Randomized Intrusion-Tolerant Asynchronous Services*

Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Veríssimo
University of Lisboa, Portugal
{hmoniz, nuno, mpc, pjv}@di.fc.ul.pt

Abstract

Randomized agreement protocols have been around for more than two decades. Often assumed to be inefficient due to their high expected communication and time complexities, they have remained largely 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 different faultloads. 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 and no performance reduction is observed under certain Byzantine faults.

1. Introduction

With the increasing need of our society to deal with computer- and network-based attacks, the area of *intrusion tolerance* has been gaining momentum over the past few years [25]. 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.

A pivotal problem in fault- and intrusion-tolerant distributed systems is *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. The relevance of consensus is considerable because it has been shown equivalent to several other distributed problems, such as state machine replication [23] and atomic broadcast [9]. Consensus, however, cannot be solved deterministically in asynchronous systems if a single processe can crash (also known as the FLP impossibility result [12]). 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 techniques have been proposed to circumvent the FLP result. Most of these solutions, however, require changes to the basic system model, with the explicit inclusion of stronger time assumptions (e.g., partial synchrony models [10]), or by augmenting the system with devices that hide in their implementation these assumptions (e.g., failure detectors [7] or wormholes [19]). *Randomization* is another technique that has been around for more than two decades [2, 20]. One important advantage of this technique is that no time assumptions are needed – to circumvent FLP, it uses a probabilistic approach where the termination of consensus is ensured with probability 1. Although this line of research produced a number of important theoretical achievements, including many algorithms, in what pertains to the implementation of practical applications, randomization has been historically overlooked because it has usually been considered to be too inefficient.

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. Many of these algorithms also rely heavily on public-key cryptography, which increases the performance costs, especially for 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

*This work was partially supported by the EU 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).

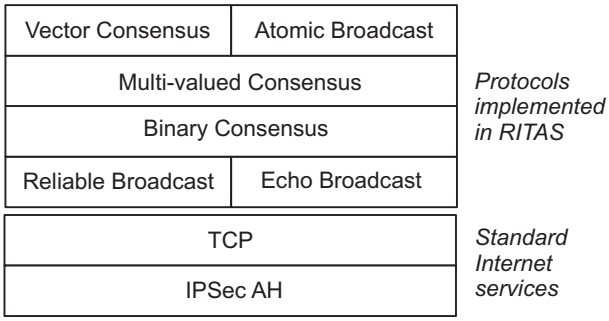

Figure 1. The RITAS protocol stack.

termination (e.g., many processes proposing the same value can lead to a quicker conclusion). 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 more simpler) manner to prevent the conclusion of the algorithm – for example, it could block the communication entirely. Therefore, in practice, the network scheduling can be "nice" and lead to a speedy termination.

The paper describes the implementation of a stack of randomized intrusion-tolerant protocols and evaluates their performance under different faultloads. 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 Figure 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 the top of this one. Building on the *binary consensus* layer is the *multi-valued consensus*, allowing the agrement 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 [13]. All these protocols have been previously described in the literature [3, 22, 9]. The implemented protocols are, in most cases, optimized versions of the original proposals that have significantly improved the overall performance (see Section 2 for a description of some of these optimizations).

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's relative execution and communication times, 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 lower 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.

The paper has two main contributions: 1) it presents the 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 above properties is novel; 2) it provides a detailed evaluation of RITAS in a LAN setting, showing that it has interesting latency and throughput values; for example, the binary consensus protocol always runs in only one round (three communication steps) with realistic faultloads, and the atomic broadcast has a very low ordering overhead (only 2.4%) when the rate of transmitted messages is high; moreover, some experimental results show that realistic Byzantine faults do not reduce the performance of the protocols.

2. System Model and Protocol Definitions

The system is composed by a group of n processes $P = \{p_0, p_1, ..., p_{n-1}\}$. Processes are said to be *correct* if they do not *fail*, i.e., if they follow their protocol until termination. Processes that fail are said to be *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. This class of unconstrained faults is usually called *arbitrary* or *Byzantine*. It is assumed that at most $f = \lfloor \frac{n-1}{3} \rfloor$ processes can be corrupt, which implies that $n \geq 3f + 1$. The system is asynchronous in the sense that there are no assumptions about bounds on processing times or communication delays.

The processes are assumed to be fully-connected. Each pair of processes (p_i, p_j) shares a secret key s_{ij}. It is out of the scope of the paper 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 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 arbitrarily length input m 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*.

The rest of the 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 the paper due to lack of space.

