Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems *

HariGovind V. Ramasamy†, Prashant Pandey††, James Lyons†, Michel Cukier‡, and William H. Sanders†

University of Illinois†
Urbana, IL 61801, USA
{ramasamy,jlyons,whs}
@crhc.uiuc.edu

IBM Almaden Research Center††
San Jose, CA 95120, USA
ppandey@us.ibm.com

University of Maryland‡
College Park, MD 20742, USA
mcukier@eng.umd.edu

Abstract

Group communication systems that provide consistent group membership and reliable, ordered multicast properties in the presence of faults resulting from malicious intrusions have not been analyzed extensively to quantify the cost of tolerating these intrusions. This paper attempts to quantify this cost by presenting results from an experimental evaluation of three new intrusion-tolerant microprotocols that have been added to an existing crash-fault-tolerant group communication system. The results are analyzed to identify the parts that contribute the most overhead during provision of intrusion tolerance at the group communication system level.

1 Introduction

An *intrusion-tolerant system* is equipped with mechanisms that allow it to continue to operate in the presence of malicious intrusions even when significant portions of it have been compromised and may be in the control of an intelligent adversary. One promising approach is to provide intrusion tolerance at the middleware level by providing intrusion-tolerant services (such as remote method invocations) to distributed applications. Ongoing projects that aim to do that include MAFTIA [9], ITDOS [8], and ITUA [7], among others, and combine ideas from fault-tolerant computing, cryptography, and computer security. Several of the projects use replication to provide availability in the presence of arbitrary faults resulting from intrusions.

Group communication systems are a well-known paradigm for addressing a key concern in such systems: maintaining the consistency of replicated information. Most group communication systems are only able to tolerate crash failures. More recently, some group communication systems, like the Practical Byzantine Fault Tolerance (PBFT) [10] algorithm, the Rampart security toolkit [2], and the SecureRing Group Communication system [12], have been designed to function properly despite malicious corruption of some processes.

The performance of such algorithms must be carefully analyzed if their applicability to the building of intrusion-tolerant systems is to be understood. While others have provided some analysis of their algorithms (e.g., the PBFT protocol has been shown to have low overhead in the fault-free case [10]), there has not been a comprehensive study that provides information regarding the overhead of providing intrusion tolerance in both fault-free and fault conditions. In this paper, we describe the results of an extensive experimental analysis of the cost, in terms of reduced performance, incurred because of providing intrusion tolerance during both normal operation and recovery from intrusions. To do the analysis, we devised intrusion-tolerant versions of several key group communication protocols, namely those that provide reliable multicast, total ordering within a group, and group membership operations. We then inserted implementations of the protocols into an existing crash-tolerant group communication system (C-Ensemble, developed by Mark Hayden [13]) and instrumented them to provide detailed information about the cost incurred during fault-free operation and when tolerating both single and multiple correlated intrusions. The results provide new insights into the cost of providing intrusion-tolerant group communication, and suggest ways that this cost could be reduced in the future.

The organization of this paper is as follows. Section 2 describes the three new microprotocols added to the C-Ensemble framework. Section 3 presents the performance results of those protocols. Finally, in Section 4, we summa-

*This research has been supported by DARPA contract F30602-00-C-0172.

rize the results and conclude by describing ways to decrease the cost of tolerating malicious intrusions.

2 Intrusion-Tolerant Protocols

Key properties that intrusion-tolerant group communication systems must provide to process groups include ensuring that all multicast messages are delivered to all correct processes; enforcing a total order among multicast messages, to help build replication protocols that use the state machine approach [15]; and maintaining consistent information about group membership. This section describes protocols that provide those properties despite the presence of intrusions. Before focusing on the individual protocols, we describe the system model they use.

2.1 System Model and Assumptions

We consider a *timed asynchronous* distributed system [1]. The system 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.

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 integer identifiers or *ranks* from 0 to $|V| - 1$. The processes are denoted by p_0, p_1, ..., $p_{|V|-1}$. In general, the process p_k has a rank k. 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 at most $f = \lfloor (|V| - 1)/3 \rfloor$ corrupt processes. 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 and is able to obtain the public keys of other processes to verify signed messages.

We assume that all processes are computationally bound. That means that a corrupt process cannot find two messages with different contents and the same digest, and it cannot produce a valid signature of a correct process, or compute the message summarized by a digest, from the digest. We also assume that private keys cannot be stolen from correct processes.

