RITAS: Services for Randomized Intrusion Tolerance

Henrique Moniz, *Student Member*, *IEEE*, Nuno Ferreira Neves, *Member*, *IEEE*, Miguel Correia, *Member*, *IEEE*, and Paulo Verissimo, *Fellow*, *IEEE*

Abstract—Randomized agreement protocols have been around for more than two decades. Often assumed to be inefficient due to their high expected communication and computation complexities, they have remained overlooked by the community-at-large as a valid solution for the deployment of fault-tolerant distributed systems. This paper aims to demonstrate that randomization can be a very competitive approach even in hostile environments where arbitrary faults can occur. A stack of randomized intrusion-tolerant protocols is described and its performance evaluated under several settings in both local-area-network (LAN) and wide-area-network environments. The stack provides a set of relevant services ranging from basic communication primitives up to atomic broadcast. The experimental evaluation shows that the protocols are efficient, especially in LAN environments where no performance reduction is observed under certain Byzantine faults.

Index Terms—Intrusion tolerance, Byzantine agreement, randomized protocols, performance evaluation.

1 Introduction

Modern society has been growing increasingly dependent on networked computer systems. The availability, confidentiality, and integrity of data and services are crucial attributes that must be enforced by real-world distributed systems. The typical approach to secure these systems has been one of almost complete prevention, i.e., to avoid successful attacks, or intrusions, at all cost. Once a breach occurs, manual intervention is necessary to restore system correctness.

A different approach to deal with attacks has been gaining momentum within the scientific community—*intrusion tolerance*. Arising from the intersection of two classical areas of computer science, *fault tolerance* and *security*, its objective is to guarantee the correct behavior of a system even if some of its components are compromised and controlled by an intelligent adversary [1], [2], [3].

Within this domain of fault- and intrusion-tolerant distributed systems, there is an essential problem: *consensus*. This problem has been specified in different ways, but basically it aims to ensure that n processes are able to propose some values and then all agree on one of these values. Consensus has been shown to be equivalent to fundamental problems, such as state machine replication [4], group membership [5], and atomic broadcast [6], [7]. Hence, the relevance of consensus is noteworthy because it is a building block of several important distributed systems services. For example, to maintain data consistency in a replicated database, some form of consensus between the sites is needed. Synchronization of clocks, leader election, or practically any kind of coordinated activity between the various nodes of a distributed system can be built using consensus. Unsurprisingly, the consensus problem has received a lot of attention from the research community.

Consensus, however, is impossible to solve deterministically in asynchronous systems (i.e., systems where there are no bounds to the communication delays and computation times) if a single process can crash (also known as the FLP result [8]). This is a significant result, in particular for intrusion-tolerant systems, because they usually assume an asynchronous model in order to avoid time dependencies. Time assumptions can often be broken, for example, with denial-of-service attacks.

Throughout the years, several researchers have investigated techniques to circumvent the FLP result. Most of these solutions, however, required changes to the basic system model, with the explicit inclusion of stronger time assumptions (e.g., partial synchrony models [9], [10]), or by augmenting the system with devices that hide in their implementation of these assumptions (e.g., failure detectors [11], [12], [13] or wormholes [14]). *Randomization* is another technique that has been around for more than two decades [15], [16]. One important advantage of this technique is that no additional timing assumptions are needed.

To circumvent the FLP result, randomization uses a probabilistic approach where the termination of consensus is ensured with probability of 1. Although this line of research produced a number of important theoretical results, including several algorithms, randomization has been historically overlooked, in what pertains to the implementation of practical applications, because it has usually been considered to be too inefficient.

- *H. Moniz is with the Departamento de Informática, Faculdade de Ciencias, Universidade de Lisboa, Bloco C6.3.30, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: hmoniz@di.fc.ul.pt.*
- *N.F. Neves and M. Correia are with the Departamento de Informática, Faculdade de Cincias, Universidade de Lisboa, Bloco C6-Piso 3, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: {nuno, mpc}@di.fc.ul.pt.*
- *P. Verissimo is with the Departamento de Informática, Faculdade de Cincias, Universidade de Lisboa, Bloco C6.3.10, Campo Grande, 1700-016 Lisboa, Portugal. E-mail: pjv@di.fc.ul.pt.*

Manuscript received 28 June 2007; revised 7 Jan. 2008; accepted 10 Nov. 2008; published online 2 Dec. 2008.
Recommended for acceptance by L. Alvisi.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2007-06-0084.
Digital Object Identifier no. 10.1109/TDSC.2008.76.

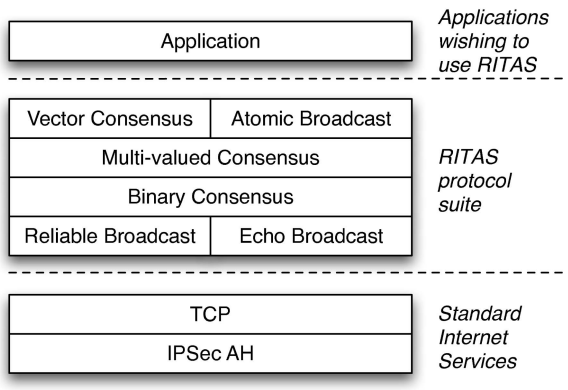

Fig. 1. The RITAS protocol stack.

The reasons for the assertion that *"randomization is inefficient in practice"* are simple to summarize. Randomized consensus algorithms, which are the most common form of these algorithms, usually have a large expected number of communication steps, i.e., a large time complexity. Even when this complexity is constant, the expected number of communication steps is traditionally significant even for small numbers of processes, when compared, for instance, with solutions based on failure detectors.[1] Many of those algorithms also rely on public-key cryptography, which increases the performance costs, especially for local area networks (LANs) or MANs in which the time to compute a digital signature is usually much higher than the network delay.

Nevertheless, two important points have been chronically ignored. First, consensus algorithms are not usually executed in oblivion, they are run in the context of a higher-level problem (e.g., atomic broadcast) that can provide a friendly environment for the "lucky" event needed for faster termination (e.g., many processes proposing the same value can lead to a quicker conclusion [20]). Second, for the sake of theoretical interest, the proposed adversary models usually assume a strong adversary that completely controls the scheduling of the network and decides which processes receive which messages and in what order. In practice, a real adversary does not possess this ability, but if it does, it will probably perform attacks in a distinct (and much simpler) manner to prevent the conclusion of the algorithm—for example, it can block the communication entirely. Therefore, in practice, the network scheduling can be "nice" and lead to a speedy termination.

This paper describes the implementation of a stack of randomized intrusion-tolerant protocols and evaluates their performance under different fault loads. One of the main purposes is to show that randomization can be efficient and should be regarded as a valid solution for practical intrusion-tolerant distributed systems.

This implementation is called RITAS which stands for *Randomized Intrusion-Tolerant Asynchronous Services*. At the lowest level of the stack (see Fig. 1), there are two broadcast primitives: *reliable broadcast* and *echo broadcast*. On top of these primitives, the most basic form of consensus is available, *binary consensus*. This protocol lets processes decide on a single bit and is, in fact, the only randomized algorithm of the stack. The rest of the protocols are built on top of this one. Building on the *binary consensus* layer, *multivalued consensus* allows the agreement on values of arbitrary range. At the highest level, there is *vector consensus*, which lets processes decide on a vector with values proposed by a subset of the processes, and *atomic broadcast*, which ensures total order. The protocol stack is executed over a reliable channel abstraction provided by standard Internet protocols—TCP ensures reliability, and IPSec guarantees cryptographic message integrity [21]. The protocols in RITAS have been previously described in the literature [22], [23], [7]. The implemented protocols are, in most cases, optimized versions of the original proposals that have significantly improved the overall performance.

The protocols of RITAS share a set of important structural properties:

- They are asynchronous in the sense that no assumptions are made on the processes' relative execution and communication delays, thus preventing attacks against assumptions in the domain of time (a known problem in some protocols that have been presented in the past).
- They attain optimal resilience, tolerating up to $f = \lfloor \frac{n-1}{3} \rfloor$ malicious processes out of a total of n processes, which is important since the cost of each additional replica has a significant impact in a real-world application.
- They are signature-free, meaning that no expensive public-key cryptography is used anywhere in the protocol stack, which is relevant in terms of performance since this type of cryptography is several orders of magnitude slower than symmetric cryptography.
- They take decisions in a distributed way (there is no leader), thus avoiding the costly operation of detecting the failure of a leader, an event that can considerably delay the execution.

This paper has two main contributions: 1) it presents the design and implementation of a stack of randomized intrusion-tolerant protocols, discussing several optimizations—to the best of our knowledge, the implementation of a stack with the four structural properties above is novel and 2) it provides a detailed evaluation of RITAS in both LAN and wide-area-network (WAN) settings, showing that it has interesting latency and throughput values.

2 Related Work

Randomized intrusion-tolerant protocols have been around since Ben-Or's and Rabin's seminal consensus protocols [15], [16]. These papers defined the two approaches that each of the subsequent works followed. Essentially, all randomized protocols rely on a *coin-tossing scheme* that generates random bits. Ben-Or's approach relies on a local coin toss, while in Rabin's shares of the coins are distributed by a trusted dealer before the execution of the protocol and, therefore, all processes see the same coins.

1. An exception is the stack of randomized protocols proposed by Cachin et al. [17], [18], which terminate in a low expected number of communication steps. They, however, depend heavily on public-key cryptography which may seriously affect their performance [19].