2.1. Reliable Channel

The two layers at the bottom of the stack implement a reliable channel (see Figure 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 [13].

2.2. Reliable Broadcast

The *reliable broadcast* primitive ensures two properties: (1) all correct processes deliver the same messages; (2) if the sender is correct then the message is delivered. The implemented reliable broadcast protocol was originally proposed by Bracha [3]. 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. Figure 2 illustrates the three communication steps of the protocol.

2.3. 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 broadcasted 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.

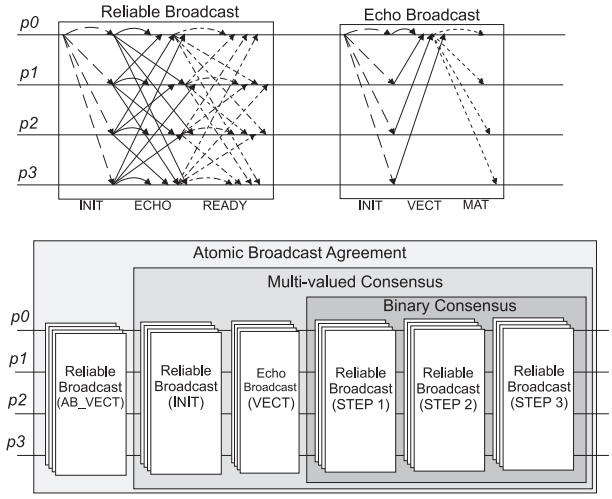

Figure 2. Overview of the messages exchanged in each protocol.

The implemented protocol – that we call *matrix echo broadcast* – is based on the *echo multicast* proposed by Reiter [22], in which digital signatures are replaced by vectors of hashes in order to improve the performance. The protocol starts with the sender broadcasting a (INIT, m) message. Upon receiving this message, each process p_i builds a vector V_i with $V_i[j] = H(m, s_{ij})$ for every $0 \leq j < n$. The hash function H is applied to a concatenation of m with the secret key shared with each process, s_{ij}. This is a simple and efficient form of *Message Authentication Code*, which guarantees that message integrity can be checked by p_j [17]. Then, p_i transmits a (VECT, i, V_i) message to the sender. The sender gathers $n-f$ vectors and uses them to construct a matrix M (each V_j becomes line j of the matrix). Next, the sender transmits to each process p_j a message (MAT, V'_j) where vector V'_j is the column of the matrix with index j. Upon receiving vector V'_i, p_i verifies the hashes and delivers the message if at least $f+1$ hashes are correct.

2.4. 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 [3]. 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 3-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 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 $k-1$. 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. 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.

2.5. Multi-valued Consensus

A *multi-valued 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 multi-valued consensus proposed by Correia et al. [9]. 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: (a) $v = \perp$; (b) 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 then it decides on that value.

2.6. 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 [9] and uses *reliable broadcast* and *multi-valued 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 *multi-valued 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.

2.7. 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 [9]. The main difference is that it has been adapted to use multi-valued 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 *multi-valued consensus* as primitives.

The atomic broadcast protocol is conceptually divided in two tasks: (1) the broadcasting of messages, and (2) the agreement over which messages should be delivered (only this part appears in Figure 2). 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 reliable 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 *multi-valued 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.

3. Implementation

This Section describes some aspects of the RITAS implementation and provides insight into the design considerations and practical issues that arose during development. The protocol stack was implemented in the C language as a shared library, which provides a simple interface to applications wishing to use the protocols. The protocol stack runs in a single thread, independent of the application thread.

3.1. Interface

The API of RITAS revolves around a data structure `ritas_t`. This structure holds all the necessary context – variables and data structures – for a communication session and is completely opaque to the application programmer. The functions provided by the API can be divided into two categories: context management and service requests. A typical RITAS session is composed by four basic steps executed by each process: 1) initialize the RITAS context by calling `ritas_init();` 2) add the participating processes to the context by calling `ritas_proc_add_ipv4();` 3) call the service request functions as many times as desired; 4) destroy the RITAS context by calling `ritas_destroy()`. There are service request functions for the broadcast, and consensus protocols. Each broadcast protocol has associated two functions: `ritas_XX_bcast()` is utilized to transmit a message, and `ritas_XX_recv()` blocks the program until a message arrives (where `XX` can be `rb`, `eb`, or `ab`, for reliable broadcast, echo broadcast, or atomic broadcast, respectively). Each consensus protocol has an associated `ritas_YY()` function that proposes a value, blocks until a decision is made, and returns the decision value (where `YY` can be `bc`, `mvc`, or `vc`, for binary consensus, multi-valued consensus, or vector consensus, respectively).

