Experiences with building an intrusion-tolerant group communication system

HariGovind V. Ramasamy[1,*], Prashant Pandey[2], Michel Cukier[3], and William H. Sanders[4]

[1] *IBM Zurich Research Laboratory, Switzerland. hvr@zurich.ibm.com*
[2] *IBM Almaden Research Laboratory, USA. ppandey@us.ibm.com*
[3] *University of Maryland, College Park, USA. mcukier@eng.umd.edu*
[4] *University of Illinois, Urbana-Champaign, USA. whs@uiuc.edu*

SUMMARY

There are many group communication systems (GCSs) that provide consistent group membership and reliable, ordered multicast properties in the presence of crash faults. However, relatively few GCS implementations are able to provide those properties in the presence of malicious faults resulting from intrusions. We describe the systematic transformation of a crash-tolerant GCS, namely C-Ensemble, into an intrusion-tolerant GCS, the ITUA GCS. To do the transformation, we devised intrusion-tolerant versions of key group communication protocols. We then inserted implementations of the protocols into C-Ensemble and made significant changes to the rest of the C-Ensemble protocol stack to make the stack intrusion-tolerant. We quantify the cost of providing intrusion-tolerant group communication in two ways. First, we quantify the implementation effort by presenting a detailed analysis of the amount of change required to the original C-Ensemble system. In doing so, we provide insight into the choice of building an intrusion-tolerant GCS from scratch versus building one by leveraging a crash-tolerant implementation. Second, we quantify the run-time performance cost of tolerating intrusions by presenting results from an experimental evaluation of the main intrusion-tolerant microprotocols. The results are analyzed to identify the parts that contribute the most overhead while providing intrusion tolerance during both normal operation and recovery from intrusions.

KEY WORDS: Intrusion Tolerance, Fault Tolerance, Group Communication, Distributed Protocols, Experimental Evaluation

*Correspondence to: IBM Zurich Research Laboratory, Rueschlikon, CH 8803, Switzerland.
†A preliminary version of this paper [35] appears in the *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (DSN-2002).
Contract/grant sponsor: DARPA; contract/grant number: F30602–00–C–0172

Introduction

The ever-growing economic and strategic importance of distributed computer systems has made it important to ensure that these systems are *intrusion-tolerant*. An intrusion-tolerant system is equipped with mechanisms that allow it to continue to operate correctly even when significant portions of it have been compromised and may be in the control of an intelligent adversary.

Intrusion tolerance can be provided at various levels in a system: at the application level, the middleware level, the OS level, or the hardware level. One promising approach is to provide intrusion tolerance at the middleware level, providing intrusion-tolerant services (such as remote method invocations) to distributed applications. Examples of intrusion-tolerant middleware include MAFTIA [51], ITDOS [46], Enclaves [20], and ITUA [26, 33], some of which use replication to increase availability. A key concern in replicated systems is the need to maintain the consistency of information.

Group communication systems (GCSs) are a well-known paradigm for providing state consistency among the processes that constitute a group in the presence of faults. An early crash-tolerant GCS was the ISIS system [5, 6] from Cornell University. It led to numerous other academic efforts for developing crash-tolerant GCSs, e.g., Horus [8] and Ensemble [25] at Cornell University, Totem [31] at UCSB, Transis [2] at the Hebrew University, Spread [1] at Johns Hopkins University, and Newtop [21] at Newcastle University. Chockler et al. give an excellent survey of crash-tolerant GCSs in [14].

The growing need to strengthen systems against malicious attacks motivates the building of GCSs that can tolerate not just benign crash faults but also the malicious corruption of some group members. However, compared to work on crash-tolerant GCSs, work in the area of intrusion-tolerant GCSs has been gaining momentum only more recently, and relatively few implementations exist. As a system builder entrusted with the task of building an intrusion-tolerant GCS, one is naturally inclined to build it *de novo*, i.e., designing and implementing all the group communication protocols from scratch. Most, if not all, existing implementations of intrusion-tolerant GCSs were built using this approach. Examples of *de novo* implementations include the Rampart toolkit [41], SecureRing [28], Correia's GCS [15, 16]), and the CoBFIT GCS [39]. However, given the large number of existing crash-tolerant GCS implementations, an alternative approach would be to reuse a crash-tolerant GCS and transform it into an intrusion-tolerant GCS. In this paper, we attempt to answer questions relating to the feasibility, incentives, limitations, and challenges of this alternative and less-used approach.

We describe the systematic transformation of a crash-tolerant GCS, namely C-Ensemble, the C implementation of the well-known Ensemble GCS [25], into an intrusion-tolerant GCS, called the *ITUA GCS*. We highlight the main issues involved in engineering such a transformation and provide insights on how one can address them. The transformation was done at two levels. At the design level, we devised key group communication protocols in the fault model normally used for describing malicious attacks, namely the arbitrary or Byzantine [29] fault model. The protocols provide reliable, ordered multicast and consistent group membership properties. At the implementation level, we inserted implementations of the protocols into C-Ensemble and made significant changes to the rest of the C-Ensemble protocol stack to make its implementation intrusion-tolerant.

Our work demonstrates the feasibility of building an intrusion-tolerant GCS by leveraging a crash-tolerant implementation. In particular, our experience shows that a significant portion of the code base for a crash-tolerant GCS implementation can be reused for building an equivalent intrusion-tolerant one. We present a detailed analysis of the amount of change required to the original C-Ensemble system.

The transformation approach is not without limitations. The price to pay is the time and effort spent in doing an exhaustive *find-and-patch* campaign. In the presence of benign crash faults, a group member can blindly trust what other members say. However, the group member cannot afford to do so when some of the other members may be acting maliciously. The find-and-patch campaign involves identifying all parts of the crash-tolerant implementation reflecting mutual trust among group members and patching those parts to reflect a fault model in which group members may trust, but must always verify communication received from others. The patching we did included adding sanity checks for all messages received at a group member and adding cryptographic support to guarantee message integrity and non-repudiation properties. The limitation of such a find-and-patch campaign is that it can only be best-effort; no guarantees can be given that all places have been patched. However, that limitation is not very different from saying that even if an intrusion-tolerant GCS were developed *de novo* (as the authors have done in their CoBFIT GCS work [38, 39]), there would be no way to guarantee that the implementation was free of vulnerabilities.

The performance of such a transformed GCS must be carefully analyzed if its applicability to the building of intrusion-tolerant systems is to be understood. We describe the results of a detailed experimental analysis of the cost, in terms of the reduced performance incurred because of providing intrusion tolerance, both during normal operation and during recovery from (single and multiple correlated) intrusions. To do the analysis, we instrumented the key intrusion-tolerant protocol implementations to provide detailed information about the cost incurred during fault-free operation and when tolerating both single and multiple correlated intrusions. We also instrumented the equivalent crash-tolerant protocol implementations to compare the performance overhead of tolerating intrusions with that of tolerating just crashes. 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.

Related Work

We now describe related work, which includes *de novo* implementations of intrusion-tolerant GCSs (e.g., Rampart [41], SecureRing [28], and WIT-GCS [16]), implementations of Byzantine-fault-tolerant atomic multicast and state machine replication algorithms (e.g., the BFT library [11] and SINTRA [9]), and the development of intrusion-tolerant systems from crash-tolerant ones. The last category includes theoretical work such as [4, 27, 30] and practical implementations such as [19, 32].

Rampart [41, 40, 42] is a toolkit for building secure, fault-tolerant services. It has protocols for providing group membership services [42] and reliable, atomic multicast services [40] 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 [43] and a cryptographic key management service [44].

The SecureRing GCS [28] 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.

Correia et al.'s wormhole-based intrusion-tolerant GCS (WIT-GCS) [16] consists of two subsystems: a wormhole subsystem and a payload subsystem. The wormhole subsystem is a small, privileged, distributed subsystem that is timely, is synchronous, and fails only by crashing. The distributed parts of the wormhole subsystem are connected by secure channels. Group communication protocols, such as view synchronous atomic multicast [7] and group membership, have been implemented and experimentally evaluated within the payload subsystem, which relies on the strong properties provided by the wormhole subsystem.

Castro and Liskov's BFT protocol [11] is a state machine replication protocol that correctly survives Byzantine faults in asynchronous networks. Clients make requests to a replicated server group and then wait for enough identical replies from the server group. The protocol uses public-key cryptography to authenticate messages. Castro and Liskov also describe a modification of the protocol that uses message authentication codes in the fault-free case to reduce cryptographic overhead. The protocol does not rely on synchrony assumptions for safety, but makes synchrony assumptions only for liveness. The BFT library contains the implementation of the protocol. Castro and Liskov report extensive results for the experimental evaluation of the BFT library. However, the library is not a GCS, since it does not provide a group membership service and uses only static groups.