Although many randomized asynchronous protocols have been designed throughout the years [15], [16], [22], [24], [25], [26], only recently one implementation of a stack of randomized multicast and agreement protocols has been reported, SINTRA [18]. These protocols are built on top of a binary consensus protocol that follows a Rabin-style approach, and in practice terminates in one or two communication steps [17]. The protocols, however, depend heavily on public-key cryptography primitives like digital and threshold signatures. The implementation of the stack is in Java and uses several threads. RITAS uses a different approach, Ben-Or-style, and resorts only to fast cryptographic operations such as hash functions.

Randomization is only one of the techniques that can be used to circumvent the FLP impossibility result. Other techniques include failure detectors [12], [27], [28], [20], partial-synchrony [9], and distributed wormholes [29], [14]. Some of these techniques have been employed in the past to build other intrusion-tolerant protocol suites.

The first evaluation of a set of asynchronous Byzantine protocols (reliable and atomic broadcast) was made for the Rampart toolkit [23]. The reliable broadcast is implemented by Reiter's echo broadcast (see Section 4.7), and the order is defined by a leader that also echo-broadcasts the order information. Even with such a simple protocol, and using small RSA keys (300 bits), this paper acknowledges that "public-key operations still dominate the latency of reliable multicast, at least for small messages." Moreover, if a process does not echo-broadcast a message to all or if a malicious leader performs some attack against the ordering of the messages, these events have to be detected and the corrupt process removed from the group. This implies that liveness is dependent on the cost of this detection [30] and synchrony assumptions are required about the network delay, allowing attacks where malicious processes delay others in order to force their removal. For this reason, Rampart relies on a group membership protocol not only to handle voluntary joins and leaves from the group but also to detect and remove corrupt processes. This is a necessity (that emerges out of design) as much as it is a feature. RITAS can indeed be extended with a group membership protocol that handles dynamic groups; however, it does not require one for its protocols to make progress because decisions are made in a decentralized way.

Like Rampart, SecureRing is an intrusion-tolerant group communication system [31]. It relies on a token that rotates among the processes to decide the order of message deliveries. This signed token carries message digests, a solution that allows a lower number of signatures and an improvement in performance when compared to Rampart. In SecureRing, malicious behavior also has to be detected for the protocols to make progress, which means that it suffers from similar problems as Rampart.

WORM-IT uses the wormhole abstraction to provide a membership service and a view-synchronous atomic multicast primitive [32]. It is designed under a hybrid system model. The system is considered to be asynchronous and subject to Byzantine failures with the exception of a small subset, the wormhole, that is assumed to be secure (i.e., can only crash) and synchronous. Critical steps of the protocols that require stronger environmental properties (such as agreement tasks) are executed inside the *wormhole*.

Byzantine JazzEnsemble is another group communication system that resists Byzantine failures [33]. It relies on *fuzzy mute* and *fuzzy verbose* failure detectors to detect *mute* failures (i.e., a process neglecting to send messages) and *verbose failures* (i.e., a process sending too many messages), respectively. These kinds of failures can be identified based on locally observed events, which motivates the use of such failure detectors. Moreover, the system provides a vector consensus protocol and a uniform broadcast protocol, as well as modifications at each layer to overcome potential Byzantine attacks.

BFT, while not a protocol stack, is an algorithm that provides a Byzantine-fault-tolerant state machine replication service [34]. In BFT, there are clients and servers. The clients issue requests to the servers, then requests are processed by the servers in total order, and a reply is returned to the clients. Servers are either primary or backup, and there is only one primary at any given moment in the system. Client requests are issued directly to the primary, which in turn multicasts the request to the backups. The replies are transmitted to clients by all servers. A client waits for $f + 1$ replies with the same result in order to obtain the response. This comprises the normal operation of the algorithm. In case a primary fails, a view change must occur and servers must agree on a new primary. View changes are triggered by timeouts. After a view change, the service resumes to its normal operation.

It is hard to compare BFT and RITAS because they are designed with different assumptions and goals in mind. BFT is centralized and requires synchrony for liveness, while RITAS is decentralized and completely asynchronous. BFT is a system designed to perform a very specific task (i.e., state machine replication), while RITAS is a stack that provides several general broadcast and consensus protocols that can be applied to a multitude of scenarios including state machine replication. For instance, the Reliable Broadcast protocol in RITAS could be used as a primitive to implement state machine replication (more specifically, the dissemination of requests from a primary to the backups).

3 System Model

The system is composed by a group of n processes $P = \{p_0, p_1, \ldots p_{n-1}\}$. Group membership is static, i.e., the group is predefined and there cannot be joins or leaves during system operation. Processes are fully connected.

There are no constraints on the kind of faults that can occur in the system. This class of unconstrained faults is usually called *arbitrary* or *Byzantine*. Processes are said to be *correct* if they do not *fail*, i.e., if they follow their protocol until termination. Processes that fail are called *corrupt*. No assumptions are made about the behavior of corrupt processes—they can, for instance, stop executing, omit messages, send invalid messages either alone or in collusion with other corrupt processes. It is assumed that at most $f = \lfloor \frac{n-1}{3} \rfloor$ processes can be corrupt.

The system is completely asynchronous. Therefore, there are no assumptions whatsoever about bounds on processing times or communications delays.

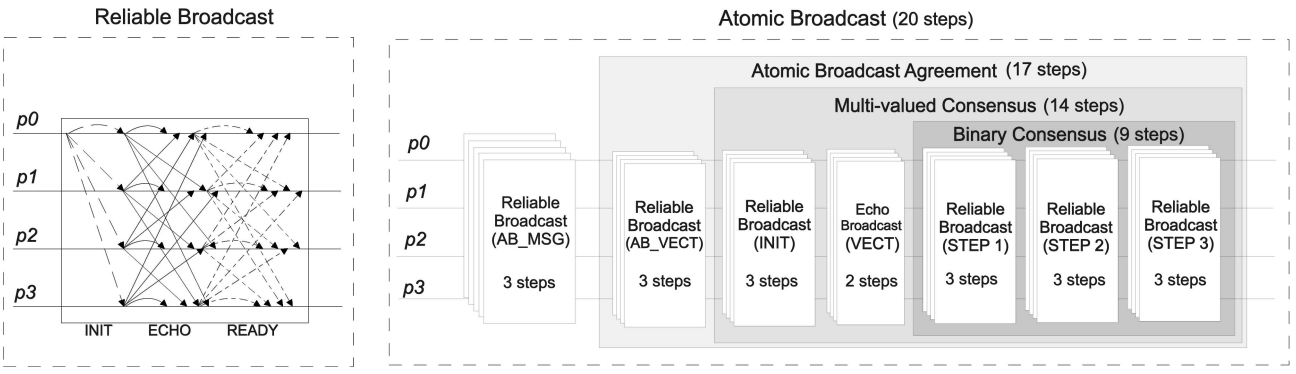

Fig. 2. Overview of the messages exchanged and best-case number of communication steps in each protocol.

Each pair of processes (p_i, p_j) shares a secret key s_{ij}. It is out of the scope of this work to present a solution for distributing these keys, but it may require a trusted dealer or some kind of key distribution protocol based on public-key cryptography. Nevertheless, this is a long-term operation, normally performed before the execution of the protocols and does not interfere with their performance.

Each process has access to a random bit generator that returns unbiased bits observable only by the process (if the process is correct).

Some protocols use a *cryptographic hash function* $H(m)$ that maps an input m of arbitrary length into a fixed-length output. We assume that it is impossible 1) to find two values $m \neq m'$ such that $H(m) = H(m')$ and 2) given a certain output, to find an input that produces that output. The output of the function is often called *a hash*.

4 Protocol Stack

This section briefly describes the function of each protocol and how it works. Since all protocols have already been described in the literature, no formal specifications are given, and some details are only provided to explain the optimizations. We have developed formal proofs showing that the optimized protocols behave according to their specification, but we could not present them in this paper due to lack of space [35].

4.1 Reliable Channel

The two layers at the bottom of the stack implement a reliable channel (see Fig. 1). This abstraction provides a point-to-point communication channel between a pair of correct processes with two properties: reliability and integrity. Reliability means that messages are eventually received, and integrity says that messages are not modified in the channel. In practical terms, these properties can be enforced using standard Internet protocols: reliability is provided by TCP, and integrity by the IPSec Authentication Header (AH) protocol [21].

4.2 Reliable Broadcast

The *reliable broadcast* primitive ensures two properties: 1) all correct processes deliver the same messages and 2) if the sender is correct then the message is delivered. The implemented protocol was originally proposed by Bracha [22]. The protocol starts with the sender broadcasting a message (INIT, m) to all processes. Upon receiving this message, a process sends a (ECHO, m) message to all processes. It then waits for at least $\lfloor \frac{n+f}{2} \rfloor + 1$ (ECHO, m) messages or $f + 1$ (READY, m) messages, and then it transmits a (READY, m) message to all processes. Finally, a process waits for $2f + 1$ (READY, m) messages to deliver m. Fig. 2 illustrates the three communication steps of the protocol.

4.3 Echo Broadcast

The *echo broadcast* primitive is a weaker and more efficient version of the *reliable broadcast*. Its properties are somewhat similar; however, it does not guarantee that all correct processes deliver a broadcast message if the sender is corrupt [24]. In this case, the protocol only ensures that the subset of correct processes that deliver will do it for the same message.

The protocol is essentially the described *reliable broadcast* algorithm with the last communication step omitted. An instance of the protocol is started with the sender broadcasting a message (INITIAL, m) to all processes. When a process receives this message, it broadcasts a (ECHO, m) message to all processes. It then waits for more than $\frac{n+f}{2}$ (ECHO, m) messages to accept and deliver m.