3.2. Internal Data Structures

Internally, three major data structures form the core of the RITAS operation. The aforementioned RITAS context, the message buffers, and the protocol handlers. Additionally, several data structures and functions provide ancillary common operations for general use.

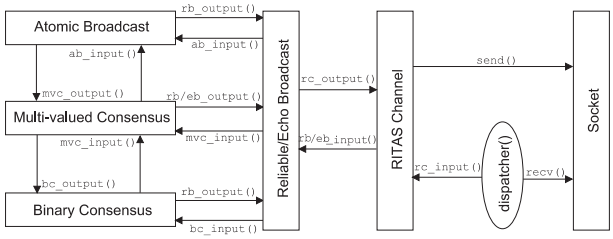

Figure 3. Communication flow among the protocol layers during an atomic broadcast.

Information is passed along the protocol stack using *message buffers* (*mbuf* for short). This data structure was inspired by the TCP/IP implementation in the Net/3 Operating System kernel [26]. *mbuf* is used to store messages and several metadata related to their management and one instance of *mbuf* can only hold a single message. All communication between the different layers is done by passing pointers to *mbufs*.

There is one data structure, the *control block*, which holds all the necessary information for an instance of a protocol. All protocols share a common internal interface. Protocols provide initialization and destruction functions which serve, 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. For inter-protocol communication two functions are provided: an input function that receives as parameters the respective control block and the *mbuf* to be processed, and an output function which uses similar parameters as the input function. The communication between the protocols is depicted in Figure 3.

There is a special protocol handler called *RITAS Channel* 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). Its purpose is to build a header containing an 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, described in the next section.

3.3. Control Block Chaining

One important mechanism used in RITAS to manage the linking of different protocol instances is the *control block chaining*. This mechanism solves several problems – it gives a means to unambiguously identifying all messages, provides for seamless protocol demultiplexing, and facili-

tates control block management.

Control block chaining works as follows. Suppose an application creates an atomic broadcast protocol instance. This task is done by calling the corresponding initialization function that returns a pointer to a control block responsible for that instance. Since atomic broadcast uses multi-valued consensus and reliable broadcast as primitives, the atomic broadcast initialization function also calls the initialization functions of 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. This creates a tree of control blocks that has its root in 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 `ritas_*_input()` function of its parent control block. The message is processed by that layer and the *mbuf* keeps being passed in the same fashion.

When a protocol instance is destroyed, all of its child protocol instances become obsolete. All protocol instances are responsible for destroying its child instances by calling the appropriate destruction functions. This way, a tree (or subtree) of control blocks is automatically destroyed when its root node is eliminated.

3.4. Out-of-Context Messages

The asynchronous nature of the protocol stack leads to situations in which a process is receiving correct messages but they are destined to a protocol instance for which a control block has not yet been created. These messages – called *out-of-context* (OOC) messages – have no context to handle them, though they will, eventually.

Since OOC messages cannot be discarded, they are stored in a hash table. When a RITAS Channel is created, it checks the hash table for 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 avoid a situation where OOC messages are kept indefinitely in the hash table, upon the destruction of a protocol, the hash table is checked and all the relevant messages are deleted.

4. Performance Evaluation

This section describes the performance experiments with RITAS in a local-area network (LAN) setting with four processes, each one running on a distinct host. Two different performance analyses are made. First, we present a comparative evaluation in order to understand how protocols relate and build on one another performance-wise. Second, we conduct an in-depth analysis of how atomic broadcast performs under various conditions. This protocol is the most interesting candidate for a detailed study because it uses all other protocols as primitives, either directly or indirectly, and it can be used for many practical applications.

The experiments were carried out on a testbed consisting of four Dell Pentium III PCs, each with 500 Mhz of clock speed and 128 MB of RAM, running Linux Kernel 2.6.5. The PCs were connected by an 100 Mbps HP ProCurve 2424M network switch. Bandwidth tests taken at different occasions with the network performance tool *lperf* have shown a consistent throughput of 9.1 MB/s. The used IPSec implementation was the one available in the Linux kernel and the *security associations* that were established between every pair of processes employed the AH protocol (with SHA-1) in transport mode [13].

4.1. Stack Analysis

In order to get a better understanding of 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 N messages, each one separated by a two second interval (in our case N 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. The broadcasted 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.

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

Table 1. Average latency for isolated executions of each protocol (with IPSec and IP).