SINTRA [9] provides a suite of asynchronous protocols for intrusion-tolerant state machine replication on the Internet. The suite includes protocols for randomized binary Byzantine agreement, multi-valued Byzantine agreement, consistent broadcast, reliable broadcast, and atomic broadcast. A Java-based implementation of the protocols has been used to demonstrate and evaluate an intrusion-tolerant domain name service (DNS) [10]. However, like BFT, SINTRA is not a GCS, since it lacks a group membership protocol and supports only static groups.

Other previous work has also looked at the problem of building systems that can withstand malicious attacks out of systems that can withstand crash faults. Malkhi and Reiter [30], and later Kihlstrom et al. [27], extended unreliable fault detectors [13] from the crash fault model to the arbitrary fault model. Baldoni et al. [4] leveraged that work to propose a generic methodology to transform a crash-tolerant algorithm into an arbitrary-fault-tolerant one. At the protocol design level, one can view our arbitrary-fault-tolerant protocols as having been designed using the principles laid out in those papers. However, our work differs from the earlier work in that we give a systems' perspective of the transformation in the context of an

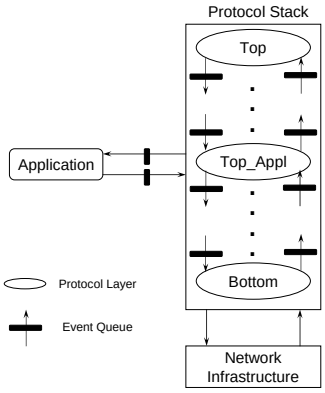


Figure 1. The Ensemble layering model

actual software implementation and describe the issues involved therein. Moreover, we also evaluate the performance costs of the transformation relative to the original crash-tolerant software.

Mpoeleng et al. [32] proposed an innovative extension of fail-stop processes [47] called *fail-signalling*, in which each process is replaced by a pair of self-checking processes. They then used those internally redundant fail-signal processes to extend a CORBA-compliant crash-tolerant GCS to one that tolerates authenticated Byzantine faults, and evaluated the performance of the resulting system. While their approach made it possible to treat a piece of crash-tolerant GCS software as a black box and to build wrappers around the software, it suffers from two drawbacks. First, it requires $n \geq 4f + 2$ nodes (instead of the usual $3f + 1$ nodes) to tolerate f simultaneous faults. Second, it assumes that the attacker cannot corrupt both of the nodes constituting a fail-signal process. A smart attacker can violate the properties of the GCS by compromising just the two nodes constituting a fail-signal process, irrespective of the value of the fault resilience f. Though our work did not treat C-Ensemble as a black box and required more effort at the development stage, it does not suffer from those drawbacks.

Following our approach, Drabkin et al. [19] have constructed an intrusion-tolerant GCS using the Ensemble GCS as a starting point. The GCS, called JazzEnsemble, focuses on providing better scalability and performance in the normal case of no failures. It includes implementations of failure detectors, protocols for vector consensus, uniform broadcast, and group management. Drabkin et al. also report extensive experimental results for the GCS in [19]. The present paper differs from [19] in that our focus is not so much on the protocols as it is on the engineering of the transformation from crash- to intrusion-tolerant GCS. Additionally, our experimental analysis provides more detailed insights into the cost of tolerating multiple correlated faults.

The Ensemble Architecture

To provide context for the rest of the paper, we familiarize the reader with the Ensemble architecture, which is shown in Figure 1.

Ensemble [24, 25] is a highly modular and flexible crash-tolerant GCS 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. Each layer makes certain assumptions about the properties provided by the other layers. As we go higher in the stack of layers, the level of abstraction usually increases.

Processes in a given process group have identical protocol stacks. Within the protocol stack of a process, a layer interacts only with the layer immediately above it and the one below it. The interaction takes place through events. Events travel up and down the protocol stack through *event queues* that connect adjacent layers. Each layer is implemented as a set of event handlers, which are invoked to process the events received from adjacent layers. Event queues are processed in FIFO order by a scheduler. Conceptually, each layer interacts directly with the corresponding layers in the stacks of other processes through events containing references to messages. A message consists of a payload (created by the application or by a layer) and a set of headers. Headers are added to the message by the layers as it passes through them. If the message is not processed at a layer, a special header, NOHDR, is added. The special header tells the corresponding layer at the recipient to pass up the received message without processing it. When an event containing a message reaches the *Bottom* layer, the associated message is transmitted over the underlying network. At the receiving process, the incoming messages are converted to *receive* events, which enter the stack at the *Bottom* layer. The layers retrieve the corresponding headers attached to the message. Although the application is conceptually thought of as being above the stack, it is placed as low in the Ensemble stack as possible (as shown in Figure 1) to minimize delays to the application messages.

Events enter the protocol stack in the following four ways:

1. A message for the stack is received from the network. An event handler in the *Bottom* layer handles the corresponding event.
2. The application sends a message, or sends some control information (e.g., to join or leave the group) to the stack. *Top_Appl*, the application's interface to the protocol stack, handles the corresponding event.
3. The environment surrounding the protocol stack, called the Ensemble *infrastructure*, sends some control information to the stack, e.g., during initialization of the stack. Such control events enter the stack at the *Bottom* or *Top* layers.
4. An event handler in a layer creates a new event while processing some event.

The group *view* refers to information related to the current membership of the group, such as the group identifier, network addresses, and *ranks* of all group members. A *rank* of a group member is the member's unique identifier within the view. A *view change* or group membership change consists of creating new protocol stacks with updated view information at all group members. The protocol stack has microprotocols for ensuring consistency of the view information across all correct group members.

Ensemble has many security features [45], but it can tolerate only crash faults. The security features mainly focus on protecting the authenticity and confidentiality of messages exchanged between the group members. The core idea is to use a symmetric group key for all intra-group

message exchanges. Ensemble has group rekeying protocols to obtain a new group key upon a group membership change or the compromise of the previous group key. While useful, the security features are not sufficient to provide tolerance against the malicious corruption of one or more members. As a consequence, Ensemble assumes that only mutually trusted processes can be part of the group.

From C-Ensemble to ITUA GCS: Motivation and Approach

When faced with the requirement to build an intrusion-tolerant GCS for the ITUA project [33], our decision to build the ITUA GCS from an existing crash-tolerant GCS was influenced in large part by evidence that suggested that the development time could be significantly less than that of building a *de novo* implementation. We observed that implementing a new GCS, irrespective of whether it is meant to be intrusion-tolerant or just crash-tolerant, is a large effort involving much more than a straightforward conversion of the pseudocode of the group communication protocols to code. The conversion can happen only after the implementation of several low-level services, such as event scheduling and notification; message buffering, marshalling, and de-marshalling; and reliable point-to-point communication, among others. Such support features form the underlying infrastructure upon which the protocols are built. Though the implementation of such support features is very well-understood, their number is not few. Consequently, the implementation of the infrastructure forms a significant portion of the total effort involved in building a new GCS. For example, in C-Ensemble, the infrastructure implementation accounts for about 65% of the total lines of code, and thus a large portion of the implementation effort. Almost all support features required to implement a crash-tolerant GCS are also needed (in a more intrusion-aware form) to implement an intrusion-tolerant GCS. That led us to believe that we could obtain significant savings in the system development time by reusing a crash-tolerant GCS instead of starting from scratch.

We decided to do the transformation from crash tolerance to intrusion tolerance in the Ensemble architecture because of its highly modular design. Due to Ensemble's layered architecture, it is possible to change a system incrementally, by replacing or modifying one layer at a time, while always having a working GCS implementation. It is also easy to validate the correctness of the individual microprotocols that were replaced, modified, or newly added. Layering also facilitates the performance evaluation of individual microprotocols and the performance comparison between intrusion-tolerant and crash-tolerant versions of the same microprotocol. One drawback of the original Ensemble implementation was that it was done 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, called *C-Ensemble* [24, 25], is the framework in which we developed our intrusion-tolerant GCS.