2.2 Reliable Multicast Protocol

Our reliable multicast protocol (like other similar protocols) takes the approach of sending cryptographically signed messages to guarantee that a multicast message is delivered properly (without a change in its contents) to all correct processes, even in the presence of corrupt senders. It takes the common approach of using message buffering, sequence numbers, and positive and negative ACKs to address the issues that arise in an unreliable network, such as messages getting lost, reordered, and delayed.

The reliable multicast protocol we evaluate provides the properties described below, which are similar to those provided by SecureRing [12] and Rampart [3].

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

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

FIFO If p and q are correct, and q delivers two messages m_1 and m_2, in that order, from p, then p must have multicast m_1 and m_2 in that order.

The reliable multicast protocol consists of three phases. Each of the phases can result in recipients of messages suspecting the sender due to behavior that does not adhere to the protocol. In the first phase the sender (say p_i) buffers a copy of message m, and assigns to it the next available sequence number[2] s. The sender then creates a digitally signed digest of a data segment $\{m, i, s\}$ and sends the digest to the group along with i and s. Each recipient ascertains that it hasn't received another digest or message with sequence number s from p_i. In the second phase, each recipient process creates a signed reply to the message and sends this reply back to the sender of the digest. The sender in turn checks that the reply is indeed for the message it sent. The sender waits until it receives $2f + 1$ replies. In the final phase, the sender collects received replies and then sends

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

[2] The sequence numbers are local to each sender and are unrelated to sequence numbers used by other protocols.

out the actual message with the $2f + 1$ signatures attached. On receiving such an authenticated message, the recipient checks the validity of the $2f + 1$ attached signatures and accepts the message. The accepted messages are stored in buffers and delivered in the order of their sequence numbers (per sender). Each of these phases is protected from network errors through the use of buffering, sequence numbers, and positive and negative acknowledgments.

A detailed description of the phases, along with informal proofs that the protocol provides the properties described above, is given in [6].

2.3 Total Ordering Protocol

The total-ordering protocol we evaluate [6] ensures that sequence numbers assigned to messages by different processes are globally unique by partitioning the set of all possible sequence numbers and assigning a partition to each process. Each process p_i is associated, at view installation time, with an initial sequence number seq_orig_i and a sequence-number-generating function gf_i. The gf_is and seq_orig_is for all of the member processes in a particular view are known to all group members. Each function is monotonically increasing, and $f_i(x) > x$. Each process p_i generates a series of sequence numbers that starts with seq_orig_i; each subsequent sequence number is generated through the application of gf_i to the previously generated sequence number. Messages are generated asynchronously by the group members and each correct process sends messages with its generated sequence numbers, using one sequence number per message in increasing order. The sets of sequence numbers ($S_i = \{seq\_orig_i,\ gf_i(seq\_orig_i),\ gf_i(gf_i(seq\_orig_i)),\ \ldots\}$) generated by a correct process have the following properties:

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

The total-ordering protocol at each process delivers incoming messages in the sequence number order. The protocol can be held up by a process that does not send a message with a particular sequence number. We avoid that problem by forcing group members to transmit protocol-level messages with no payload (*null* messages) if they don't have any other messages to send. All processes monitor the progress of other processes. If some process is not sending any messages, and thus is holding up the progress of the protocol, this fact is reported to the fault detector implemented by the group membership protocol of the GCS. This scheme can be seen as an example of a "born-order" protocol [16] for total ordering, in which the messages contain information about the order in which they should be delivered.

The efficiency of this protocol depends on the relationship between the actual message traffic generated by the group members and the ordering forced by the sequence numbers assigned to them. If there is a close match between the two, this protocol will be more efficient than protocols that need sequencers or depend on some form of distributed consensus, because the extra step of deciding the order is avoided. We expect that the protocol will perform well in the ITUA project [7], since our intended application in that project has a predictable pattern of message traffic under normal circumstances. For a process group of size n in which all members generate similar traffic, a good set of generating functions is f_i such that $f_i(x) = x + n$. All the functions are the same, but the initial sequence number assigned to each process is its rank (ranks are unique to each member and are integers in the range $[0, n-1]$). The protocol will work even if a bad choice is made for the generating functions, but the penalty because of *null* messages will be high, leading to low performance.