The results, shown in Table 1, demonstrate the interdependencies among protocols and how much time is spent on each protocol. For example, in a single atomic broadcast instance roughly $\frac{2}{3}$ of the time is taken running a multi-valued consensus (see also Figure 2). For a multi-valued consensus about $\frac{1}{2}$ of the time is used by the binary consensus. And for vector consensus about $\frac{3}{4}$ of the time is utilized by the multi-valued consensus. The experiments also show that consensus protocols were always able to reach a decision in one round because the initial proposals were identical.

The cost of using IPSec is also represented in Table 1. The 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 header adds another 24 bytes, which accounts for an extra 30%.

4.2. 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 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 k^{th} 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).

The measurements were taken under three different faultloads. In the failure-free faultload all processes behave correctly. In the fail-stop faultload one process crashes before the measurements are taken (1 is the maximum number of processes that can fail because $n \geq 3f + 1$). Finally, in the Byzantine faultload one process permanently tries to disrupt the protocols. At the binary consensus layer, it always proposes zero trying to impose a zero decision. At the multi-valued consensus layer, it always proposes the default value in both INIT and VECT messages, trying to force correct processes to decide on the default value. The impact of 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, consequentely, would have to start a new agreement round.

Failure-free faultload. Figure 4 shows the performance of the atomic broadcast when no faults occur in the system. Each curve shows the latency or throughput for a different message size m. From the graph it is possible to observe that the burst latency L_{burst} is linear with the burst size. The stabilization point in the throughput curves indicates the maximum throughput T_{max}.

For a burst of 1000 messages, L_{burst} has a value of 1386 ms for a message size of 10 bytes, 1539 ms for 100 bytes, 2150 ms for 1K bytes and 12340 ms for 10K bytes. The stabilization point in the throughput curves indicates the maximum throughput T_{max}. The throughput stabilizes around 721 messages per second for a message size of 10 bytes, 650 msgs/s for 100 bytes, 465 msgs/s for 1K bytes, and 81 msgs/s for 10K bytes. These results were expected because larger messages impose a higher load on the network, which decreases the maximum throughput.

Fail-stop faultload. The performance of the atomic broadcast protocol with one crashed process is presented in Figure 5. In this faultload, each correct process sends a burst of $\frac{k}{n-1}$ messages. Each curve shows the latency or throughput for a different message size m.

Looking at the curves, it is possible to conclude that performance is noticeably better with one fail-stop process than in the failure-free scenario. This is because with one less process there is less contention in the network allowing operations to be executed faster. In more detail, the numbers show for a burst of 1000 messages, L_{burst} has a value of 988 ms for 10-byte messages, 1164 ms for 100 bytes, 1607 ms for 1K bytes, and 8655 ms 10K bytes. The maximum throughput T_{max} is around 858 messages per second for a message size of 10 bytes, 621 msgs/s for 100 bytes, 834 msgs/s for 1K bytes, and 115 msgs/s for 10K bytes.

Byzantine faultload. Figure 6 shows the performance of atomic broadcast for different message sizes, with one process trying to disrupt the protocol. In more detail, the numbers for a burst of 1000 messages are: L_{burst} has a value

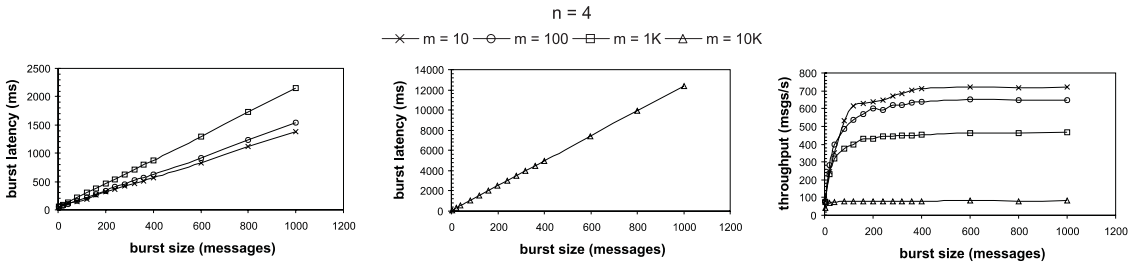

Figure 4. Latency and throughput for atomic broadcast with failure-free faultload.