A closer examination of the C-Ensemble code revealed that the transformation would by no means be straightforward. It was clear that the C-Ensemble microprotocols that provide key group communication properties, namely those concerned with reliable ordered multicast message delivery and consistent group membership, had to be completely replaced with ones

that were designed to work in the arbitrary fault model. Other C-Ensemble layers that provided side functionalities had to be significantly modified to reflect the shift in the trust model among group members from *always-trust* (for crash-tolerant C-Ensemble) to *trust-but-verify* (for the ITUA GCS). Since C-Ensemble tolerated only crash faults and not malicious faults, group members trusted each other. An assumption in the crash fault model is that group members do not actively misbehave. For example, if any group member suspects that another member has crashed, and multicasts a suspicion to the rest of the group, the group will reconfigure to remove the suspected member. As another example, the group members follow the leader's suggestion regarding the set of multicast messages to deliver before switching to the next group membership. While trust among group members is perfectly acceptable when the fault model includes only crash faults, it is quite obvious that such trust had to be removed if we wanted the process group to be able to tolerate malicious faults; otherwise, by compromising a single group member, an attacker could subvert the whole group. The only way one can determine whether a layer requires modification to remove mutual trust is by closely inspecting the code for the layer. Of course, there is always a risk of overlooking places where modifications are necessary. However, the risk of overlooking vulnerabilities or creating vulnerabilities also exists when developing a new intrusion-tolerant GCS from scratch.

Overview of the Design of Key Intrusion-Tolerant Protocols

Intrusion-tolerant GCSs must provide process groups with several properties related to message delivery and group membership despite the malicious corruption of some group members. Messages multicast by a correct process must be delivered to all correct processes without change of contents. The *reliable multicast* protocol provides this property. Fault-tolerant applications designed using the state machine replication approach [48] require *atomic multicast*, a property that not only ensures reliable multicast, but also ensures that all correct replicas receive a set of incoming multicast messages in the same order. Atomic multicast in the modified C-Ensemble stack is provided by the combination of the total ordering protocol and the reliable multicast protocol. Process groups also need to maintain consistent information about group membership; faulty processes need to be detected and removed, and authorized new processes must be allowed to join the group. The *group membership protocol* provides those properties despite faulty members.

In this section, we provide an overview of the operation of the three key protocols and a specification of the properties they guarantee. A detailed description of the protocols accompanied by pseudocode and proof of correctness can be found in [34, 37]; we omit them here to keep the focus on the implementation and evaluation of the modified C-Ensemble stack. Before focusing on the individual protocols, we describe the system model they use.

System Model and Assumptions We consider a *timed asynchronous* distributed system [17]. 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). Timeouts are defined for message transmission and scheduling delays. When an experienced delay is greater than the associated

timeout delay, a *performance fault* is said to have occurred. This *timed* asynchronous system assumption circumvents the impossibility of consensus in an asynchronous environment [22].

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 $p_0, p_1, \ldots, p_{|V|-1}$. 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 (which we explain later) compared to the rest of the group. If corruption of the leader is detected, 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. Thus, a corrupt process cannot find two messages with different contents and the same digest. Furthermore, a corrupt process 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, or, equivalently, that a process whose private keys are compromised is corrupt.

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 implemented provides the properties described below, which are similar to those provided by SecureRing [28] and Rampart [40].

Integrity A correct process p_j delivers a message m allegedly multicast by a process p_i at most once, and if p_i is correct, only if p_i actually multicast m.

Agreement If a correct process p_i delivers message m in a particular view, then all correct processes also deliver m in that view.

[†]Group key management is an area that several works (such as [3, 49, 52, 53]) have examined in the past. In our paper, we do not focus on group key management and consider those works as complimentary to ours. In a real-world setting, one of the existing group key management schemes can be used for the generation of keys and their distribution to processes.

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

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 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. Note that the sequence numbers are local to each sender and are unrelated to sequence numbers used by other protocols. 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 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.

Total Ordering Protocol The total ordering protocol we implemented [34] provides the following property:

Total Order If correct processes p_i and p_j both deliver m_1 and m_2, then they deliver them in the same order.

The total ordering protocol delivers incoming messages in the sequence number order and 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. The protocol can be seen as an example of a "born-order" protocol [7] for total ordering, in which the messages contain information about the order in which they should be delivered. The protocol depends on the services of a reliable multicast protocol (like the protocol described above) that guarantees FIFO ordering.

At view installation time, each process p_i is associated with an initial sequence number seq_orig_i and a monotonically increasing sequence-number-generating function f_i, both of which are known to all other group members. The set of sequence numbers generated by a correct process p_i is $S_i = \{seq\_orig_i,\ f_i(seq\_orig_i),\ f_i(gf_i(seq\_orig_i)),\ \ldots\}$. 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 generated by all processes 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. The sets are pair-wise disjoint, i.e., $i \neq j \Rightarrow S_i \cap S_j = \phi$.

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 thereby stalling the progress of the protocol, this fact is reported to the fault detector implemented by the group membership protocol of the GCS.

Group Membership Protocol The intrusion-tolerant group membership protocol we implemented ensures that all correct processes maintain consistent information about the current membership of the group. The protocol is also responsible for removing processes from the group and joining new processes into the group. The protocol depends on a reliable multicast protocol (such as the one described above) to deliver the messages it sends. The group membership protocol provides the following properties, which are similar to those provided by Kihlstrom et al.'s protocol [28] and Reiter's protocol [42].

Agreement If p and q are two correct members of the view V_x, then 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. If q is not part of V_x but is included in V_{x+1}, then at least one correct member of V_x recommended q's addition to the group.

Liveness Suppose there exists a correct process in view V_x that is not suspected by $n - f$ correct members of V_x. Then,

- a process $p \in V_x$ that is suspected by $f + 1$ correct members of V_x will be removed from the group eventually, and
- a process $q \notin V_x$ that $f + 1$ correct members of V_x recommend for addition to the group will become a member of the group eventually.

We now provide an overview of the protocol for removing corrupt member(s) from the group. The interested reader is referred to [36] for details about adding new members to the group.

The group membership protocol provides an interface *suspect(rank, reason)* to the microprotocols of the GCS. At a correct process, this function is invoked if the process detects the deviation of another member from its specified behavior; a corrupt process may invoke this function at any time. When the function is invoked at a process p_i that suspects process p_j, the group membership protocol at p_i will multicast a signed *Suspect* message for p_j to the group.

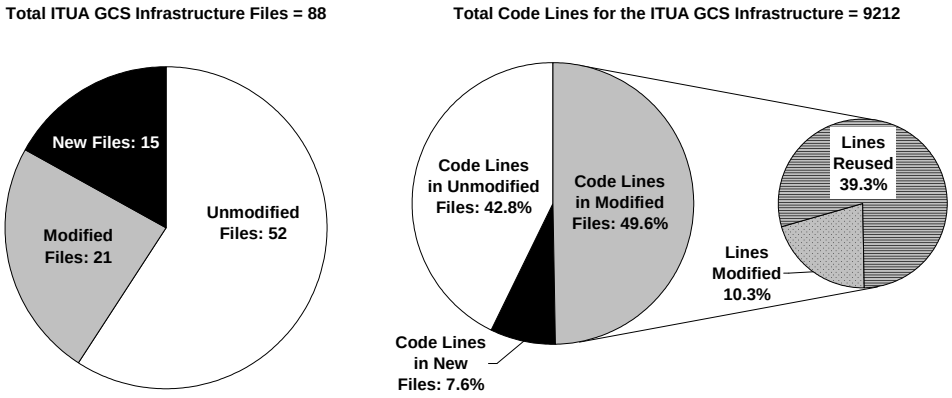
When $f + 1$ *Suspect* messages for a group member have been received at a process p_i, the process initiates a *view installation protocol*. The view installation consists of a series of steps at the end of which the member suspected to be corrupted by $f + 1$ other members will be removed from the group. If p_i is a non-leader process, then it 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. If p_i is the leader, then it multicasts a signed *New-View*

message, which contains 1) the list of processes 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 multicasts a signed acknowledgment, the *Ack-New-View* message. If a process p_k acknowledges a *New-View* message from p_j, then it does not acknowledge any more *New-View* messages from processes of lower rank than p_j in that view.

Suppose that a correct process p_i receives $2f + 1$ *Ack-New-View* messages for a *New-View* message. If p_i is a non-leader, it starts a timer, and expects the leader to take action before the timer expires. On the other hand, if p_i is a correct leader, it multicasts 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 multicasts 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, the members of the current view that are also members of the next view try to reach a consensus on which messages have hitherto been delivered by each of them. This is the *message stabilization phase*, which is needed to ensure that all correct group members deliver the same set of messages multicast in any given view, a property referred to as *view synchrony* [7, 23]. After that phase, each correct process that is a member of the new view installs a new protocol stack. 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.

An additional fault may occur during a view installation, resulting in the receipt of $f + 1$ *Suspect* messages for a member that is not among those processes being removed by the current view installation. In that case, a *Commit* message from the leader is acknowledged not with a *Ready-to-Switch* message, but with a *Need-More-Change* message 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 those corrupt members. After sending this *Need-More-Change* message, a correct process p_k starts a timer, and expects a fresh *New-View* message from the (possibly new) leader. The message should contain a view that excludes at least one more known corrupt member from the next view than it did in the last *New-View* message. If the additional fault was at the leader, then the deputy takes over as the new leader and multicasts a *New-View* message. The previous *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 multicast) are part of what we call a *transitional view*. There could be a cycle of transitional views until a *New-View* message finally excludes all known corrupt members. At that point, all three phases of the view installation would complete, resulting in a new protocol stack at the group members.