The total-ordering protocol depends on the services of a reliable multicast protocol (like the protocol described in the previous section) that guarantees that FIFO ordering (per sender) is available. The total-ordering protocol provides the following property:

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

An informal proof that our total-ordering protocol indeed provides this property is given in [6].

2.4 Group Membership Protocol

The intrusion-tolerant group membership protocol we evaluate ensures that all correct processes maintain consistent information about the current membership of the group in spite of intrusions that may occur. In providing this function, the group membership protocol relies on a reliable multicast protocol (such as the one described in Section 2.2) to deliver the messages it sends to maintain group membership. The group membership protocol is responsible for maintaining group membership information, removing processes from the group, and joining new processes into the group. It has the properties described below, which are similar to those provided by [12] and [4].

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

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

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

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.

In this paper, we discuss how the protocol acts to remove corrupt member(s) from the group. The interested reader is referred to [5] for details about how new members can be added to the group, pseudo-code-level description of the entire group membership protocol and informal proofs that the protocol satisfies the abovementioned properties.

2.4.1 View Installation to Remove a Single Corrupt Member

We represent the group membership protocol as a set of communicating finite state machines, each as shown in Figure 1. In this section, we exclude the possibility that additional processes will exhibit faulty behavior[3]. Upon initialization, all state machines are in the NORMAL state. The group membership protocol provides an interface *suspect(process-rank i, reason R)* to the microprotocols of the group communication system. At a correct process, this function will be invoked if the process detects that another member has deviated from its specified behavior; a corrupt process may invoke this function at any time. When the function is invoked, the group membership protocol of the process that suspects another process will broadcast a signed *Suspect* message to the group 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. That marks the initiation of a three-phased view installation procedure, which is a series of steps at the end of which the member suspected to be corrupted by $f + 1$ other members will be removed from the group. 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, then the leader is suspected to be corrupt; hence, the deputy is expected to become leader and take action. When the leader sees $f + 1$ *Suspect*s, it broadcasts a signed *New-View* message, which contains 1) the list of endpoints for the next view that excludes the corrupt member, and 2) justification for this exclusion in the form of $f + 1$ *Suspect* messages received from the group.

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 does 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 process changes its state to WAITING-PHASE2. If it is a non-leader, it starts a timer, and expects the leader to take action before the timer expires. If it is a correct leader, it 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-View*s 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 each member 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. After that phase, each correct process that is a member of the new view installs a new protocol stack 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.

2.4.2 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) indicating 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 the other corrupt members. After sending this *Need-More-Change* message, p_k starts a timer, and expects a *Ready-to-Switch* message or

[3]However, we do tolerate that scenario. A description of how we do so is in Section 2.4.2.

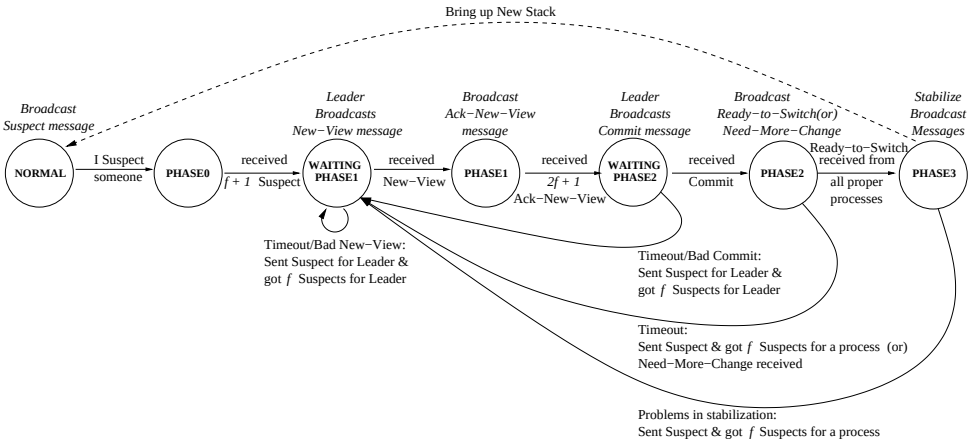


Figure 1. Finite state automaton for view installation

a *Need-More-Change* message from each of the members of the view proposed by the last *New-View* message, except the corrupt ones, before timer expiry. After receiving those *Ready-to-Switch* or *Need-More-Change* messages, p_k changes its state to WAITING-PHASE1 (as shown in Figure 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 (as shown in Figure 1). 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. That is shown in Figure 1 by the reverse transitions from the states PHASE2 and PHASE3 to the state WAITING-PHASE1. 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.