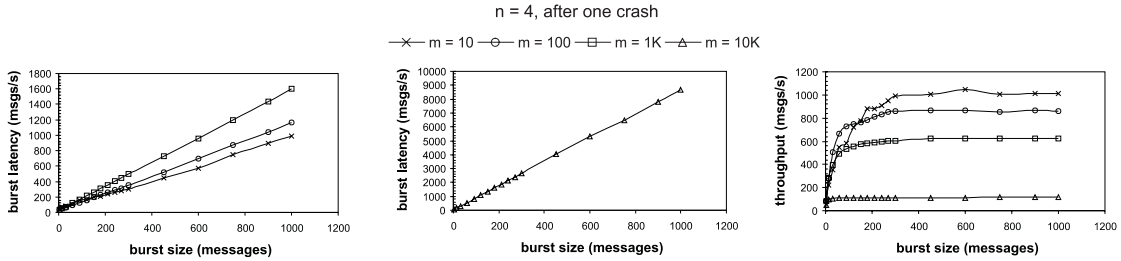

Figure 5. Latency and throughput for atomic broadcast with fail-stop faultload.

of 1404 ms for 10-byte messages, 1576 ms for 100-byte messages, 2175 ms for 1K-byte messages, and 12347 ms for 10K-byte messages. The maximum throughput T_{max} is around 711 messages per second for 10-byte messages, 634 msgs/s for 100 bytes, 460 msgs/s for 1K bytes, and 81 msgs/s for 10K bytes. From the numbers it is possible to observe that performance is basically immune from the attacks of the Byzantine process, when compared with the failure-free scenario. The Byzantine process never managed to foil any of the consensus protocols due to the robustness of the atomic broadcast protocol.

An important result is that all the consensus protocols reached agreement within one round, even under Byzantine faults. This can be explained in a intuitive way as follows. The experimental setting was a LAN, which not only provides a low-latency, high-throughput environment, it also keeps the nodes within simmetrical distance of each other. Due to this simmetry, 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 relatively the same time. Any slight inconsistencies that ocasionally existed over this view were squandered when processes broadcasted the 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 multi-valued 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 multi-value consensus, which allowed one-round decisions. In a more asymmetrical environment, like a WAN, it is not guaranteed that this result can be reproduced.

Relative Cost of Agreement. On all experiments only two 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 first agreement task was being run, the remaining burst would be received. Therefore, this remaining set of messages could be delivered with a second agreement. This behavior has the interesting effect of diluting the cost of the agreements as the load increases.

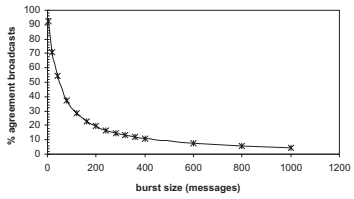

Figure 7. Percentage of broadcasts that are due to the agreements.

Figure 7 shows the relative cost of the agreements with respect to the total number of (reliable and echo) broadcasts that was observed in the experiments. Basically, two quantities were obtained for the transmission of every burst: the total number of (reliable and echo) broadcasts, and the total number of 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 4 messages, it represents about 92% of

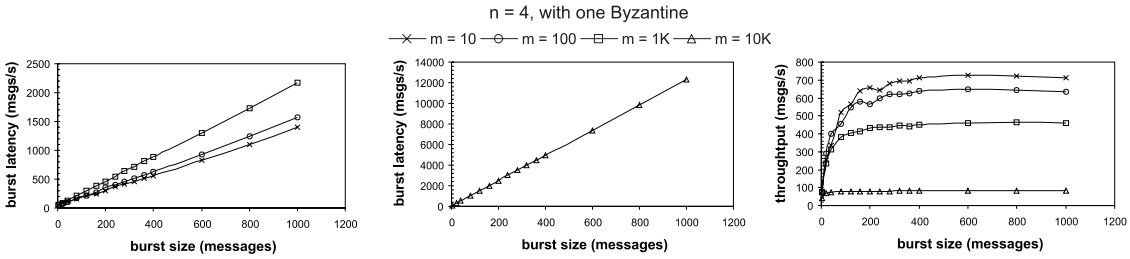

Figure 6. Latency and throughput for atomic broadcast with Byzantine faultload.

all broadcasts. This number, however, drops exponentially, reaching as low as 2.4% for a burst size of 1000 messages.

4.3. Summary of Main Results

The protocols are robust. Performance (and correctness) is not affected by the tested fault patterns, even when a malicious process tries to delay the execution of the protocols.

The protocols are efficient with respect to the number of rounds to reach agreement. In the experiments, 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 special actions when a failure occurs, crashes have the effect of making executions faster. Less processes means less contention on the network.