(a) File Metrics (b) Code Line Metrics

Figure 2. The ITUA GCS Infrastructure vis-a-vis the C-Ensemble Infrastructure

Implementation of the ITUA GCS

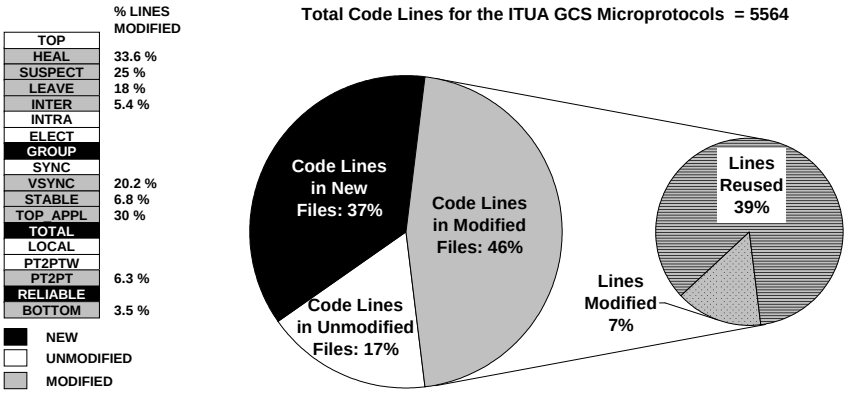
Changes to C-Ensemble's Infrastructure The C-Ensemble infrastructure provides the support services needed for implementing the group communication protocols.

Figure 2 quantifies the change made to the C-Ensemble infrastructure in order to obtain the ITUA GCS infrastructure. The two metrics used are the number of files (Figure 2(a)) and the number of code lines (Figure 2(b)). A line is counted to be a *code line* if it has any code in it. Figure 2 shows that a substantial portion of the C-Ensemble infrastructure was reused. Out of the 9212 total code lines in the ITUA GCS infrastructure, only 7.6% were due to files that were newly added to the ITUA GCS infrastructure. 42.8% were due to the C-Ensemble infrastructure files that were used unmodified for the ITUA GCS infrastructure. The remaining 49.6% of the code lines were due to modified C-Ensemble infrastructure files; however, only $(\frac{10.3 \times 100}{49.6} =)$ 20.8% or approximately one-fifths of the code lines in those files were actually modified, and the rest were directly reused without any change.

The changes we made to the C-Ensemble infrastructure include:

1. the definition of new events and messages associated with the implementation of the intrusion-tolerant protocols,
2. the implementation of new data structures that are common to many of the new protocols, and
3. the addition of routines for marshalling and de-marshalling new messages associated with the new protocols.

When the fault model consists only of crash faults, group members can be trusted to adhere to specified message formats. However, such trust is no longer possible in an arbitrary fault model. Without proper checks, a malformed message from a corrupted member could easily cause a correct recipient to crash. Hence, it was necessary to add sanity checks at many parts of the C-Ensemble infrastructure. Such sanity checks included message format checks on received messages and bounds checks on array and pointer references.



(a) Protocol Stack (b) Code Line Metrics

Figure 3. The ITUA GCS Microprotocols vis-a-vis the C-Ensemble Microprotocols

Since the intrusion-tolerant group communication protocols use message digests and exchange digitally signed messages, we added cryptographic support to the C-Ensemble infrastructure. We used Peter Gutmann's Cryptlib [18] as the core cryptographic library. Using Cryptlib, we wrote wrapper functions for generating and verifying message digests and digital signatures. For signature generation, we implemented the common approach of signing the digest of a message rather than the full message. The microprotocols in the ITUA GCS protocol stack make extensive use of these wrapper functions. Note that the files and the code lines in the Cryptlib library were not factored into the calculation of the numbers reported in Figure 2. Specifically, we did not factor in the `cryptlib.h` file, which is part of the Cryptlib package and provides a uniform interface to the library for the rest of the ITUA GCS code base.

To facilitate easy access to cryptographic key information by the ITUA GCS microprotocols, we updated the two main data structures related to group membership provided by the C-Ensemble infrastructure, namely the *view state* and *local view state* data structures. When a new view is installed, the infrastructure calls the initialization function of each layer of the new protocol stack with references to the *view state* and *local view state* records as arguments. The *view state* data structure stores group membership information such as the group identifier, the rank of the leader, and the network addresses of all group members. We added extra fields for storing the key IDs and the public keys of the group members in the order of increasing rank. The *local view state* data structure stores group membership information that is specific to the local process, such as the member's rank, its network address, and whether it is a leader or not. To the *local view state* data structure, we added extra fields for accessing the private key of the group member.

Protocol Stack for the Intrusion-tolerant GCS Figure 3 quantifies the amount of change made to the C-Ensemble protocol stack to obtain the ITUA GCS protocol stack. Figure 3(a) shows the customized ITUA GCS protocol stack that we used in our experimental

evaluation. The figure indicates the layers of the stack that were taken unmodified from the original C-Ensemble stack, the layers of the C-Ensemble stack that were modified, and the layers that were newly added. The figure also shows the percentage of the code lines in the modified layers that were actually modified compared to the total size of those layers. Figure 3(b) shows that 37% or approximately one-third of the 5564 code lines for the ITUA GCS microprotocols were concentrated in the three new layers, namely, the *Group*, *Reliable*, and *Total* layers. 17% were from C-Ensemble microprotocol files that were used for the ITUA GCS protocol stack unmodified. The remaining 46% of the code lines were from modified C-Ensemble microprotocol files; however, only ($\frac{7\times100}{46}$ =) 15.2% of the code lines in those files were actually modified, and the rest were directly reused without any change.

A significant portion of the intrusion-tolerant reliable, ordered multicast message delivery and group membership functionalities is implemented by the new *Reliable*, *Total*, and *Group* layers (Figure 3(a)). The *Total* and *Reliable* layers implement the total ordering and reliable multicast protocols, respectively. The *Group* layer, in conjunction with the *Heal, Suspect, Leave, Inter*, and *Intra* layers, provides the properties relating to consistent group membership. The *VSync*, *Sync*, *Top_Appl*, and *Stable* layers implement the message stabilization functionality needed for view synchrony.

Many other C-Ensemble layers did not have to be completely re-implemented, but had to be modified for use in the intrusion-tolerant GCS. Examples include the *Bottom*, *Pt2pt*, *Top_Appl*, *Stable*, *Vsync*, *Inter*, *Leave*, *Suspect*, and *Heal* layers. Tables I and II describe the original functionality in C-Ensemble for those layers, the key changes that we made, and the main reasons for those changes. A close examination of the tables would reveal that much of the modification to the C-Ensemble layers was related to removing the mutual trust among group members. Though many C-Ensemble layers required modification, as Figure 3(a) shows, only a small amount of modification was done relative to the original size of those layers.

There were other layers that did not require modification, either because they were implementing the Ensemble layering architecture and not any protocol functionality (such as the *Top* and *Local* layers), or because the protocol functionality in the crash model can be applied unmodified to the malicious fault model, e.g., the C-Ensemble *Elect*, *Intra*, *Sync*, and *Pt2PtW* layers. The C-Ensemble *Elect* layer chooses the lowest-ranked non-faulty group member as the leader. The intrusion-tolerant group membership protocol chooses the next leader in the same manner. The *Intra* layer forwards to the group any local group membership events such as fault detections and view change notifications. The *Sync* layer notifies higher-level microprotocols not to send further messages when a view change is underway. The *Pt2PtW* layer uses a credit-based flow control mechanism for process-to-process messages.

We now describe interesting issues that arose during the implementation of our intrusion-tolerant protocols, and how they were solved.