After p_k moves to WAITING-PHASE1, the view installation follows the procedure outlined in 2.4.1, 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 the *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.

3 Performance Measurement

In this section, we quantify the cost of the reliable multicast, total order, and group membership protocols described in the previous section. The reliable multicast and group membership protocols frequently use digital signatures to verify the authenticity of messages received. One of the changes we had to make to the C-Ensemble infrastructure was the addition of cryptographic support. We used Peter Gutmann's Cryptlib [14] as the core cryptographic library, and wrote wrapper functions around it.

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.

3.1 Results for Message Delivery

To measure the different costs associated with the message delivery protocols, we devised an application in which

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

The results reported here were obtained from 10 independent runs; each run collected data for varying parameter values. The same application was run on top of four different protocol stacks:

mnak-no_total: This stack has no total-ordering protocol, and the reliable delivery property is provided by the *mnak* protocol from C-Ensemble [13], which tolerates only crash faults. This stack does not use cryptography.

reliable-no_total-dummy_crypt: This stack includes the intrusion-tolerant reliable delivery protocol, but has a dummy version of the cryptography library that returns from cryptographic function calls immediately without performing the actual cryptographic operations.

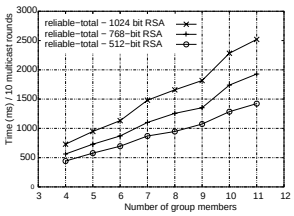
reliable-no_total: This stack includes the reliable delivery protocol (with normal cryptography functions), but does not provide total-ordering guarantees.

reliable-total: This is the complete stack with both reliable delivery (with normal cryptographic functions) and total-ordering protocols.

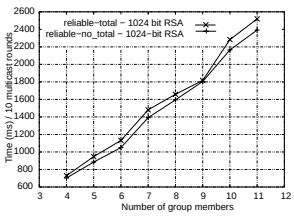
For each stack configuration, a run varied the *group_size* from 4 to 11, and if the stack used cryptography, *key_size*s of 512, 768, and 1024 were used.

The stack configurations were chosen so that we would be able to compare the individual costs of various factors. Comparing the message delivery times for the **reliable-no_total** and **reliable-total** stacks gives us a good estimate of the additional latency caused by the total-ordering protocol. The difference between delivery times for **reliable-no_total-dummy_crypt** and **reliable-no_total** represents the overhead caused by computation needed for cryptography. It should be noted that the reliable protocol depends on cryptography for correctness, and that the **reliable-no_total-dummy_crypt** stack does not provide intrusion tolerance. Another comparison we made was between the **reliable-no_total-dummy_crypt** and **mnak-no_total** stacks to see the overhead caused by the phases of sending the digest and collecting replies before sending the actual message.

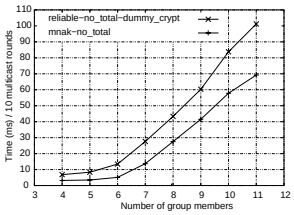
The variations in time of completion of the application with the reliable and total protocols with changing group and key sizes can be seen in Figure 2(a). It is interesting to



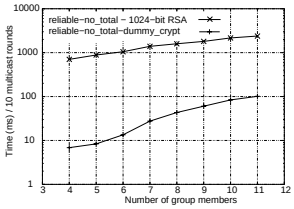
(a) Effect of increasing group and key sizes



(b) Cost of total ordering layer



(c) Overhead of the extra messages in reliable multicast



(d) Cost of cryptography in reliable multicast

Figure 2. Performance results for reliable multicast and total ordering

note that for all key sizes, there is a sharp increase in time for completion for group sizes 7 and 10. This is because 7 and 10 are of the form $3f + 1$, and the number of faults being tolerated changes at those group sizes. Thus, when the group size changes from 6 to 7, the number of faults tolerated, $\lfloor (n-1)/3 \rfloor$, goes from 1 to 2. Correspondingly, the number of replies[4] that the sender of a message digest has to wait to collect before sending the message ($2f + 1$) changes from 3 to 5. There is a similar change when the group size goes from 9 to 10. The figure also shows that increasing key size causes additional overheads. The differences become more pronounced for higher group sizes, because the message traffic increases, so that more messages need to be processed for the same number of multicast rounds.