4.4 Binary Consensus

A *binary consensus* allows correct processes to agree on a binary value. The implemented protocol is adapted from a randomized algorithm by Bracha [22]. Each process p_i proposes a value $v_i \in \{0, 1\}$ and then all correct processes decide on the same value $b \in \{0, 1\}$. In addition, if all correct processes propose the same value v, then the decision must be v. The protocol has an expected number of communication steps for a decision of 2^{n-f}, and uses the underlying *reliable broadcast* as the basic communication primitive.

The protocol proceeds in three-step rounds, running as many rounds as necessary for a decision to be reached. In the first step, each process p_i (reliably) broadcasts its proposal v_i, waits for $n - f$ *valid* messages (the definition of *valid* is given in the next paragraph), and changes v_i to reflect the majority of the received values. In the second step, p_i broadcasts v_i, waits for the arrival of $n - f$ *valid* messages, and if more than half of the received values are equal, v_i is set to that value; otherwise, v_i is set to the undefined value $\perp$. Finally, in the third step, p_i broadcasts v_i, waits for $n - f$ *valid* messages, and decides if at least

$2f + 1$ messages have the same value $v \neq \perp$. Otherwise, if at least $f + 1$ messages have the same value $v \neq \perp$, then v_i is set to v and a new round is initiated. If none of the above conditions apply, then v_i is set to a random bit with value 1 or 0, with probability $\frac{1}{2}$, and a new round is initiated.

A message received in the first step of the first round is always considered *valid*. A message received in any other step k, for $k > 1$, is *valid* if its value is congruent with any subset of $n - f$ values accepted at step $k - 1$. For example, suppose that process p_i receives $n - f$ messages at step 1, where the majority has value 1. Then at step 2, it receives a message with value 0 from process p_j. Remember that the message a process p_j broadcasts at step 2 is the majority value of the messages received by it at step 1. That message cannot be considered *valid* by p_i since value 0 could never be derived by a correct process p_j that received the same $n - f$ messages at step 1 as process p_i (i.e., value 0 is not congruent). If process p_j is correct, then p_i will eventually receive the necessary messages for step 1, which will enable it to form a subset of $n - f$ messages that validate the message with value 0. This validation technique has the effect of causing the processes that do not follow the protocol to be ignored.

4.5 Multivalued Consensus

A *multivalued consensus* allows processes to propose a value $v \in \mathcal{V}$ with arbitrary length. The decision is either one of the proposed values or a default value $\perp \notin \mathcal{V}$. The implemented protocol is based on the multivalued consensus proposed by Correia et al. [7]. It uses the services of the underlying *reliable broadcast*, *echo broadcast*, and *binary consensus* layers. The main differences from the original protocol are the use of echo broadcast instead of reliable broadcast at a specific point, and a simplification of the validation of the vectors used to justify the proposed values.

The protocol starts when every process p_i announces its proposal value v_i by reliably broadcasting a (INIT, v_i) message. The processes then wait for the reception of $n - f$ INIT messages and store the received values in a vector V_i. If a process receives at least $n - 2f$ messages with the same value v, it echo-broadcasts a (VECT, v, V_i) message containing this value together with the vector V_i that justifies the value. Otherwise, it echo-broadcasts the default value $\perp$ that does not require justification. The next step is to wait for the reception of $n - f$ *valid* VECT messages. A VECT message, received from process p_j, and containing vector V_j, is considered *valid* if one of two conditions hold: 1) $v = \perp$ and 2) there are at least $n - 2f$ elements $V_i[k] \in \mathcal{V}$ such that $V_i[k] = V_j[k] = v_j$. If a process does not receive two *valid* VECT messages with different values, and it received at least $n - 2f$ *valid* VECT messages with the same value, it proposes 1 for an execution of the *binary consensus*; otherwise, it proposes 0. If the binary consensus returns 0, the process decides on the default value $\perp$. If the binary consensus returns 1, the process waits until it receives $n - 2f$ *valid* VECT messages (if it has not done so already) with the same value v and decides on that value.

4.6 Vector Consensus

Vector consensus allows processes to agree on a vector with a subset of the proposed values. The protocol is the one described in [7] and uses *reliable broadcast* and *multivalued consensus* as underlying primitives. It ensures that every correct process decides on a same vector V of size n; if a process p_i is correct, then $V[i]$ is either the valued proposed by p_i or the default value $\perp$, and at least $f + 1$ elements of V were proposed by correct processes.

The protocol starts by reliably broadcasting a message containing the proposed value by the process and setting the round number r_i to 0. The protocol then proceeds in up to f rounds until a decision is reached. Each round is carried out as follows. A process waits until $n - f + r_i$ messages have been received and constructs a vector W_i of size n with the received values. The indexes of the vector for which a message has not been received have the value $\perp$. The vector W_i is proposed as input for the *multivalued consensus*. If it decides on a value $V_i \neq \perp$, then the process decides V_i. Otherwise, the round number r_i is incremented and a new round is initiated.

4.7 Atomic Broadcast

An *atomic broadcast* protocol delivers messages in the same order to all processes. One can see atomic broadcast as a reliable broadcast plus the total order property. The implemented protocol was adapted from [7]. The main difference is that it has been changed to use multivalued consensus instead of vector consensus and to utilize message identifiers for the agreement task instead of cryptographic hashes. These changes were made for efficiency and have been proved not to compromise the correctness of the protocol. The protocol uses *reliable broadcast* and *multivalued consensus* as primitives.

The atomic broadcast protocol is divided in two tasks (see Fig. 2): 1) the broadcasting of messages and 2) the agreement over which messages should be delivered. When a process p_i wishes to broadcast a message m, it simply uses the reliable broadcast to send a (AB_MSG, i, *rbid*, m) message where *rbid* is a local identifier for the message. Every message in the system can be uniquely identified by the tuple (i, *rbid*).

The agreement task 2 is performed in rounds. A process p_i starts by waiting for AB_MSG messages to arrive. When such a message arrives, p_i constructs a vector V_i with the identifiers of the received AB_MSG messages and reliably broadcasts a (AB_VECT, i, r, V_i) message, where r is the round for which the message is to be processed. It then waits for $n - f$ AB_VECT messages (and the corresponding V_j vectors) to be delivered and constructs a new vector W_i with the identifiers that appear in $f + 1$ or more V_j vectors. The vector W_i is then proposed as input to the *multivalued consensus* protocol and if the decided value W' is not $\perp$, then the messages with their identifiers in the vector W' can be deterministically delivered by the process.

5 Implementation

This section describes the internal structure of the protocol stack and provides an insight into the design considerations and practical issues that arose during the development of RITAS. The protocol stack was implemented in the C language and was packaged as a shared library with the goal of offering a simple interface to applications

wishing to use the protocols. Some of the concepts presented here have been studied in other group communication systems such as Horus and Ensemble [36], [37].

5.1 Single-Threaded Operation

When developing a software component such as a protocol stack, there are two possible options regarding its operation: multithreaded or single threaded. The RITAS protocol stack runs in a single thread, independent of the application threads.

In a typical multithreaded protocol stack, every instance of a specific protocol is handled by a separate thread. Usually, there is a pivotal thread that reads messages from the network and instantiates protocol threads to handle messages that are specific to them. Another option is to avoid the pivotal thread, and have the protocol threads reading messages directly from the network.

The multithreaded approach may be simpler to implement since each context is self-contained in a given thread, and there is virtually no need for protocol demultiplexing since messages can be addressed directly to the threads handling them. This leads to a cleaner implementation (i.e., more verbatim translations from pseudocode) because the protocol code has only to deal with one protocol instance (the context is implicit). Nevertheless, in a loaded system, with potentially several hundreds of threads, the constant context switching and synchronization between threads poses a serious performance impact on the stack, and may provoke an unfair internal scheduling.

A single-threaded approach, while more complex to develop, allows a much more efficient stack operation when properly implemented. A single-threaded protocol stack ensures a fair first-come, first-served scheduling as messages are processed by the relevant protocol instances one-by-one as they are received. But this approach poses additional challenges. The contexts for the different protocol instances are not self-contained and require explicit management, which adds complexity to such tasks as message passing, protocol demultiplexing, and packet construction. The specific protocol code also becomes harder to implement since it has to juggle between multiple contexts.

Since one of the main goals of RITAS was the implementation of an efficient protocol stack, the extra complexity of a single-threaded approach was outweighed by its potential performance advantages.

5.2 Message Buffers

In a multilayered network protocol stack, messages have to be passed back and forth. A certain degree of flexibility is needed to manipulate the buffers that hold the messages because data may need to be prepended or appended to these buffers, and existing data may need to be transformed or deleted. Additionally, the number of operations that actually copy data has to be kept to a minimum to reduce performance penalties.

In RITAS, information is passed along the protocol stack using *message buffers* (*mbuf* for short). A *mbuf* is used to store a message and several metadata related to its management. All communication between the different layers is done by passing pointers to *mbufs*. This way, it is possible to both eliminate the need to copy large chunks of data when passing messages from one layer to another, and have a data structure that facilitates the manipulation of messages. This data structure was inspired by the TCP/IP implementation in the Net/3 Operating System kernel [38].

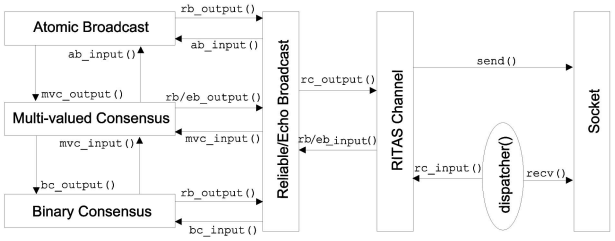

Fig. 3. Communication flow among the protocol layers during an atomic broadcast.