Reliable Multicast An important issue in the implementation of the reliable multicast protocol was keeping the message buffers bounded. At the end of the three phases of the reliable multicast protocol, it is guaranteed that no two correct processes will deliver different contents for a particular sequence number. However, if the sender is faulty, it can cause the delivery of the message at some but not all correct parties by sending the message in the final phase to a subset of the group. To handle such a faulty sender, the *Reliable* layer cannot immediately

Table I. Key Modifications to C-Ensemble Layers (Part I)

Layer	Original Functionality	Changes Made for the ITUA GCS
Bottom	This layer interacts with the underlying communication transport by sending and receiving messages and by scheduling or handling timeouts. The layer *bounces* other local events up the stack, i.e., it sends the events received from the next higher layer in the stack back to the same layer.	We modified the layer to bounce some newly defined local events.
Pt2pt	This layer implements reliable process-to-process message delivery by resending messages till it receives acknowledgments or ACKs for them. Out-of-order messages result in negative acknowledgments or NACKs, which cause messages to be resent without timeouts.	We added timeout mechanisms and limits on the number of retransmissions to prevent a malicious process from stalling the garbage collection of the protocol stack indefinitely after a view change.
Top_Appl	This layer is the application interface, and it buffers, transmits, and delivers application-level messages. During a view change, it conveys information about the number of application-level messages multicast or sent to the *Vsync* layer, which then conveys the information to the leader. Through the leader, the *Top_Appl* layer learns about messages that are expected to be delivered before switching to the next view. The *Top_Appl* layer also notifies the *Sync* layer when all the expected multicast and point-to-point messages have been received, at which time the *Sync* layer *unblocks* the group to indicate that normal group operations may be resumed.	We modified the *Top_Appl* layer so that the group members do not rely solely on the leader to help deliver the messages that are expected before the switch to the next view, but have not yet been received. The layer reports, as suspects to the *Group* layer, any members that claim to have received certain messages but are not re-transmitting them within a timeout interval.
Stable	This layer keeps track of the number of multicast messages from each member that are stable. Periodically, the layer unreliably multicasts (using IP multicast) that information in the form of a *stability array* to the group. A stability array from each group member is needed to decide which messages have to be delivered before switching to the new view.	We added mechanisms to report, as suspects to the *Group* layer, those group members from which a stability array was not received in a timely manner during a view change. The added mechanisms prevent a malicious group member from indefinitely stalling the progress of a view installation.
Vsync	At each group member, this layer keeps track of information about the number of application-level messages that were multicast and sent by the group member. Periodically, the layer sends that information to the leader, which in turn unreliably multicasts the consolidated information from all members in the form of a matrix.	The group members do not rely solely on the leader to act as a central point for gathering the information and disseminating the matrix. The information is unreliably multicast (using IP multicast) to all members. Each member calculates its own matrix from the consolidated information and unreliably multicasts it to the group.

Table II. Key Modifications to C-Ensemble Layers (Part II)

Layer	Original Functionality	Changes Made for the ITUA GCS
Inter	This layer handles view changes involving two groups, of which the one with more members acts as the *primary partition*, while the other is called the *merging group*. The leader of the merging group *blocks* its group (i.e., temporarily suspends application-level multicasts by any member in the group) and sends a merge request to the primary partition's leader. The primary partition's leader then blocks its group, installs a new view that includes the merging group's members, and sends the view to the merging leader. The merging leader then installs the view in its group.	Since the intrusion-tolerant group membership protocol can only handle the joining of new members one at a time, we modified the *Inter* layer to entertain merge requests only from singleton groups. We added more checks to be done before a merge request is considered favorably. For example, the public key of the process requesting the merge must be in the list of processes authorized to join the group, and the process must not have been removed previously for exhibiting a malicious fault. All members, and not just the primary partition's leader, receive and process merge requests. Instead of having the leader trigger a view change at will, we have the *Inter* layer invoke the agreement protocol, which is implemented by the *Group* layer, before allowing the requesting partition to merge.
Leave	The *Leave* layer initiates the sequence of events to garbage-collect the protocol stack after a view change. The layer notifies the leader when multicast messages have stabilized. When the leader has heard from all the non-faulty members, it instructs the group to garbage-collect the old protocol stacks.	We modified the layer so that the notification of that the multicast messages have stabilized is sent to the whole group, and not just the leader. Each member decides for itself when to garbage-collect based on the notifications it receives, without relying on the leader's instruction. Waiting for notifications from all non-faulty members before garbage collection is a vulnerability that could be exploited by a malicious member to stall the garbage collection. We fixed the vulnerability by adding mechanisms to report as suspects those group members that do not send their notifications in a timely manner. We also added state transfer mechanisms that allow for garbage collection after have been received notifications from only $n - f$ members of the next view.
Suspect	This layer implements a heartbeat mechanism [13] to check for suspected crash faults.	A malicious member can faithfully send heartbeats, but refuse to send other required messages. Hence, *Suspect* is no longer the only layer that detects faults. Heartbeat messages are signed to prevent spoofing. Suspicions for a member are carried over to subsequent views until the member is removed from the group.
Heal	This layer is used to merge partitions of a group. Periodically, a leader of the merging partition unreliably multicasts a message containing information about the group identifier, membership of the partition, and leader's identifier. The leader of the primary partition examines the message to see whether the requesting partition can be allowed to merge with the primary partition. If so, the *Heal* layer notifies the *Inter* layer.	Only singleton partitions are allowed to merge with the group. The message from the merging process contains the public key identifier of the process and is digitally signed so that it cannot be faked by a malicious process. All members of the primary partition (and not just the leader) examine the message and use it to decide whether to recommend the addition of the process to the group membership.

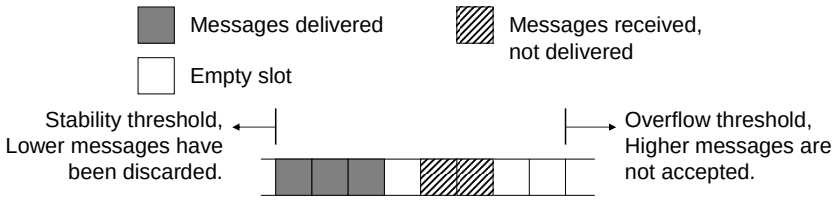


Figure 4. Message buffer for a single sender at a process

discard messages that it delivers. If a negative acknowledgment for a delivered message is received later, the message must be available so that it can be forwarded to members that did not receive it directly from the original sender. However, if messages are buffered indefinitely, the message queues will grow without bound, and the process will run out of memory.

The *Stable* layer helps keep the buffers of the *Reliable* layer bounded. Through the *Stable* layer, all processes exchange information about the number of messages that they have delivered. The *Stable* layer reports, as suspects to the *Group* layer, faulty members that refuse to send such information in a timely manner. It should be noted that, in contrast to the global set of sequence numbers employed by the total ordering protocol, the sequence numbers used by the reliable multicast protocol are per-sender. Once *all* group members agree through the *Stable* layer that messages up to a particular sequence number from a sender have been delivered, all processes can discard messages from that sender with lesser sequence numbers. Messages that all processes have delivered are considered to be *stable* messages, and the sequence number of the highest stable message is called the *stability threshold*. Processes ignore any messages or retransmission requests for sequence numbers less than the stability threshold. Processes also ignore messages with sequence numbers that are higher than an *overflow threshold*. The overflow threshold is obtained by adding a window size to the stability threshold. The overflow threshold helps prevent a corrupted process from filling up the message buffers of other group members by recklessly initiating a large number of multicasts. Figure 4 shows the message queue for a single sender.

Total Ordering Working together, the *Total* and *Reliable* layers provide atomic multicast, an important primitive for building fault-tolerant and highly available services using the state machine replication approach [48]. The *Total* layer operates on top of the *Reliable* layer and buffers the reliably delivered messages until they can be delivered in a total order. Typically, the application, represented by the *Top_Appl* layer in the ITUA GCS protocol stack, generates the messages for atomic multicast. The *Top_Appl* layer flags the events containing such messages as events that require processing by the *Total* layer. Upon receiving such an event, the *Total* layer enqueues the message in its message queue.

The efficiency of the total ordering 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, the protocol will be more efficient than protocols that need sequencers (such as Reiter's protocol [40]) or depend on some form of distributed consensus (such as the SINTRA atomic broadcast protocol [9]),

because the extra step of deciding the order is avoided. 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.

For the intended use of the GCS in the ITUA project [33], all group members were expected to generate similar traffic. Consequently, the generating function we implemented at a group member p_i was $f_i(x) = x + n$, where n is the number of members in the current view. The generating function was the same at all group members, and the initial sequence number assigned to each group member was its rank. We show through our experimental results below that the generating function is indeed a good fit for an application in which all members generate similar amounts of messages.

Group Membership The *Group* layer implements the data structures and algorithms for the view installation protocol. In C-Ensemble, the *Suspect* layer detects the crash of a group member by noticing missing heartbeat messages from the member. Upon detecting a crash, the *Suspect* layer sends down a suspicion event that directly triggers the installation of a new view. However, in the presence of malicious faults, the members must first agree on the next view before they install the new protocol stack. Accordingly, we modified the protocol stack so that the suspicion event generated by the *Suspect* layer is handled by the *Group* layer, resulting in the multicast of a *Suspect* message. The *Group* layer starts the new view installation only after receiving $f + 1$ *Suspect* messages for a group member.