As mentioned before, the efficiency of the total-ordering protocol depends on the closeness of the match between the chosen sequence-number-generating functions and the actual traffic. In the measurement application, the group members generate similar traffic. The default generating function (see Section 2.3) is a good fit for that application. Therefore, the total-ordering scheme has low overhead, as can be seen from Figure 2(b). The total-ordering protocol slows down the delivery of messages because it forces the faster processes to wait for the slower ones. If some process is very slow compared to others, *null* messages are

[4]The sender counts his own signature as one reply, so the number of messages it waits for is actually 2 and 4 respectively.

exchanged, causing additional overhead. Finally, there is some slowdown due to the fact that every message delivered has to go through an extra protocol in the stack.

The cost of reliable multicast can be broken down into two parts. The first part of the cost is introduced by the cryptographic functions. The other part is caused by the exchange of extra messages needed before the actual message can be sent. To isolate that overhead, the time taken by the application when using the **mnak-no_total** stack was compared to the time taken when using the **reliable-no_total-dummy_crypt** stack. We expected the increase in cost to be less than threefold, because the two additional control messages that must be exchanged in order to authenticate the message are small and don't need the same delivery guarantees that a regular message needs. (The difference can be seen in Figure 2(c).) We thereby obtained a lower bound on the overhead of using the intrusion-tolerant reliable delivery protocol relative to the cost of using the crash-tolerant *mnak* protocol.

The performance costs due to the use of cryptography are shown in Figure 2(d), which compares running times for the application using only the reliable protocol with and without 1024-bit RSA cryptography. We see a 1 to 2 orders of magnitude difference between the performances depending upon group size. Such results were expected, because of the high computational costs of public-key cryptography.

3.2 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 been fault-injected. 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 that corrupt member. The member then starts the view installation protocol described in Section 2.4. At each correct member, we take time measurements whenever a fault has been detected and whenever 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.

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. The group membership protocol ensures that all processes join a single group. When the group size reaches the target group size, each process starts multicasting messages to all 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 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 (see Section 2.3) when application-level multicasts are stopped.

The 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 time, the group member is suspected to have crashed. Mutant messages are identified by the reliable multicast protocol. For 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 time, and reports the suspected member to the group membership protocol.

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 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. The fault at the leader will be detected by the time-out mechanisms described in Section 2.4 and result in a transitional view.

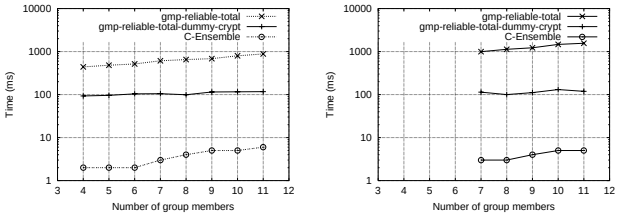
A group whose size is greater than 9 processes can tolerate three simultaneous faults (we call such faults *triple faults*). For groups larger than 9, we inject a crash fault into one of the group members, inject one of the faults in Table 1 at the leader, and inject one of the faults in Table 1 at the 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 3 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 the normal cryptography functions.

Table 1. Faults injected during a view installation to make it a transitional view

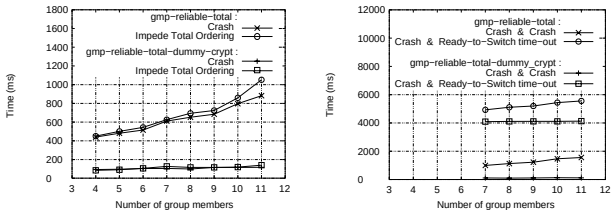
Fault	Fault Activation Point	Description
Bad New-View	start of PHASE1	Leader sends a *New-View* message with insufficient justification (less than $f+1$ signed suspects)
New-View Time-out	start of PHASE1	Leader does not send a *New-View* message after receiving $f+1$ suspects
Bad Commit	start of PHASE2	Leader sends a *Commit* message with insufficient justification (less than $2f+1$ signed *Ack-New-View*s)
Commit Time-out	start of PHASE2	Leader does not send a *Commit* message after receiving $2f+1$ *Ack-New-View*s
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 suspecting it.



