curves in the top cluster involve two transitional

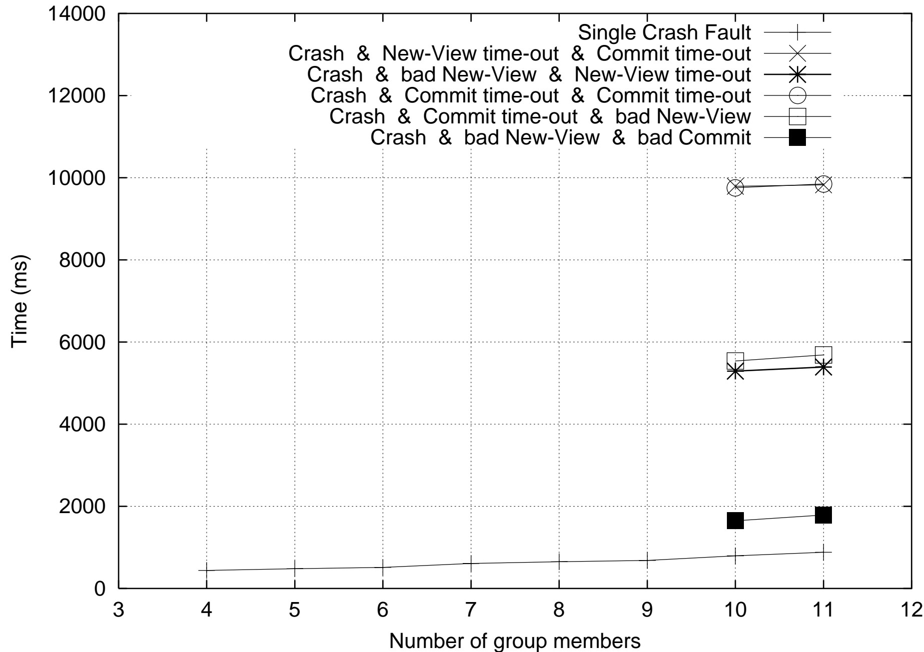


Figure 4.6: Effect of using time-outs for fault detection in transitional views

views, both of which use time-outs (of 4 seconds) to detect faults. The curves in the middle cluster also involve two transitional views, but have just one time-out. The curves in the lower cluster do not involve any time-outs. The difference in values between the top and middle cluster, and between the middle and lower cluster, would be about 4 seconds (the time-out value), suggesting that if not for the time-outs, the time taken for the other parts of the view installation is about the same in all cases. We also observe that the view installation time for a group size of 10 for the single-fault case differs from that for the triple-faults case (which does not involve any time-outs) by about 1 second. This large difference is due to the fact that in the triple-faults case, the view installation time also includes the time to *detect* two additional faults.

The results obtained for the intrusion-tolerant group membership protocol can thus be summarized as follows. The overhead for tolerating malicious faults due to intrusions is significant, compared to the overhead for tolerating just benign faults, like crashes. For most applications that use replication to achieve intrusion tolerance, a group size of 7, which protects from two simultaneous malicious faults, should be sufficient. For that group size, if the fault detection does not involve a time-out, the view installation time using the intrusion-tolerant group membership protocol is still less than a second. If fault detection does involve time-outs, then the time for changing the group membership can be significantly higher, based on the time-out value. Hence time-outs should be fine-tuned, so that they are not too

large to cause an unacceptable performance drop during recovery from simultaneous faults, but not too small to cause many spurious suspicions. Cryptographic operations account for a large percentage of the cost of tolerating malicious faults. This percentage increases with increasing group size, and decreases with the increasing availability of computing resources.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis described research work involved in building a group membership protocol that can provide consistent group membership in the presence of malicious faults resulting from intrusions. We inserted an implementation of this intrusion-tolerant group membership protocol, and other protocols that provide ordered reliable message delivery, into an existing crash-tolerant group communication system (C-Ensemble) to form a working prototype of an intrusion-tolerant GCS. We then tested this prototype with multiple applications. We devised various attack scenarios by making a subset of the process group faulty, and tested the ability of the GCS to withstand such attacks. We then did an extensive study [RPL+02] of the costs incurred when tolerating these attacks.

Our performance results confirmed the notion that the ability to tolerate malicious faults does not come without cost, and also provide more detailed information than was previously available about the cost of removing corrupt members from a process group. The results for our group membership protocol show that the cost of removing the corrupt member(s) depends on the detection mechanism used to detect the fault(s) (detection mechanisms based on timers are slower than mechanisms based on direct examination of message contents or patterns), and, if multiple faults occur, depends on the phase of recovery in which the second and third faults are activated. Whether this cost of tolerating faults is acceptable depends on the application considered. In all scenarios studied, the most significant contributor to the cost was public-key cryptography. That leads us to believe that specialized hardware, faster machines, or symmetric cryptography [CL99a](and modified protocols) could be used to reduce the overall cost significantly. We believe that our work would be useful both to application designers, who want to structure distributed applications in such a way that the costs are acceptable (for example, by building distributed objects that are heavy enough

that the cost of the group communication system is a small fraction of the total method invocation cost), and to protocol designers, who want to gain insight into how to build better intrusion-tolerant group communication systems.

5.2 Future Work

Although we have provided informal proofs about the correctness of the group membership protocol, we have not yet formally validated the protocol. Formal validation of complex distributed systems is known to be a tedious task, but is still an important one, because the creation of such systems is notoriously error-prone. Such a formal validation of our intrusion-tolerant protocols is an important future research step.

We are currently investigating alternatives to public-key cryptography, like using a combination of symmetric cryptography and public-key cryptography in the group membership protocol. That would require modifications to the current group membership protocol, but we believe that it would result in a significant reduction in the cost of recovering from faults.

Our protocol has not addressed the possibility of adjusting the time-outs in fault detection dynamically, but that would be an interesting subject for future work. Dynamic time-outs could be adjusted to reflect the current network load and the load on the other members' hosts.

In our experiments, the cost for cryptographic operations is high compared to the communication costs. However, it should be borne in mind that all of our experiments were conducted on a LAN. Deployment of our intrusion-tolerant GCS in a WAN environment may require modifications to the protocols to take into account the unpredictable message delays in such environments. That would be another interesting area for future research work.

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.

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

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

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

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

[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.

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

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

[Gro95] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.0. available at http://www.infosys.tuwien.ac.at/Research/Corba/OMG/cover.htm, July 1995.

[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.

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

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

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

[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.

[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)*, pages 24–26, Boston, MA, October 2000.

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

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

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

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

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

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

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

[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)*, Washington, DC, June 2002. To appear.

[Sch90] Fred Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

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

[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.