The *Vsync*, *Top_Appl*, and *Stable* layers were modified to ensure that messages multicast in the current view are delivered at all correct processes before the group switches to the next view. We had to account for the possibility that new corruptions may be discovered during the message stabilization phase of the view installation. During view change, the members exchange information about the multicasts received from each other. We added data structures that indicate, for each sender, the process that has the highest-sequence-numbered multicast message from that sender. We also added timeouts for preventing malicious group members from stalling the message stabilization. In C-Ensemble, if a group member p claimed to have the highest-sequence-numbered multicast from another member, other group members trusted p's claim and waited until p resent the message. However, in the ITUA GCS, p could act maliciously and stall the view installation by not resending the messages it claims to have received. To prevent that situation, we modified the *Top_Appl* layer so that if p does not resend the messages in a timely manner, other group members suspect p. That will cause the *Group* layer at those members to multicast a *Suspect* message for p and will eventually result in p's exclusion from the group.

Experimental Evaluation of Protocols

We now quantify the cost of the reliable multicast, total order, and group membership protocols in both faulty and fault-free conditions. The tests were carried out on a testbed of ten 1GHz Pentium III computers with 256MB PC133 RAM; the machines were otherwise unloaded. The computers were connected by a full-duplex 100 Mbps switched Ethernet network. Unless specified otherwise, a single process ran on each machine.

Results for Message Delivery 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. The values of *group_size*, *num_init_casts*, and *key_size* are command-line arguments to the processes. We ran the same application above the following four different protocol stacks in order to compare the individual costs of various factors.

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

(2) **reliable-no_total-dummy_crypt** This stack includes the intrusion-tolerant reliable multicast protocol, but a dummy version of the cryptography library that returns from function calls immediately without performing the expensive operations to sign, verify, encrypt, or decrypt messages. This stack does not provide intrusion tolerance.

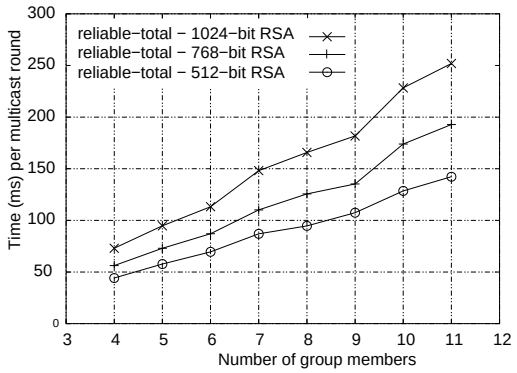
(3) **reliable-no_total** This stack includes the reliable delivery protocol, but does not provide total ordering guarantees. The reliable protocol uses the normal cryptography functions.

(4) **reliable-total** This is the complete stack with both reliable delivery and total ordering protocols. Again, the reliable protocol uses the normal cryptography functions.

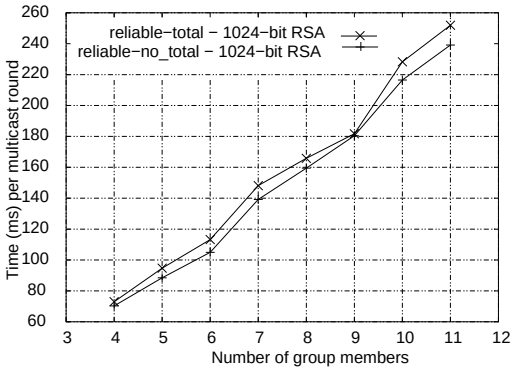
Comparing the message delivery times for the **reliable-no_total** and **reliable-total** stacks gives us a good estimate of the additional latency caused by the total ordering protocol. The difference between delivery times for **reliable-no_total-dummy_crypt** and **reliable-no_total** represents the overhead due to cryptography. 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 Figures 5(a) and 5(b). For all key sizes, there is a sharp increase in time for completion for group sizes 7 and 10. The reason is that 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 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 the key size causes additional overheads. The differences become more pronounced for higher group sizes.

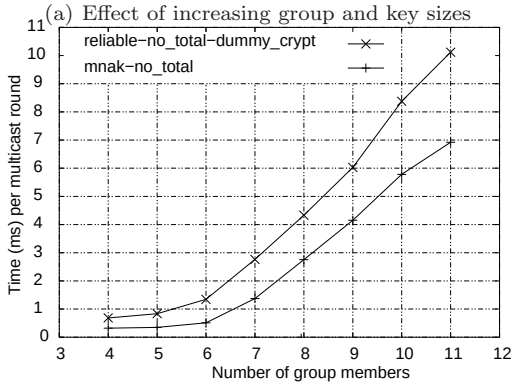
In the measurement application, the group members generate similar traffic. Since the default generating function divides the sequence numbers equally among the processes, it is a good fit for this application, and therefore the total ordering scheme has low overhead, as can be seen from Figure 5(b). 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 exchanged, causing additional overhead. Finally, there is some slowdown because every message delivered has to go through an extra protocol in the stack.



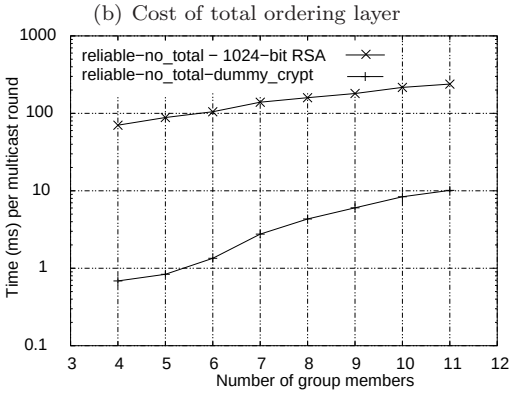
(a) Effect of increasing group and key sizes



(b) Cost of total ordering layer



(c) Overhead of extra messages in reliable multicast



(d) Cost of cryptography in reliable multicast

Figure 5. Performance results for reliable multicast and total ordering

The cost of reliable multicast consists of two parts. The first is the cost due to cryptography. The second is the cost caused by the exchange of extra messages needed before the actual message can be sent. To isolate the second overhead, we compared the change in time taken by the application when the C-Ensemble crash-tolerant reliable protocol (i.e., the *mnak* protocol that doesn't use cryptography) was replaced by our intrusion-tolerant protocol, which was modified to use dummy cryptographic functions. The comparison is shown in Figure 5(c). The increase in cost was expected to be less than threefold. The reason is that 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 performance costs due to cryptography are shown in Figure 5(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. The results confirm the high computational costs of public-key cryptography.

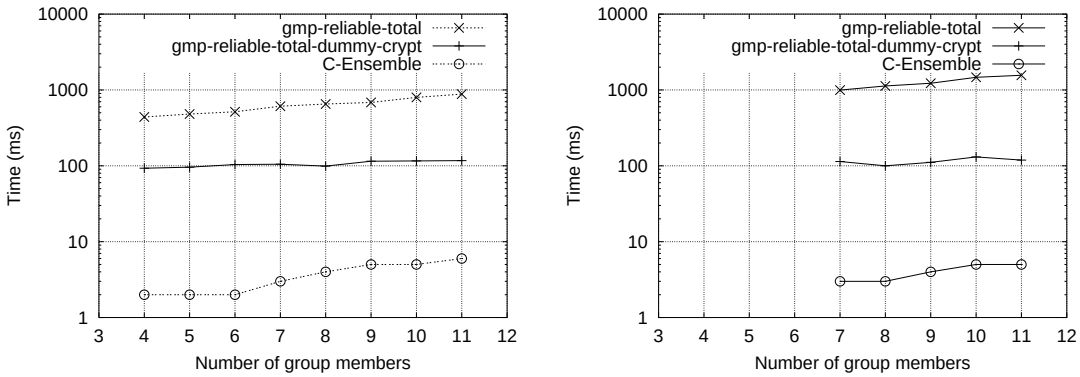
Table III. Faults Injected during View Installation to Obtain a Transitional View

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

Results for Group Membership We now quantify the cost of excluding corrupt 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 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 a corrupt member. The group member then starts the view installation protocol. 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 the times across all correct group members is the presented time for view installation for that particular group size. Since the nodes had identical hardware capabilities and the variance in message latencies among pairs of nodes was negligible (due to LAN interconnection), the standard deviation of the times measured for the individual group members was very small.

We used the following scenario to quantify the cost of removing corrupt group members. Each process is started with the same group name and the same intended 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 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*, which causes 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 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.