(a) View installation time for single crash fault

(b) View installation time for double crash faults

Figure 3. Comparing gmp-reliable-total, gmp-reliable-total-dummy_crypt, & C-Ensemble stacks



(a) Single fault: with and without cryptography

(b) Double faults: with and without cryptography

Figure 4. Comparing gmp-reliable-total & gmp-reliable-total-dummy_crypt stacks for different faults

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 stack does *not* provide intrusion tolerance.

C-Ensemble: This is the original C version of Ensemble [13], 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 3(a) shows the comparison for the single crash fault case. Figure 3(b) shows the comparison for the double faults case, where two non-leader processes in the group are injected with crash faults. From Figures 3(a) and 3(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 the group size is very low in the **C-Ensemble** stack, 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. That increase is particularly pronounced in the **gmp-reliable-total** stack, when moving from group size 6 to 7, and from 9 to 10. That is because the number of messages that need to be collected and processed before progression to the next step of the view installation ($(f+1)$ *Suspect* messages, $(2f+1)$ *Ack-New-View* messages) increases, because f increases by 1.

Figure 4 shows comparisons between the **gmp-reliable-total** and **gmp-reliable-total-dummy_crypt** stacks, with additional fault types. From Figure 4(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. That is because, for a particular combination of double faults, the number of messages exchanged (communication cost) is the same for both stacks. The same observation applies to the single-fault case, shown in Figure 4(a). The increase in the cost with increasing group size is more pronounced for the **gmp-reliable-total** stack than the **gmp-reliable-total-dummy_crypt** stack. That indicates that the cost associated with cryptography increases more steadily than communication costs with increase in the group size.

Figure 5 compares the view installation times for different combinations of double faults. There are three clusters of curves in the figure. The one at the bottom is close to the curves for single faults; it corresponds to those combinations of double faults for which the fault detection does not rely on any time-outs. The view installation times for curves in this cluster are higher than those for single faults, because the values include the time to detect the additional

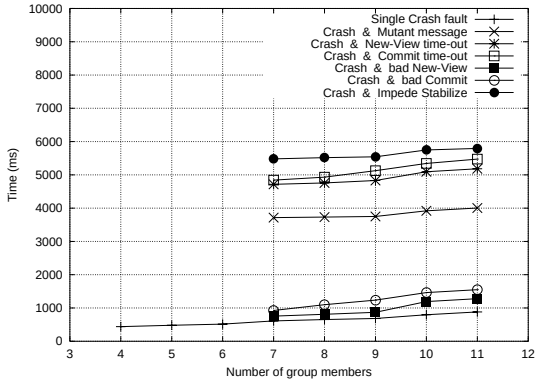


Figure 5. Comparing the view installation times for single and double faults

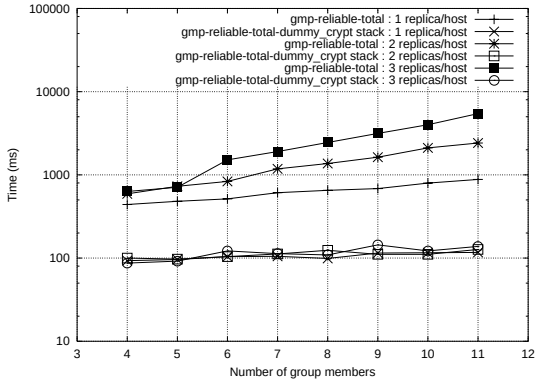


Figure 6. Variation of view installation time with load (single crash fault)

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. The third cluster of curves involved are higher than 4 seconds, because they involve a full time-out of 4 seconds needed to detect the second fault during the transitional view.

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 6 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 cost associated with the view installation is the same in all three cases (one replica per host, two replicas per host, and three replicas per host). That 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 sizes. That highlights the fact that cryptographic overheads will be significantly greater when computing power is at a premium.

Figure 7 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 views, both of which use time-outs (of 4 seconds) to detect faults. Each curve in the middle cluster also involves two transitional views, but has just one time-out. The curves in the lower cluster do not involve any time-outs. The curves suggest 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. Detection mechanisms based on time-outs are slower than mechanisms based on direct examination of message content or patterns. Fine-tuning time-outs so as to take into account the current network load and the load on the other members' hosts may significantly reduce this latency and also the possibility of excluding members who are genuinely suffering shortage of resources. Cryptographic operations account for a larger percentage of the cost of tolerating malicious faults, than communication does.