A *mbuf* is usually created when a new message arrives from the network. The RITAS network scheduler creates a *mbuf*, then it reads the message from the socket directly into the *mbuf*, and passes the *mbuf* to the appropriate protocol layer. A *mbuf* can also be created by a specific protocol layer, for instance if it needs to send a message to other processes. Every *mbuf* is reutilized as much as possible.

There are also specific rules as to when a *mbuf* should be destroyed. An outbound *mbuf* should be destroyed immediately after its message is sent to all relevant processes. The exception is when a `RITAS_MBUF_PROTECTED` flag is set. In this case, the *mbuf* was explicitly marked for no destruction by a particular protocol layer, which then becomes solely responsible for the *mbuf* destruction. For an inbound *mbuf*, the last protocol to which the *mbuf* is going to be passed is responsible for its management. A protocol layer has three options, which are mutually exclusive, after it has processed the message contained in the *mbuf*: it passes the *mbuf* to an upper layer protocol, it destroys the *mbuf*, or it reuses the *mbuf* to transmit a new message. The chosen action depends on the semantic of the protocol and the current state of the particular protocol instance context to which the *mbuf* is relevant.

5.3 Control Blocks and Protocol Handlers

Each protocol implemented in RITAS is formed by two protocol-specific components: the *control block* and the *protocol handler*. The control block is a data structure that holds the state of a specific instance of the protocol. It keeps track of things like the instance identification, the current protocol step, and the values received so far.

The protocol handler is the set of functions that implements the operation of the protocol. It is formed by initialization and destruction functions, input and output functions, and one or more functions that export the protocol functionality. The purpose of the initialization and destruction functions is, respectively, to allocate a new control block and initialize all its variables and data structures, and to destroy the internal data structures and the control block itself. The input and output functions are used for inter-protocol communication, and both receive as parameters the respective control block and the *mbuf* to be processed. The communication between the protocols is depicted in Fig. 3.

6 The RITAS Channel and Control Block Chaining

Since applications might perform several broadcast and/or agreement operations simultaneously, the ability to execute multiple instances of the same protocol is a requisite. Therefore, one needs to support many contexts for the different protocol instances. When a message is passed to a given protocol layer, that layer must be able to identify the relevant context for the message, and process the message according to it. This hints a necessity of having each protocol instance uniquely identified, and to have messages addressed to specific protocol instances to avoid overlapping of multiple instances. Two techniques in RITAS make possible the efficient implementation of this functionality: the RITAS Channel and Control Block Chaining.

6.1.1 RITAS Channel

This is a special protocol handler that sits between the broadcast layers and the Reliable Channel layer (the Reliable Channel layer corresponds to the implementation of TCP and IPSec that is accessed through the socket interface) (see Fig. 3). It is the first layer to process messages after they are read from the network, and the last one before they are written to the network.

The purpose of the RITAS channel is to build a header containing a unique identifier for each message. Messages are always addressed to a given RITAS Channel. The message is then passed along the appropriate protocol instances by a mechanism called control block chaining.

6.1.2 Control Block Chaining

This mechanism manages the linking of different protocol instances, solving several problems: it gives a means to unambiguously identify all messages, provides for seamless protocol demultiplexing, and facilitates control block management.

Control block chaining works in the following way. Suppose an application executes an atomic broadcast. The creation of the atomic broadcast protocol instance is done by calling the corresponding initialization function that returns a pointer to a control block responsible for that instance. Since atomic broadcast uses multivalued consensus and reliable broadcast as primitives, the atomic broadcast initialization function also calls the initialization functions for such protocols in order to create as many instances of these protocols as needed. The returned control blocks are kept and managed in the atomic broadcast control block. This mechanism is recursive since second-order protocol instances may need to use other protocols as primitives and so on. The result is a tree of control blocks that has its root at the protocol called by the application and goes down all the way, having control blocks for RITAS Channels as the leaf nodes.

A unique identifier is given to each outbound message when the associated *mbuf* reaches the RITAS Channel layer. The tree is traversed bottom-up starting at the RITAS Channel control block and ending at the root control block. The message identifier is generated by appending the protocol instance ID of each traversed node to a local message identifier that was set by the node that created the *mbuf*.

Protocol demultiplexing is done seamlessly. When a message arrives, its identification defines an association with a particular RITAS Channel control block. The RITAS Channel passes the *mbuf* to the upper layer by calling the appropriate input function of its parent control block. The message is processed by that layer and the *mbuf* keeps being passed in the same fashion.

6.2 Out-of-Content Messages

The asynchronous nature of the protocol stack allows scenarios where a process receives messages for a protocol instance whose context has not yet been created. These messages—called *out-of-context* (OOC) messages—have no context to handle them, though they will, eventually.

Since the correctness of the protocols depends on the eventual delivery of these messages, they cannot simply be discarded. All OOC messages are stored in a hash table. When a RITAS Channel is created, it checks this hash table for relevant messages. If any relevant messages exist, they are promptly delivered to the upper protocol instance.

It is also possible for a protocol instance to be destroyed before consuming all of its OOC messages. To scenarios where OOC messages are kept indefinitely in the hash table, upon the destruction of a protocol, the hash table is checked and all relevant messages are deleted. This is not a solution for the case where a malicious process sends bogus OOC messages that will never have a context. The problem of finite memory in Byzantine message-passing systems is an open issue in the research community. In principle, RITAS and other group communication systems could benefit from an approach such as the one in [39].

7 Performance Evaluation

This section describes the performance evaluation of the protocol stack in both LAN and WAN environments. Two different performance analyses are made. First, a comparative evaluation is presented in order to gain insight on the stack, and on how protocols relate and build on one another performance-wise. Second, an in-depth analysis is conducted on how atomic broadcast performs under various conditions. This protocol is arguably the most interesting candidate for a detailed study because it utilizes all other protocols as primitives, either directly or indirectly, and it can be used for many practical applications [34], [32], [40].

7.1 Testbeds

The experiments were carried out on three different testbeds. Two represent LAN environments which differ on the hardware and the number of nodes they accommodate, and the third represents a WAN environment with four nodes.

The first LAN testbed, which will be referred as *tb-lan-slow*, consisted of four 500-MHz Dell Pentium III PCs with 128 Mbytes of RAM, running Linux kernel 2.6.5. The PCs were connected by a 100-Mbps HP ProCurve 2424M network switch. Bandwidth tests taken with the network performance tool *lperf* have shown a consistent throughput of 9.1 Mbytes/s in full-duplex mode.

The second LAN testbed, which will be referred as *tb-lan-fast*, consisted of 10 Dell PowerEdge 850 servers.

TABLE 1
Average Round-Trip Latency and Bandwidth between Every Pair of Nodes in Testbed *tb-wan* (Variance is Shown in Parentheses)

	Latency (ms)	Bandwidth (Kb/s)
Berkeley - Ishikawa	131 (0.26)	1894
Lisbon - Berkeley	210 (1.12)	1167
Berkeley - Campinas	243 (1.37)	990
Lisbon - Campinas	281 (1.24)	845
Lisbon - Ishikawa	322 (1.78)	740
Campinas - Ishikawa	472 (0.85)	165

TABLE 2
Average Latency for Isolated Executions of Each Protocol in Testbed *tb-lan-slow* (100 Mbps) with Four Processes

	w/ IPSec (μs)	w/o IPSec (μs)	Overhead
Echo Broadcast	1724	1497	15%
Reliable Broadcast	2134	1641	30%
Binary Consensus	8922	6816	30%
Multi-valued Cons.	16359	11186	46%
Vector Consensus	20673	15382	34%
Atomic Broadcast	23744	18604	27%

These servers have Pentium 4 CPUs with 2.8 GHz of clock speed, and 2 Gbytes of RAM. They were connected by a Dell PowerConnect 2724 network switch with 10/100/1,000 Mbps of bandwidth capacity. The operating system was Linux 2.6.11. Bandwidth tests showed a consistent throughput of 1.16 Mbytes/s for the 10-Mbps setting, 11.5 Mbytes/s for the 100 Mbps setting, and 67.88 Mbytes/s for the 1,000-Mbps setting. All values were taken in full-duplex mode, which was used in the experiments.

The WAN testbed, which will be referred as *tb-wan*, consisted of four nodes, each one located in a different continent: a European node in Lisbon, Portugal (P4, 3 GHz, 1-Gbyte RAM), a North American node in Berkeley, CA (P4, 2.4 GHz, 1-Gbyte RAM), a South American node in Campinas, Brazil (Xeon, 3 GHz, 1.5-Gbyte RAM), and an Asian node in Ishikawa, Japan (P4, 3.4 GHz, 3.5-Gbyte RAM). These nodes belong to the Planetlab platform [41] and their operating system was Linux 2.6.12. Table 1 shows the round-trip latency and bandwidth measurements taken between each pair of nodes.

For testbeds *tb-lan-fast* and *tb-lan-slow*, the IPSec implementation that was used was the one available in the Linux kernel and the reliable channels that were established between every pair of processes employed the IPSec AH protocol (with SHA-1) in transport mode [21]. For testbed *tb-wan*, there was no IPSec available, so the experiments for this testbed were carried out with regular IP. This makes the protocols insecure since the integrity property of the channels is not provided, but our interest here is to evaluate the performance of the protocols. In practice, this is not affected because the latency added by the cryptographic operations (at the microseconds order) is negligible compared to the latency of the WAN links (around hundreds of milliseconds).