(a) View installation time for single crash fault (b) View installation time for double crash faults

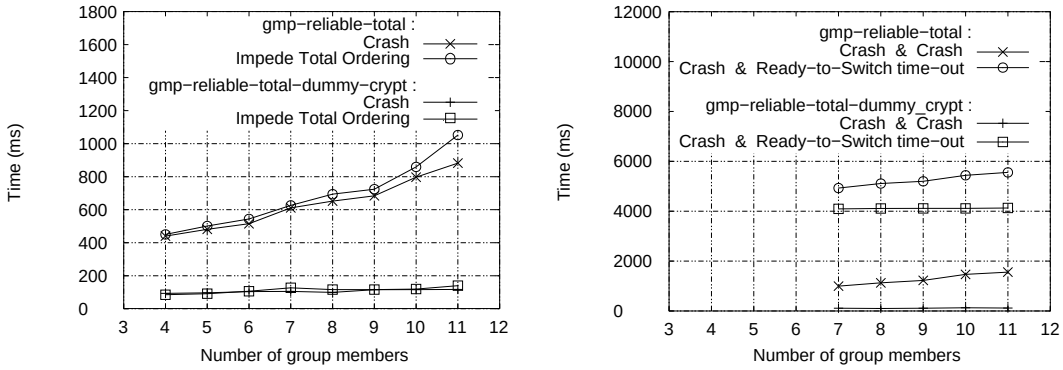
Figure 6. Comparing **gmp-reliable-total**, **gmp-reliable-total-dummy_crypt**, and **C-Ensemble** stacks

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 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 inject one of the faults in Table III 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 group membership protocol has timeouts and other mechanisms to ensure liveness. Those 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 III describes a list of such faults that can occur during the view installation, and the stage in the view installation when they are activated.

Figure 6 shows the view installation times for three protocol stacks:
(1) **gmp-reliable-total** This is the customized stack with the new intrusion-tolerant protocols for group membership, reliable multicast, and total ordering. This stack uses normal cryptography.
(2) **gmp-reliable-total-dummy_crypt** This stack has the same microprotocols as the first stack, but has a dummy version of the cryptographic library. This stack does not provide intrusion tolerance.
(3) **C-Ensemble** This is the original C version of Ensemble, which is tolerant only to crash faults.

We compare the view installation time 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 6(a) shows the comparison for the single crash fault case. Figure 6(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 6(a) and 6(b), we see that the time difference for



(a) Single fault: with and without cryptography (b) Double faults: with and without cryptography

Figure 7. Comparing **gmp-reliable-total** & **gmp-reliable-total-dummy_crypt** stacks for different faults

view installation between **C-Ensemble** and the **gmp-reliable-total** stacks is two orders of magnitude, and becomes higher with increasing group sizes. The causes are the multiple rounds of message exchange and the 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. The 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 7 compares the **gmp-reliable-total** and **gmp-reliable-total-dummy_crypt** stacks, with additional fault types. From Figure 7(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. The reason is that 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 7(a). The increase in the cost with increasing group size is more pronounced for the **gmp-reliable-total** stack than for the **gmp-reliable-total-dummy_crypt** stack. That indicates that the cryptographic costs increase more steadily than communication costs with increase in the group size.

Figure 8 compares the view installation times for different combinations of double faults. Let us examine the three clusters of curves in the figure from bottom to top. The one at the bottom is close to the curve for single faults. The bottom cluster of curves corresponds to the combinations of double faults for which the fault detection does not rely on any timeouts. Their values are higher than those of single faults, because the view installation time includes the time to detect the additional 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

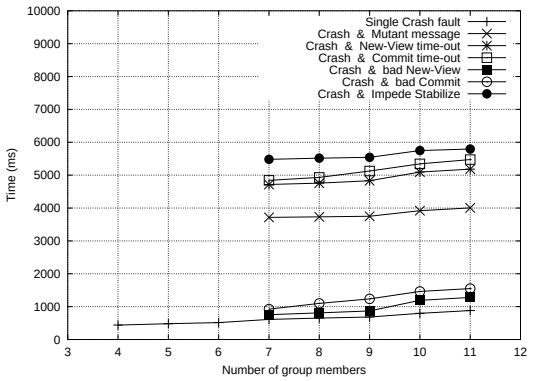


Figure 8. View installation time: single and double faults

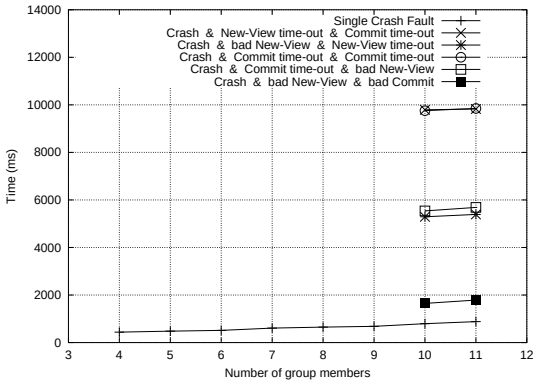


Figure 9. Effect of timeout-based fault detection in transitional views

involve timeouts for fault detection. The timeout 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 timeout for the crash is subsumed in the transitional view installation intended to remove the mutant message fault. The third cluster of curves involved is higher than 4 seconds, because those curves involve a timeout of 4 seconds needed to detect the second fault during the transitional view.

Figure 9 shows the effect on the view installation times of using timeouts for fault detection in the triple faults case. The curves in the top cluster involve two transitional views, both of which use timeouts (of 4 seconds) to detect faults. The curves in the middle cluster also involve two transitional views, but have just one timeout. The curves in the lower cluster do not involve any timeouts. The difference in values between the top and middle cluster, and between the middle and lower cluster, is about 4 seconds (the timeout value), suggesting that if not for the timeouts, the time taken for the other parts of the view installation would be 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 timeouts) 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 group membership protocol can thus be summarized as follows. In a LAN environment, cryptographic operations account for a larger percentage of the cost of tolerating malicious faults than communication. The overhead for tolerating malicious faults is significant compared to the overhead for tolerating just crash faults. Timeout-based detection mechanisms are slower than mechanisms based on direct examination of message content or patterns. In a real-world setting, timeouts 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. Timeouts should be adjusted to take into account the current network load and the load on the other members' host computers.

Discussion: How Intrusion-Tolerant is the ITUA GCS?

The arbitrary fault model is attractive for designing distributed protocols that must operate in adversarial environments, because it abstracts the behavior of the adversary. The model frees the protocol designer from being concerned about what methods an attacker uses to break into a system. Instead, it allows him/her to focus on tolerating the *effects* of successful attacks.

That said, it is highly unlikely that an *implementation* of a distributed protocol designed to operate in the arbitrary fault model can tolerate the effects of all possible attacks. Thus, when an abstracted protocol is translated to code, the system developer does have to concern himself with making the implementation resistant to specific attacks. The requirements of the intended application in the ITUA architecture [26, 33] dictated the specific fault types that the ITUA GCS implementation had to tolerate. The faults we injected to evaluate the group membership protocol are a representative subset of the ITUA fault types.

The fact that a system implementation is focused on tolerating specific classes of attacks implies that there are attack scenarios that the system might not handle gracefully. For example, our GCS implementation uses the standard C library functions and the TCP/IP stack, and does not handle attacks that exploit vulnerabilities at those levels. The implementation also does not handle attacks on the services provided by the underlying operating system. These examples show that to be truly effective against a wide variety of attacks, intrusion tolerance must be provided at all levels of the system: hardware, OS, middleware, and application.

Another issue that any real-world deployment of an intrusion-tolerant GCS must address is the coverage of the assumption that at most f group members can fail. Though it was outside the scope of this paper, work by the authors in the ITUA [33] and DPASA [50] projects has shown that process diversity can help significantly reduce the chance that multiple replicas will be simultaneously corrupted by exploitation of the same vulnerability. The benefits of obtaining such diversity by deploying the processes on heterogeneous platforms and/or diverse administrative domains were quantified in [50]. The issue has also been the focus of other recent work, e.g., the proactive recovery of replicas by Castro and Liskov [11] and the use of diverse replicas based on commodity off-the-shelf (COTS) software in the BASE work of Rodrigues, Castro, and Liskov [12]. It is clear that a real-world deployment of our intrusion-tolerant GCS should incorporate such proactive recovery and diversity techniques.

Conclusion

We demonstrated the feasibility of building an intrusion-tolerant GCS using a crash-tolerant GCS as the starting point. The main advantage of such an approach is the possibility of reusing a large portion of the original code base, potentially resulting in substantial savings in the development effort. The main limitation is that identifying and patching of vulnerabilities can only be best-effort. We designed group communication protocols in the arbitrary fault model and then implemented our abstract protocols in C-Ensemble. We described the modifications we made to C-Ensemble to transform it into the ITUA GCS. We believe that the transformation

procedure can be reproduced in other crash-tolerant GCSs that have modular designs like C-Ensemble. Finally, we experimentally evaluated the performance of the group communication protocols and quantified the cost of tolerating intrusions.