4 Conclusion

This paper provides an extensive study, under both fault and fault-free conditions, of the cost incurred when toler-

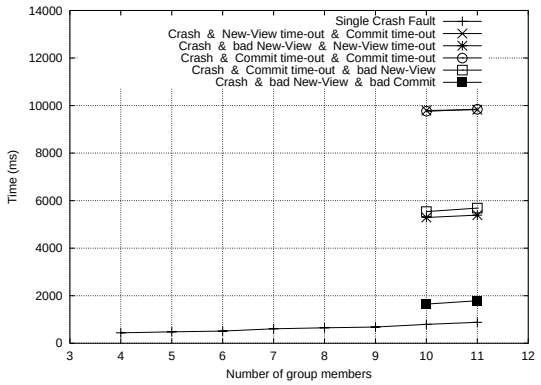


Figure 7. Effect of using time-outs for fault detection in transitional views

ating malicious faults due to intrusions. In the fault-free case, we showed the cost, relative to reliable multicast in C-Ensemble, of both intrusion-tolerant reliable delivery and intrusion-tolerant total order. We also investigated the cost associated with removing corrupt group member(s) when single, double, and triple malicious faults occur, for a representative selection of fault types. The results show that the cost of removing the corrupt member(s) depends on the detection mechanism that is used to detect the fault(s), and, if multiple faults occur, depends on the phase of recovery in which the second and third faults are activated.

In general, the results confirm the notion that the ability to tolerate malicious faults does not come without cost, and also provide more detailed information than was previously available. Whether the cost is acceptable depends on the application considered. In all scenarios studied, the most significant contributor to the cost was public key cryptography, which was especially significant under loaded conditions and for large group sizes. That leads us to believe that specialized hardware, faster machines, or symmetric cryptography [11] (and modified protocols) could be used to reduce the overall cost significantly. The results should be useful both to application designers, who can use them 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 can use them to gain insight into how to build better intrusion-tolerant group communication systems.

Acknowledgments: We would like to thank other members of the ITUA project, Michael Atighetchi, David Corman, Jeanna Gossett, Chris Jones, Joe Loyall, Partha Pal, Paul Rubel, Richard Schantz, Ron Watro, and Franklin Webber, for many useful discussions. We are grateful to Jenny Applequist for helping us to improve the readability of the paper.

References

[1] Flaviu Cristian and Christof Fetzer, "The Timed Asynchronous Distributed System Model," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 6, pp. 642-657, 1999

[2] Michael K. Reiter, "The Rampart Toolkit for Building High-integrity Services," *Lecture Notes in Computer Science*, Vol. 938, pp. 99-110, 1995

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

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

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

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

[7] Michel Cukier, James Lyons, Prashant Pandey, HariGovind V. Ramasamy, William H. Sanders, Partha Pal, Franklin Webber, Richard Schantz, Joseph Loyall, Ron Watro, Michael Atighetchi, and Jeanna Gossett, "Intrusion Tolerance Approaches in ITUA," FastAbstract in *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pp. B64-B65, 2001

[8] Brian Matt, Brian Niebuhr, David Sames, Gregg Tally, Brent Whitmore, and David Bakken, "Intrusion Tolerant CORBA Architectural Design," *Technical Report 01-007, NAI Labs*, 2001

[9] Paulo Veríssimo, Nuno Ferreira Neves, and Miguel Correia, "The Middleware Architecture of MAFTIA: A Blueprint," *Technical Report DI/FCUL TR 00-6, Department of Computer Science, University of Lisbon*, 2000

[10] Miguel Castro and Barbara Liskov, "Practical Byzantine Fault Tolerance," *Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, 1999

[11] Miguel Castro and Barbara Liskov, "Authenticated Byzantine Fault Tolerance Without Public-Key Cryptography," *Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science*, June 1999

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

[13] Mark Hayden, "Ensemble Reference Manual," Cornell University, 2001

[14] Peter Gutmann, "Cryptlib Security Toolkit," April 2001

[15] Fred Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, Vol. 22, No. 4, 1990

[16] Kenneth P. Birman, *Building Secure and Reliable Network Applications*, Manning, 1996