7.2 Stack Analysis

In order to get a better understanding about the relative overheads of each layer of the stack, we have run a set of experiments to determine the latencies of the protocols. These measurements were carried out in the following manner: a signaling machine that does not participate in the protocols is selected to control the benchmark execution. It starts by sending a 1-byte UDP message to the n processes to indicate which specific protocol instance they should create. Then, it transmits M messages, each one separated by a 2-seconds interval (in our case, M was set to 100). Whenever one of these messages arrives, a process runs the protocol, either a broadcast or a consensus. In case of a broadcast, the process with the lowest identifier acts as the sender, while the others act as receivers. In case of a consensus, all processes propose identical initial values.[2] The broadcast messages and the consensus proposals all carry a 10-byte payload (except for binary consensus where the payload is 1 byte). The latency of each instance was obtained at a specific process. This process records the instant when the signal message arrives and the time when it either delivers a message (for broadcast protocols) or a decision (for consensus protocols). The measured latency is the interval between these two instants. The *average latency* is obtained by taking the mean value of the sample of measured values. Outliers were identified and excluded from the sample.

The results for testbed *tb-lan-slow* with four processes, shown in Table 2, demonstrate the interdependencies among protocols and how much time is spent on each protocol. For example, in a single atomic broadcast instance, roughly 2/3 of the time is taken running a multivalued consensus. For a multivalued consensus, about 1/2 of the time is used by the binary consensus. And for vector consensus, about 3/4 of the time is utilized by the multivalued consensus. The experiments also demonstrated that consensus protocols were always able to reach a decision in one round because the initial proposals were identical.

The table also shows the cost of using IPSec. This overhead could in part be attributed to the cryptographic calculations, but most of it is due to the increase on the size of the messages. For example, the total size of any Reliable Broadcast message—including the Ethernet, IP, and TCP headers—carrying a 10-byte payload is 80 bytes. The IPSec AH adds another 24 bytes, which accounts for an extra 30 percent.

Table 3 shows the performance results for testbed *tb-lan-fast*. The average latency for all protocols is presented for three different group sizes: 4, 7, and 10 processes. The relative slowdown with respect to the four-process scenario is also shown for each protocol.

The first conclusion that can be extracted from these results is that protocols in this testbed exhibit a much better performance than the previous testbed. The use of more powerful hardware had a significant impact on the

2. The only protocol whose performance may directly suffer from different initial values is binary consensus since its termination is probabilistic. This protocol has been subject to a thorough evaluation in a different paper and its performance has been shown not to be significantly affected by different initial values [19].

TABLE 3
Average Latency and Relative Slowdown (w.r.t. to the Four-Process Scenario) for Isolated Executions of Each Protocol (with IPSec) in Testbed *tb-lan-fast* (1,000 Mbps)

	n	w/ IPSec (μs)	relative slowdown
Echo Broadcast	4	584	-
	7	805	38%
	10	1045	79%
Reliable Broadcast	4	667	-
	7	907	36%
	10	1172	76%
Binary Consensus	4	3094	-
	7	8991	190%
	10	19741	538%
Multi-valued Consensus	4	4952	-
	7	13335	169%
	10	25652	418%
Vector Consensus	4	6022	-
	7	16826	179%
	10	32674	443%
Atomic Broadcast	4	6467	-
	7	18496	186%
	10	33474	418%

performance of all protocols. For instance, in the case of binary consensus, the performance was improved threefold, while for atomic broadcast, performance was increased almost four times. The network switch with increased bandwidth capacity, the network interface cards with better performance, and the machines in general with greater computational power are the obvious candidates to justify the performance gain. It is unclear, however, in the relative weight of the various hardware components on the faster protocol execution. Later experiments isolate some of these parameters and demonstrate in greater depth the impact of network bandwidth and host computational power on the protocol stack performance.

Another interesting observation from the results in Table 3 is the relative slowdown of each protocol when the group size increases. The reliable and echo broadcast protocols were less sensitive to a larger group size, while the slowdown for the remaining protocols was considerably accentuated due to the increase in the number of exchanged messages. Reliable and echo broadcast exchange $O(n^2)$ messages per communication step, while the remaining protocols exchange $O(n^3)$, thus being more sensitive to increasing group sizes.

The results for the WAN environment are shown in Table 4. The performance of the protocols is significantly affected by the higher-latency, lower-bandwidth links of this testbed. As expected, the protocols with a larger number of message exchanges suffer more due to the network delays.

7.3 Atomic Broadcast Analysis

This section evaluates the atomic broadcast protocol in more detail. The experiments were carried out by having the n processes send a burst of k messages and measuring the interval between the beginning of the burst and the delivery of the last message. The benchmark was performed in the following way: processes wait for a 1-byte UDP message from the signaling machine, and then each one atomically broadcasts a burst of $\frac{k}{n}$ messages. Messages

TABLE 4
Average Latency for Isolated Executions of Each Protocol in Testbed *tb-wan* (with Four Processes)

	Latency (ms)
Echo Broadcast	312.62
Reliable Broadcast	486.24
Binary Consensus	1408.75
Multi-valued Consensus	2232.30
Vector Consensus	2629.34
Atomic Broadcast	2998.70

have a fixed size of m bytes. For every tested workload, the obtained measurement reflects the average value of 10 executions.

Two metrics are used to assess the performance of the atomic broadcast: *burst latency* (L_{burst}) and *maximum throughput* (T_{max}). The burst latency is always measured at a specific process and is the interval between the instant when it receives the signal message and the moment when it delivers the kth message. The throughput for a specific burst is the burst size k divided by the burst latency L_{burst} (in seconds). The maximum throughput T_{max} can be inferred as the value at which the throughput stabilizes (i.e., does not change with increasing burst sizes). Although no graphs for the burst latency are provided due to space constrains, by dividing the burst size by the throughput value one can obtain the corresponding burst latency in seconds.

The measurements were taken by varying several system parameters: group size, network bandwidth, fault load, and message payload size. In the LAN environment, the impact of all these parameters is tested. In the WAN environment, only the fault load and the payload size are tested.

The *group size* defines the number of processes n in the system and can assume three values: 4, 7, and 10.

The *network bandwidth* is the amount of data that can be passed between every pair of processes in a given period of time. It can take three values: 10 Mbps, 100 Mbps, and 1,000 Mbps.

The *fault load* defines the types of faults that are injected in the system during its execution. The measurements were obtained under three fault loads. In the fault-free fault load, all processes behave correctly. In the fail-stop fault load, f processes crash before the measurements are taken (f is always set to the maximum number of processes that can fail as dictated by the system model, which means that $f = \lfloor \frac{n-1}{3} \rfloor$). Finally, in the Byzantine fault load, f processes permanently try to disrupt the behavior of the protocols. At the binary consensus layer, they always propose zero trying to impose a zero decision. At the multivalued consensus layer, they always propose the default value in both INIT and VECT messages trying to force correct processes to decide on the default value. The impact of any such attack, if successful, would be that correct processes do not reach an agreement over which messages should be delivered by the atomic broadcast protocol and, consequently, would have to start a new agreement round.

The *message payload size* is the length of the data transmitted in each atomic broadcast (excluding protocol headers). Four values were used in the experiments: 10 bytes, 100 bytes, 1 Kbyte, and 10 Kbytes.

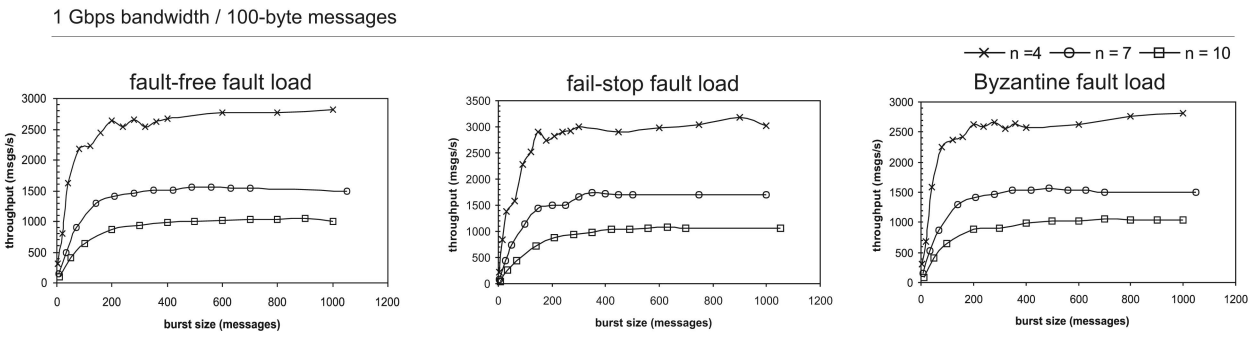

Fig. 4. Throughput for atomic broadcast with different group sizes and fault loads for testbed *tb-lan-fast*.

7.3.1 Group Size and Fault Load in LAN

The set of experiments described in this section had the objective of measuring the impact of both the group size and the fault load in a LAN environment. The network bandwidth was fixed to 100 Mbps in testbed *tb-lan-slow* and to 1,000 Mbps in testbed *tb-lan-fast*. The message payload size was 100 bytes. The group size was set for 4, 7, and 10 processes. All three fault loads were tested: fault-free, fail-stop, and Byzantine.

Fig. 4 shows the performance of the atomic broadcast in testbed *tb-lan-fast* for the three different fault loads. Each curve shows the throughput for a different group size n.