ACKNOWLEDGEMENTS

We would like to thank all members of the ITUA project for many useful discussions. We are grateful to Jenny Applequist for helping us to improve the readability of the paper.

REFERENCES

1. Amir Y, Stanton J. The Spread wide area group communication system. Johns Hopkins University, Technical Report CNDS 98-4, 1998.
2. Amir Y, Dolev D, Kramer S, Malkhi D. Transis: a communication sub-system for high availability. *Proc. 22nd Annual International Symposium on Fault-Tolerant Computing*, 1992; 76–84.
3. Amir Y, Kim Y, Nita-Rotaru C, Tsudik G. On the performance of group key agreement protocols. *ACM Transactions on Information and System Security* 2004; **7**(3):457–488.
4. Baldoni R, Helary J-M, Raynal M. From crash fault-tolerance to arbitrary-fault tolerance: towards a modular approach. *Proc. International Conference on Dependable Systems and Networks (DSN-2000)*, 2000; 273–282.
5. Birman K, Cooper R. The ISIS project: real experience with a fault tolerant programming system. *ACM Operating Systems Review, SIGOPS 25*, 1991; 103–107.
6. Birman K, van Renesse R. *Reliable distributed computing with the Isis toolkit*. IEEE Computer Society Press, 1994.
7. Birman KP. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, 1996.
8. Birman K, Constable B, Hayden M, Hickey J, Kreitz C, van Renesse R, Rodeh O, Vogels W. The Horus and Ensemble projects: accomplishments and limitations. *Proc. DARPA Information Survivability Conference and Exposition (DISCEX-2000)*, 2000; 149–161.
9. Cachin C, Poritz JA. Secure intrusion-tolerant replication on the Internet. *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, 2002; 167–176.
10. Cachin C, Samar A. Secure Distributed DNS. *Proc. International Conference on Dependable Systems and Networks (DSN-2004)*, 2004; 423–432.
11. Castro M, Liskov B. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems* 2002; **20**(4):398–461.
12. Castro M, Rodrigues R, Liskov B. BASE: using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems* 2003; **21**(3):236–269.
13. Chandra TD, Toueg S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 1996; **43**(2):225–267.
14. Chockler GV, Keidar I, Vitenberg R. Group communication specifications: a comprehensive study. *ACM Computing Surveys* 2001; **33**(4):1–43.
15. Correia M. Intrusion tolerance based on architectural hybridization. Ph.D. dissertation, University of Lisbon, 2003.

16. Correia M, Neves NF, Lung LC, Veríssimo P. Worm-IT: A wormhole-based intrusion-tolerant group communication system. *Journal of Systems and Software* 2006; In Press.
17. Cristian F, Fetzer C. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems* 1999; **10**(6):642–657.
18. Cryptlib toolkit page. http://www.cs.auckland.ac.nz/~pgut001/cryptlib/ [25 July 2006].
19. Drabkin V, Friedman R, Kama A. Practical Byzantine group communication. *Proc. 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, 2006; 36.
20. Dutertre B, Crettaz V, Stavridou V. Intrusion-tolerant enclaves. *Proc. 2002 IEEE International Symposium on Security and Privacy*, 2002; 216–226.
21. Ezhilchelvan PD, Macedo RA, Shrivastava SK. Newtop: a fault-tolerant group communication protocol. *Proc. International Conference on Distributed Computing Systems (ICDCS-1995)*, 1995; 296–306.
22. Fischer MJ, Lynch NA, Paterson MS. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 1985; **32**(2):372–382.
23. Friedman R, van Renesse, R. Strong and weak virtual synchrony in Horus. *Proc. 15th Symposium on Reliable Distributed Systems*, 1996; 140–149.
24. Hayden M. The Ensemble system. Ph.D. dissertation, Cornell University, 1998.
25. Hayden M. *Ensemble Reference Manual*, Cornell University, 2001.
26. The ITUA project page. http://itua.bbn.com [25 July 2006].
27. Kihlstrom KP, Moser LE, Melliar-Smith PM. Byzantine fault detectors for solving consensus. *The Computer Journal* 2003; **46**(1):16–35.
28. Kihlstrom KP, Moser LE, Melliar-Smith PM. The SecureRing group communication system. *ACM Transactions on Information and System Security* 2001; **4**(4):371–406.
29. Lamport L, Shostak R, Pease M. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 1982; **4**(3):382–401.
30. Malkhi D, Reiter M. Unreliable intrusion detection in distributed computations. *Proc. 10th Computer Security Foundations Workshop (CSFW97)*, 1997; 116–124.
31. Moser LE, Melliar-Smith PM, Agarwal DA, Budhia RK, Lingley-Papadopoulos CA. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM* 1996; **39**(4):54–63.
32. Mpoeleng D, Ezhilchelvan P, Speirs N. From crash tolerance to authenticated Byzantine tolerance: a structured approach, the cost and benefits. *Proc. International Conference on Dependable Systems and Networks*, 2003; 227–236.
33. Pal P, Rubel P, Atighetchi M, Webber F, Sanders WH, Seri M, Ramasamy H, Lyons J, Courtney T, Agbaria A, Cukier M, Gossett J, Keidar I. An architecture for adaptive intrusion-tolerant applications. *Software—Practice and Experience, Special Issue on Auto-Adaptive and Reconfigurable Systems* 2006; **36**(12):1331–1354.
34. Pandey P. Reliable delivery and ordering mechanisms for an intrusion-tolerant group communication system. Master's thesis, University of Illinois at Urbana-Champaign, 2001.
35. Ramasamy HV, Pandey P, Lyons J, Cukier M, Sanders WH. Quantifying the cost of providing intrusion tolerance in group communication systems. *Proc. International Conference on Dependable Systems and Networks*, 2002; 229–238.
36. Ramasamy HV. Group membership protocol for an intrusion-tolerant group communication system. Master's thesis, University of Illinois at Urbana-Champaign, 2002.
37. Ramasamy HV, Cukier M, Sanders WH. Formal verification of an intrusion-tolerant group membership protocol. *IEICE Transactions on Information and Systems* 2003; **E86-D**(12):2612–2622.

38. Ramasamy HV, Agbaria A, Sanders WH. CoBFIT: a component-based framework for intrusion tolerance. *Proc. 30th Euromicro Conference (EUROMICRO-2004)*, 2004; 591-600.
39. Ramasamy HV, Parsimonious service replication for tolerating malicious attacks in asynchronous environments. PhD thesis, University of Illinois at Urbana-Champaign, 2005.
40. Reiter MK. Secure agreement protocols: reliable and atomic group multicast in Rampart. *Proc. 2nd ACM Conference on Computer and Communication Security*, 1994; 68–80.
41. Reiter M. The Rampart toolkit for building high-integrity services. *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, 1995; Springer-Verlag, 99–110.
42. Reiter MK. A secure group membership protocol. *IEEE Transactions on Software Engineering* 1996; **22**(1):31–42.
43. Reiter MK, Franklin MK. The design and implementation of a secure auction service. *IEEE Transactions on Software Engineering* 1996; **22**(5):302–312.
44. Reiter MK, Franklin MK, Lacy JB, Wright RN. The Omega key management service. *Journal of Computer Security* 1996; **4**(4):267–287.
45. Rodeh O, Birman K, Dolev D. The architecture and performance of security protocols in the Ensemble group communication system. *ACM Transactions on Information and System Security* 2001; **4**(3):289-319.
46. Sames D, Matt B, Niebuhr B, Tally G, Whitmore B, Bakken DE. Developing a heterogeneous intrusion tolerant CORBA system. *In Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, 2002; 239–248.
47. Schlichting R, Schneider FB. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems* 1983; **1**(3):222–238.
48. Schneider FB. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys* 1990; **22**(4):299–319.
49. Sherman AT, McGrew DA. Key establishment in large dynamic groups using one-way function trees. *IEEE Transactions on Software Engineering* 2003, **29**(5):444–458.
50. Stevens F, Courtney T, Singh S, Agbaria A, Meyer JF, Sanders WH, Pal PP. Model-based validation of an intrusion-tolerant information system. *Proc. 23rd Symposium on Reliable Distributed Systems (SRDS-2004)*, 2004; 184–194.
51. Veríssimo P, Neves NF, Correia M. The middleware architecture of MAFTIA: a blueprint. *In Proc. IEEE Third Information Survivability Workshop (ISW-2000)*, 2000.
52. Wallner D, Harder E, Agee R. Key management for multicast: issues and architectures. *RFC 2627*, 1999.
53. Wong CK, Gouda MG, Lam SS. Secure group communications using key graphs. *Transactions on Networking* 2000; **8**(1):16-30.