On the atomic broadcast protocol, the cost of the agreements is diluted when the load is high. For a burst of 1000 messages, it represents only 2.4% of all (reliable or echo) broadcasts that were made.

5. Related Work

Randomized intrusion-tolerant protocols have been around since Ben-Or's and Rabin's seminal consensus protocols [2, 20]. These two 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 so all processes see the same coins.

Although many randomized asynchronous protocols have been designed throughout the years [2, 20, 3, 24, 6, 18], only recently one implementation of a stack of randomized multicast and agreement protocols has been reported, SINTRA [5]. These protocols are built on top of a binary consensus protocol that in practice terminates in one or two communication steps [4]. The implementation of the stack is in Java and uses several threads. The protocols depend heavily on public-key cryptography primitives like digital and threshold signatures. The performance values are presented in *time between successive deliveries* (TBSD). In a LAN, the average TBSD for atomic and reliable broadcast was, respectively, 690 ms and 130 ms. The inverses give throughput values of 1.45 and 7.69 msgs/s, respectively. There is no information about latencies. RITAS uses a Ben-Or-style protocol that uses no public-key cryptography and theoretically runs in expected 2^{n-f} communication steps [3] but, in practice, in a LAN with realistic faultloads, we observed that it runs in only three communication steps.

Randomization is only one of the techniques that can be used to circumvent the FLP impossibility result. Other techniques include failure detectors [15, 1, 16], partial-synchrony [11] and distributed wormholes [8, 19]. It has been proven that deterministic asynchronous consensus requires a minimum of two communication steps in fault-free executions [16]. The same number of steps has been shown to be attainable extending the "normal" asynchronous system with a synchronous and secure distributed component called a wormhole, even when faults occur [8]. Partially synchronous protocols have been presented that run in a minimum of four steps in fault-free executions [11].

The first evaluation of a set of asynchronous Byzantine protocols (reliable and atomic broadcast) was made for the Rampart toolkit [22]. The reliable broadcast is implemented by Reiter's echo broadcast (see Section 2) 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), Reiter 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 detection is very costly in terms of time [21] and requires synchrony assumptions about the network delay, allowing attacks where malicious processes delay others to force their removal. Our protocols do not suffer from any of these problems since decisions (e.g., the message order) are made in a distributed way. Our experiments have shown that some attacks do not impact on the performance of our protocols.

Like Rampart, SecureRing is an intrusion-tolerant group communication system [14]. It relies on a token that rotates among the processes. This signed token takes message digests, a solution that allows a lower number of signatures and an improvement of the performance when compared to Rampart. In SecureRing malicious behavior has also to be detected, which means that it suffers from the same problems as Rampart.

6. Conclusion

The paper presents an implementation and evaluation of a stack of 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 conclusions: 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 consensus. Consequently, an atomic broadcast can cost almost as much as a reliable broadcast. Third, taking decisions in a distributed way is important to avoid performance penalties due to the existence of faults (the performance of our protocols is approximately the same, or even improved, with realistic faultloads). This property is also important to avoid attacks against time assumptions.

References

[1] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy. Consensus in Byzantine asynchronous systems. In *Proc. of the Int. Colloquium on Structural Information and Communication Complexity*, pages 1–16, June 2000.

[2] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. of the 2nd ACM Symp. on Principles of Distributed Computing*, 1983.

[3] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proc. of the 3rd ACM Symp. on Principles of Distributed Computing*, pages 154–162, Aug. 1984.

[4] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography. In *Proc. of the 19th annual ACM symp. on Principles of distributed computing*, pages 123–132, New York, NY, USA, 2000. ACM Press.

[5] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 167–176, June 2002.

[6] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. of the 25th Annual ACM Symp. on Theory of Computing*, pages 42–51, 1993.

[7] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43, 1996.

[8] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Low complexity Byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[9] M. Correia, N. F. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 41(1):82–96, Jan. 2006.

[10] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan. 1987.

[11] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[13] S. Kent and R. Atkinson. Security architecture for the internet protocol. IETF RFC 2093, Nov. 1998.

[14] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4, 2001.

[15] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proc.of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.

[16] J. P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, June 2005.

[17] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[18] L. E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Information and Computation*, 150:75–111, 1999.

[19] N. F. Neves, M. Correia, and P. Veríssimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12), Dec. 2005.

[20] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, Nov. 1983.

[21] H. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 229–238, June 2002.

[22] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, Nov. 1994.

[23] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[24] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, Aug. 1984.

[25] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*. Springer-Verlag, 2003.

[26] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison Wesley, 1995.