Fault-free fault load. From the graph in Fig. 4, it is possible to observe that the stabilization point in the throughput curves indicates the maximum throughput T_{max}. This value was around 2,800 messages/s for a group size of four processes, 1,500 msgs/s for seven processes, and 1,000 msgs/s for 10 processes. The burst latency for a burst size of 1,000 was 354, 700, and 995 ms for 4, 7, and 10 processes, respectively. The group size had a significant impact on the protocol performance. The maximum throughput dropped almost to half from the 4-process to the 7-process scenario, and then about one third from the 7-process to the 10-process scenario. These results were expected because larger group sizes implicate that a larger number of messages must be exchanged. This imposes a higher load on the network, which decreases the maximum throughput.

Fail-stop fault load. In this fault load, where f processes crash, each correct process sends a burst of $\frac{k}{n-f}$ messages. Looking at the curves, it is possible to conclude that performance is noticeably better with f crashed processes than in the fault-free situation. This happens because with f fewer processes there are fewer messages. The decreased contention, which does not necessarily occur at the network since the individual nodes are also susceptible to resource contention, allows operations to be executed faster. The maximum throughput T_{max} is around 3,000 messages/s for a group size of four processes, 1,700 msgs/s for seven processes, and 1,050 msgs/s for 10 processes. The burst latency for a burst size of 1,000 was 330, 587, and 989 ms for 4, 7, and 10 processes, respectively.

Byzantine fault load. In this fault load, f processes try to disrupt the protocol. The maximum throughput T_{max} is around 2,800 messages/s for a group size of four processes, 1,500 msgs/s for seven processes, and 1,000 msgs/s for 10 processes. The burst latency for a burst size of 1,000 was 355, 704, and 966 ms for 4, 7, and 10 processes, respectively.

There is no noticeable performance penalty when compared to the fault-free fault load. An important result is that all the consensus protocols reached agreement within one round, even under Byzantine faults. This can be explained in an intuitive way as follows. The experimental setting was a LAN, which not only provides a low-latency, high-throughput environment but also keeps the nodes within symmetrical distance of each other. Due to this symmetry, in the atomic broadcast protocol, correct processes maintained a fairly consistent view of the received AB_MSG messages because they all received these messages at approximately the same time. Any slight inconsistencies that, on occasion, existed over this view were squandered when processes broadcast vector V (which was built with the identifiers of the received AB_MSG messages) and then constructed a new vector W (which serves as the proposal for the multivalued consensus) with the identifiers that appeared in, at least, $f+1$ of those V vectors. This mechanism caused all correct processes to propose identical values in every instance of the multivalued consensus, which allowed one-round decisions.

Testbed tb-lan-slow versus tb-lan-fast. Fig. 6a compares the performance for the fault-free and fail-stop scenarios with four processes in both testbeds. The curves for the Byzantine scenario were left out for legibility since, as observed above, they are practically the same as for the fault-free scenario. The bandwidth for testbed *tb-lan-slow* is 100 Mbps, and for *tb-lan-fast* is set to 1,000 Mbps.

Unsurprisingly, it can be observed that the performance is clearly superior in testbed *tb-lan-fast*. The greater computational power and network capacity of *tb-lan-fast* allows a maximum throughput about four times larger in the fault-free scenario (2,800 msgs/s versus 650 msgs/s), and three times larger in the fail-stop scenario (3,000 msgs/s versus 1,000 msgs/s). The performance factor is larger in the fault-free case because the load increase in this scenario w.r.t. the fail-stop scenario pushes tb-lan-slow closer to its limit (i.e., consumes a greater percentage of resources) than tb-lan-fast.

7.3.2 Network Bandwidth and Message Size in LAN

This section analyzes in greater detail the impact of network bandwidth and message payload size in the protocol performance. In these experiments, no faults were injected and the group size was set to four processes. The

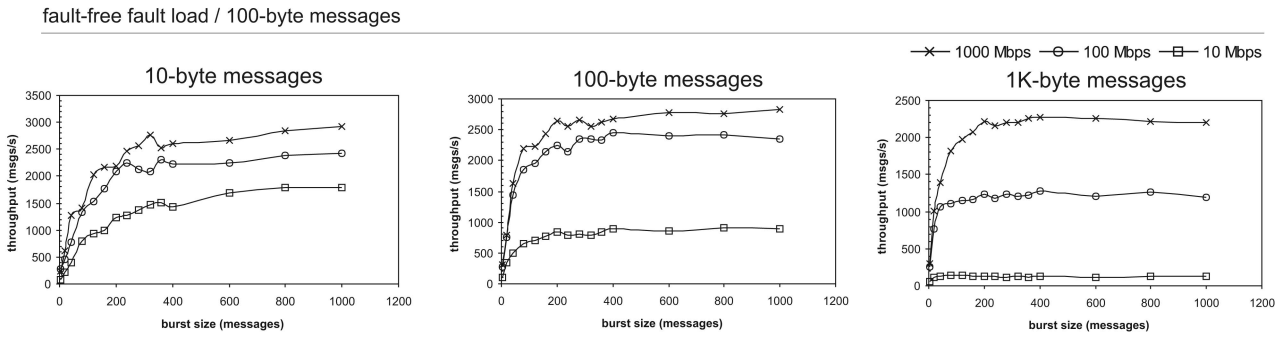

Fig. 5. Throughput for atomic broadcast with different bandwidth settings and message sizes for testbed *tb-lan-fast*.

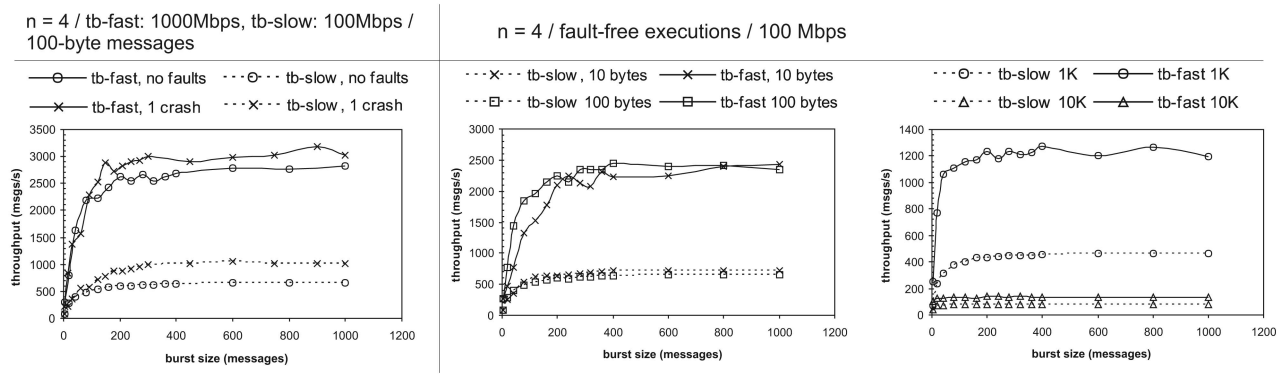

Fig. 6. Comparative throughput for atomic broadcast for testbeds *tb-lan-slow* and *tb-lan-fast*.

network bandwidth was 1,000, 100, and 10 Mbps for testbed *tb-lan-fast*, and 100 Mbps for testbed *tb-lan-slow*. Four message payload sizes were used: 10 bytes, 100 bytes, 1 Kbyte, and 10 Kbytes.

Fig. 5 shows the performance curves for testbed *tb-lan-fast* with 10-byte, 100-byte, and 1-Kbyte message payloads. Each curve represents a different bandwidth value.

While there is a clear performance difference between the protocol execution in the three network bandwidth scenarios, it is not accentuated as one would expect, if considering the bandwidth as the sole performance bottleneck. For instance, while the 1,000-Mbps scenario has 100 times more bandwidth than the 10-Mbps scenario, the maximum throughput is only about 1.6 higher in the 1,000-Mbps case (2,900 msgs/s versus 1,800 msgs/s) with 10-byte messages. It is only for larger message payloads that the network bandwidth becomes a restricting factor. As later experiments confirm, the processing power of the individual nodes and the network latency considerably affect the performance, especially for small payload sizes.

Finally, the charts of Figs. 6b and 6c compare the protocol performance on both testbeds with similar bandwidth values. The purpose is solely to compare the impact of the individual node computational power on the protocol performance. As can be easily observed, testbed *tb-lan-fast* clearly outperforms testbed *tb-lan-slow*. It is only for large payloads (e.g., 10 Kbytes) that their performance becomes comparable as both the network bandwidth and latency become more restricting factors.

7.3.3 Fault Load and Message Size in WAN

This section describes the experiments that measure the impact of the message payload size and different types of faults in the WAN environment (testbed *tb-wan*).

Fault-free fault load. The chart of Fig. 7a shows the performance of the atomic broadcast in testbed *tb-wan* when

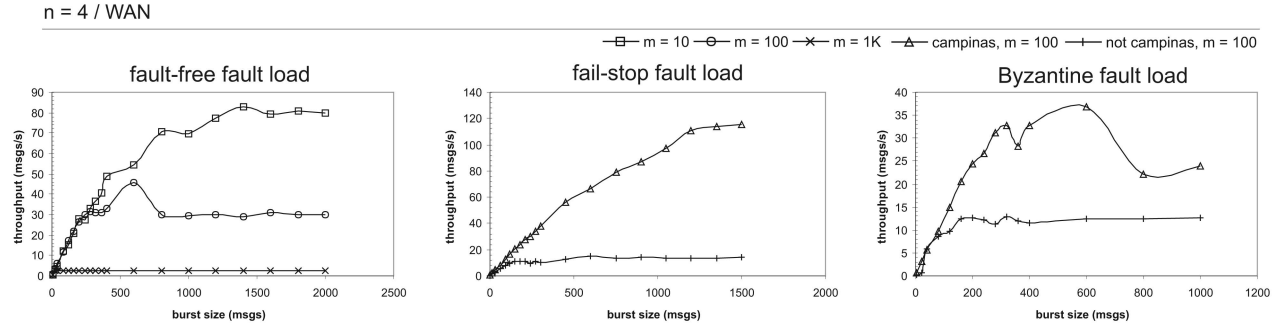

Fig. 7. Throughput for atomic broadcast for testbed *tb-wan*.

no faults occur in the system. Each curve shows the throughput for a different payload size. For 10-byte payloads, the maximum throughput is around 80 msgs/s (burst latency of 13 seconds for $k = 1{,}000$). For 100-byte payloads, the maximum throughput is around 32 msgs/s (burst latency of 34 seconds for $k = 1{,}000$). Finally, for 1-Kbyte payloads, the throughput stabilizes around 25 msgs/s (burst latency of 400 seconds for $k = 1{,}000$).

As expected, the throughput in the WAN environment is considerably lower than in the LAN environment. The higher latency and lower bandwidth of such an environment has a negative impact on the atomic broadcast performance, and makes it extremely sensitive to the message payload size.

Fail-stop fault load. The chart of Fig. 7b shows the performance of the atomic broadcast (using 100-byte payload messages) in testbed *tb-wan* when one of the processes fails by crashing. One curve shows the performance impact on the atomic broadcast protocol when the Campinas node crashes, and the other curve when a node other than Campinas crashes. When the crashed node is not Campinas, the performance of the protocol is similar for all the remaining scenarios, with the throughput stabilizing around 14 msgs/s (burst latency of 71 seconds for $k = 1{,}000$). On the other hand, when the crashed node is Campinas, the performance of the atomic broadcast is boosted to around 120 msgs/s (burst latency of eight seconds for $k = 1{,}000$), a significant increase even if compared to the fault-free scenario.

The first observation of these results is that when the crashed node is not Campinas, the performance is worse than the fault-free scenario by about 50 percent. Messages from the Campinas node were consistently the last ones to arrive at any given process for any particular communication step. This observation is coherent with the latency and bandwidth measurements taken. The links connecting to the Campinas node had the worst results on average. The conclusion is that the Campinas node is a performance bottleneck. When one process crashes (other than Campinas), this forces all processes to wait for the Campinas messages at every communication step. In the fault-free scenario, this is offset by the fact that the other processes need not wait for the Campinas messages to advance in the execution of the protocols. They only need to wait for the messages that Campinas atomically broadcasts (but not the messages related to agreement executions) since, by definition of the experiment, all processes wait for $\frac{k}{n}$ messages from each process.

The second observation is that the atomic broadcast has a considerably higher throughput when the crashed node is Campinas. This can be explained using the same rationale as the previous observation. Since the process crashed and by definition of the experiment, the other processes do not expect any messages from the Campinas node (not even atomically broadcast messages). Hence, the higher performance in this case, even when compared to the fault-free scenario. What is striking is really how much of a performance impact one slower process can have on the execution of the protocol.

Byzantine fault load. The performance of the atomic broadcast in testbed *tb-wan* is shown on the chart of Fig. 7c when one of the processes tries to disrupt the execution of the protocol. One curve shows the performance impact on the atomic broadcast protocol when the Campinas node fails, and the other curve when a node other than Campinas fails. When the Byzantine node is not Campinas, the performance of the protocol is again very similar for all the cases with the maximum throughput being roughly around 12 msgs/s. When the Byzantine node is Campinas, the throughput climbs up to around 35 msgs/s but drops to around 20-25 msgs/s for higher burst sizes (i.e., $k > 600$).

The main observations are similar for the fail-stop scenario. The protocol performance is worse when the Byzantine node is not Campinas, and better when it is Campinas. Naturally, this implies that when the Byzantine node is among the $n - f$ fastest, its power to delay the execution of the protocols is greater. Because the messages from the slower node are rarely processed by the other processes, the impact of its Byzantine actions is minimized or even nonexistent. The performance of the atomic broadcast when the Byzantine node is Campinas is similar to the fault-free scenario for burst sizes less or equal to 600. It is only when the burst rises above this threshold that the node begins to show some capacity to delay the protocol execution. When more messages are processed in the system, there is a higher chance for some of the messages sent by Campinas to be among the first $n - f$ to be received by other processes.

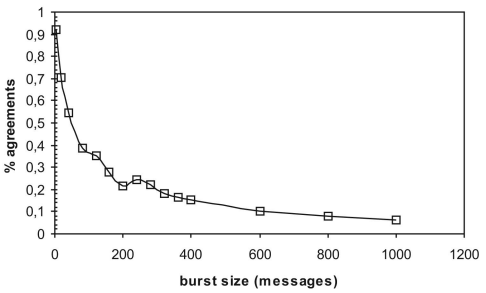

Fig. 8. Relative cost: percentage of (reliable or echo) broadcasts that are due to the agreements when a burst of messages is atomically broadcast.

7.3.4 Relative Cost of Agreement

On all experiments, only a few agreements were necessary to deliver an entire burst. The observed pattern was that a consensus was initiated immediately after the arrival of the first message. While the agreement task was being run, a significant portion of the burst would arrive, and so on until all the messages were delivered. This has the interesting effect of diluting the cost of the agreements when the load increases.

Fig. 8 shows the *relative cost* of the agreements with respect to the total number of (reliable and echo) broadcasts that was observed in the fault-free scenario with four processes and 100-byte messages in testbed *tb-lan-fast*. This relative cost is referred to as the *efficiency* of the atomic broadcast protocol. The curves for the other scenarios are almost identical; none of the testing parameters had a noticeable effect on the efficiency. Basically, two quantities

were obtained for the transmission of every burst: the total number of (reliable and echo) broadcasts and the total number of (reliable and echo) broadcasts that were necessary to execute the agreement operations. The values depicted in the figure are the second quantity divided by the first. It is possible to observe that for small burst sizes, the cost of agreement is high—in a burst of four messages, it represents about 92 percent of all broadcasts. This number, however, drops exponentially, reaching as low as 6.3 percent for a burst size of 1,000 messages.

There is a downside to this result that is related to the individual message latency under an atomic broadcast burst. According to the observed pattern, for a burst of k messages, $k-1$ are delivered exactly at the end of the burst. This means the individual message latency for those $k-1$ messages matches the whole burst latency and suggests that in a certain usage scenario the protocols could be optimized to provide a more sparse distribution of message delivery inside a burst (by sacrificing some efficiency).

7.4 Summary of Results

Some of the conclusions of the experimental evaluation are summarized in the following points:

- The protocols are robust. In LAN environments, performance (and also correctness) is not affected by the tested fault patterns.
- The protocols are efficient with respect to the number of rounds to reach agreement. In the experiments with no Byzantine failures, the multi-valued consensus always reached an agreement with a value distinct from the default $\perp$, and the binary consensus always terminated within one round.
- Since protocols do not carry out any recovery actions when a failure occurs, crashes have the effect of making executions faster for LAN environments. Fewer processes mean less contention on the network.
- The network bandwidth only becomes a serious performance bottleneck when it becomes relatively small (i.e., 10 Mbps or WAN) or the message payloads become relatively large (i.e., 1 Kbyte and 10 Kbytes).
- Protocols perform much worse in a WAN, due to its higher-latency and lower-bandwidth links.
- For LANs, the computational capability of the individual nodes has a strong influence on the protocol stack performance.
- In a WAN, the performance impact of a process crash can be positive or negative, depending on whether the process is relatively slow or relatively fast, respectively.
- A Byzantine process can have a negative impact on the performance of the protocols in a WAN environment, but only if the process can consistently broadcast valid messages that are delivered among the first $n-f$ messages for any given step.
- On the atomic broadcast protocol, the cost of the agreements is diluted when the load is high. For a burst of 1,000 messages, it represents only 6.3 percent of all (reliable or echo) broadcasts that were made.

8 Conclusion

This paper has presented an implementation and evaluation of a stack of intrusion-tolerant randomized protocols. These protocols have a set of important structural properties, such as not requiring the use of public-key cryptography (relevant for good performance) and optimal resilience (significant in terms of system cost).

The experiments led to several observations. First, randomized binary consensus protocols that in theory run in high numbers of steps, in practice, may execute in only a few rounds under realistic conditions. Second, although atomic broadcast is equivalent to consensus, with the right implementation, a high number of atomic broadcasts can be done with a small number of rounds of consensus. Consequently, the average cost in terms of throughput for atomic broadcast can be almost as little as a reliable broadcast. Third, taking decisions in a decentralized way is important to avoid performance penalties due to the existence of faults. In fact, the performance of our protocols is approximately the same, or even improved, with realistic fault loads.

In conclusion, randomization can, in fact, and contrary to a widespread belief in the scientific community, be a valid solution for the deployment of efficient distributed systems. This is true even if they are deployed in hostile environments where they are usually subject to malicious attacks.

Acknowledgments

This work was partially supported by the European Union through NoE IST-4-026764- NOE (RESIST) and project IST-4-027513-STP (CRUTIAL), and by the FCT through project POSI/EIA/60334/2004 (RITAS) and the Large-Scale Informatic Systems Laboratory (LASIGE).

References

[1] J.S. Fraga and D. Powell, "A Fault- and Intrusion-Tolerant File System," *Proc. Third IFIP Int'l Conf. Computer Security (IFIP/Sec '85)*, pp. 203-218, Aug. 1985.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.

[3] P.E. Verissimo, N.F. Neves, and M.P. Correia, "Intrusion-Tolerant Architectures: Concepts and Design," *Architecting Dependable Systems*, R. Lemos, C. Gacek, and A. Romanovsky, eds. Springer-Verlag, vol. 2677, 2003.

[4] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299-319, Dec. 1990.

[5] R. Guerraoui and A. Schiper, "The Generic Consensus Service," *IEEE Trans. Software Eng.*, vol. 27, no. 1, pp. 29-41, Jan. 2001.

[6] V. Hadzilacos and S. Toueg, "A Modular Approach to Fault-Tolerant Broadcasts and Related Problems," Dept. Computer Science, Cornell Univ., Technical Report TR94-1425, May 1994.

[7] M. Correia, N.F. Neves, and P. Verissimo, "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures," *The Computer J.*, vol. 41, no. 1, pp. 82-96, Jan. 2006.

[8] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, pp. 374-382, Apr. 1985.

[9] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *J. ACM*, vol. 35, no. 2, pp. 288-323, Apr. 1988.

[10] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM*, vol. 34, no. 1, pp. 77-97, Jan. 1987.
[11] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM*, vol. 43, no. 2, pp. 225-267, Mar. 1996.
[12] D. Malkhi and M. Reiter, "Unreliable Intrusion Detection in Distributed Computations," *Proc. 10th Computer Security Foundations Workshop (CSFW '97)*, pp. 116-124, June 1997.
[13] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith, "Byzantine Fault Detectors for Solving Consensus," *The Computer J.*, vol. 46, no. 1, pp. 16-35, Jan. 2003.
[14] N.F. Neves, M. Correia, and P. Verissimo, "Solving Vector Consensus with a Wormhole," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 12, Dec. 2005.
[15] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," *Proc. Second ACM Symp. Principles of Distributed Computing (PODC '83)*, pp. 27-30, Aug. 1983.
[16] M.O. Rabin, "Randomized Byzantine Generals," *Proc. 24th Ann. IEEE Symp. Foundations of Computer Science (FOCS '83)*, pp. 403-409, Nov. 1983.
[17] C. Cachin, K. Kursawe, and V. Shoup, "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography," *Proc. 19th ACM Symp. Principles of Distributed Computing (PODC '00)*, pp. 123-132, July 2000.
[18] C. Cachin and J.A. Poritz, "Secure Intrusion-Tolerant Replication on the Internet," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '02)*, pp. 167-176, June 2002.
[19] H. Moniz, M. Correia, N.F. Neves, and P. Verissimo, "Experimental Comparison of Local and Shared Coin Randomized Consensus Protocols," *Proc. 25th IEEE Symp. Reliable Distributed Systems (SRDS '06)*, pp. 235-244, Oct. 2006.
[20] R. Friedman, A. Mostefaoui, and M. Raynal, "Simple and Efficient Oracle-Based Consensus Protocols for Asynchronous Byzantine Systems," *Trans. Dependable and Secure Computing*, vol. 2, no. 1, pp. 46-56, Jan.-Mar. 2005.
[21] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," IETF Request for Comments: RFC 2093, Nov. 1998.
[22] G. Bracha, "An Asynchronous $\lfloor (n-1)/3 \rfloor$-Resilient Consensus Protocol," *Proc. Third ACM Symp. Principles of Distributed Computing (PODC '84)*, pp. 154-162, Aug. 1984.
[23] M. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," *Proc. Second ACM Conf. Computer and Comm. Security (CCS '94)*, pp. 68-80, Nov. 1994.
[24] S. Toueg, "Randomized Byzantine Agreements," *Proc. Third ACM Symp. Principles of Distributed Computing (PODC '84)*, pp. 163-178, Aug. 1984.
[25] R. Canetti and T. Rabin, "Fast Asynchronous Byzantine Agreement with Optimal Resilience," *Proc. 25th Ann. ACM Symp. Theory of Computing (STOC '93)*, pp. 42-51, 1993.
[26] L.E. Moser and P.M. Melliar-Smith, "Byzantine-Resistant Total Ordering Algorithms," *Information and Computation*, vol. 150, pp. 75-111, 1999.
[27] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy, "Consensus in Byzantine Asynchronous Systems," *Proc. Seventh Int'l Colloquium on Structural Information and Comm. Complexity (SIROCCO '00)*, pp. 1-16, June 2000.
[28] J.P. Martin and L. Alvisi, "Fast Byzantine Consensus," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '05)*, June 2005.
[29] M. Correia, N.F. Neves, L.C. Lung, and P. Verissimo, "Low Complexity Byzantine-Resilient Consensus," *Distributed Computing*, vol. 17, no. 3, pp. 237-249, 2005.
[30] H. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W.H. Sanders, "Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '02)*, pp. 229-238, June 2002.
[31] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith, "The SecureRing Group Communication System," *ACM Trans. Information and System Security*, vol. 4, no. 4, pp. 371-406, 2001.
[32] M. Correia, N.F. Neves, L.C. Lung, and P. Verissimo, "Worm-IT—A Wormhole-Based Intrusion-Tolerant Group Communication System," *J. Systems and Software*, vol. 80, no. 2, pp. 178-197, 2007.
[33] V. Drabkin, R. Friedman, and A. Kama, "Practical Byzantine Group Communication," *Proc. 26th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '06)*, p. 36, 2006.
[34] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proc. Third Symp. Operating Systems Design and Implementation (OSDI '99)*, pp. 173-186, Feb. 1999.
[35] H. Moniz, "Randomized Intrusion-Tolerant Asynchronous Services, Master's Thesis," Dept. Informatics, Univ. of Lisbon, master's thesis, DI/FCUL TR-07-2, http://www.di.fc.ul.pt/tech-reports/07-2.pdf, Feb. 2007.
[36] R. van Renesse, K.P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Comm. ACM*, vol. 39, no. 4, pp. 76-83, 1996.
[37] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building Adaptive Systems Using Ensemble," *Software—Practice and Experience*, vol. 28, no. 9, pp. 963-979, 1998.
[38] G.R. Wright and W.R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, 1995.
[39] G.S. Veronese, M. Correia, L.C. Lung, and P. Verissimo, "On the Effects of Finite Memory on Intrusion-Tolerant Systems," *Proc. 13th Pacific Rim Int'l Symp. Dependable Computing (PRDC '07)*, pp. 401-404, 2007.
[40] M.K. Reiter, "The Rampart Toolkit for Building High-Integrity Services," *Proc. Int'l Workshop Theory and Practice in Distributed Systems*, pp. 99-110, 1995.
[41] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: An Overlay Testbed for Broad-Coverage Services," *SIGCOMM Computer Comm. Rev.*, vol. 33, no. 3, pp. 3-12, 2003.

Henrique Moniz is a PhD student in the Departamento de Informática, Faculdade de Cincias, Universidade de Lisboa, Portugal. He is a member of the LASIGE laboratory and the Navigators research group. He is also a teaching assistant at the University of Lisbon. His research interests include distributed algorithms in hostile environments. He is involved in several research projects related to intrusion tolerance and security, including the RITAS (FCT), the CRUTIAL and HIDENETS (EC-IST) projects, and the ReSIST NoE. He is a student member of the IEEE.

Nuno Ferreira Neves received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1998. He is an assistant professor in the Departamento de Informática, Faculdade de Cincias, Universidade de Lisboa, Portugal, and also an adjunct faculty at the Information Network Institute, Carnegie Mellon University, for activities related to the MSc in Information Technology-Information Security. His main research interests include dependable and secure parallel and distributed systems. In the recent years, he has participated in several European and national research projects in these areas, namely CRUTIAL, Resist, AJECT, RITAS, and MAFTIA. His work has been recognized with the IBM Scientific Prize in 2004 and the William C. Carter award at the IEEE FTCS in 1998. He is currently a member of the editorial board of the *International Journal of Critical Computer-Based Systems*. He is a member of the IEEE.

Miguel Correia received the PhD degree in computer science from the Universidade de Lisboa, Portugal, in 2003. He is an assistant professor in the Departamento de Informática, Faculdade de Cincias, Universidade de Lisboa. His main research interests include intrusion tolerance, security, distributed systems, and distributed algorithms. He is a member of the LASIGE research unit and the Navigators research team. He has been involved in several international and national research projects related to intrusion tolerance and security, including the MAFTIA and CRUTIAL EC-IST projects, and the ReSIST NoE. He is currently the coordinator of the Universidade de Lisboa's degree on Informatics Engineering and an instructor at the joint Carnegie Mellon University and Universidade de Lisboa MSc in Information Technology-Information Security. He is a member of the IEEE.

Paulo Verissimo is currently a professor in the Departamento de Informática (DI), Faculdade de Cincias, Universidade de Lisboa, Portugal (http://www.di.fc.ul.pt/~pjv) and the director of LASIGE, a research laboratory of the DI (http://lasige.di.fc.ul.pt). He leads the Navigators research group of LASIGE. His current research interests include architecture, middleware, and protocols for distributed, pervasive, and embedded systems, in the facets of real-time adaptability and fault/intrusion tolerance. He has authored more than 130 refereed publications in international scientific conferences and journals in these areas, and has coauthored five books (e.g., http://www.navigators.di.fc.ul.pt/dssa/). He is an associate editor of the *Elsevier International Journal on Critical Infrastructure Protection* and a past associate editor of the *IEEE Transactions on Dependable and Secure Computing*. He belonged to the European Security and Dependability Advisory Board. He is the past chair of the IEEE Technical Committee on Fault-Tolerant Computing and of the Steering Committee of the DSN conference, and belonged to the Executive Board of the CaberNet European Network of Excellence. He was a coordinator of the CORTEX IST/FET project (http://cortex.di.fc.ul.pt). He is a fellow of the IEEE.