UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

RANDOMIZED INTRUSION-TOLERANT ASYNCHRONOUS SERVICES

Henrique Lícias Senra Moniz

MESTRADO EM INFORMÁTICA

September 2006

RANDOMIZED INTRUSION-TOLERANT ASYNCHRONOUS SERVICES

Henrique Lícias Senra Moniz

Dissertação submetida para obtenção do grau de
MESTRE EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Orientador:

Nuno Fuentecilla Maia Ferreira Neves

Co-Orientador:

Miguel Nuno Dias Alves Pupo Correia

Júri:

Henrique João Lopes Domingos

António Casimiro Costa

Maria Teresa Caeiro Chambel

September 2006

Resumo

Os protocolos distribuídos com recurso à aleatoriedade foram propostos há mais de duas décadas. Tradicionalmente, estes protocolos têm sido considerados demasiado ineficientes, uma vez que apresentam complexidades teóricas elevadas quer para a comunicação como para o tempo, o que tem impedido a sua utilização prática na concretização de sistemas tolerantes a faltas. Esta tese pretende contrariar esta visão, demonstrando que a aleatoriedade pode ser uma solução competitiva, mesmo em ambientes hostis sujeitos a faltas maliciosas. Na tese é descrita a concretização de uma de pilha de protocolos tolerantes a intrusões com recurso à aletoriedade, sendo efectuada a respectiva análise de desempenho sob diversos tipos de critérios. A pilha de protocolos fornece um conjunto de serviços relevantes, desde primitivas básicas de acordo até à difusão atómica. Os protocolos partilham de um conjunto importante de propriedades estruturais, nomeadamente, toleram faltas arbitrárias, possuem resistência óptima, são assíncronos, completamente descentralizados, e apenas usam criptografia de chave simétrica. A análise de desempenho mostra que os protocolos são eficientes e que o seu desempenho não sofre degradação mesmo quando sujeitos a certos tipos de faltas maliciosas.

PALAVRAS-CHAVE: Tolerância a Intrusões, Acordo Bizantino, Algoritmos Aleatórios, Avaliação de Desempenho.

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 thesis aims to demonstrate that randomization can be a very competitive approach even in hostile environments where arbitrary faults can occur. The implementation of 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 protocols share a set of important structural properties, namely they tolerate arbitrary faults, have an optimal resilience, are time-free, completely decentralized, and signature-free. The experimental evaluation shows that the protocols are efficient and no performance reduction is observed under certain Byzantine faults.

KEY WORDS: Intrusion Tolerance, Byzantine Agreement, Randomized Protocols, Performance Evaluation.

Acknowledgments

I want to thank a handful of people whom I believe had a significant impact, even if not in an obvious way, on this thesis.

First of all, my parents for passing on to me their exceptional genes, for providing me with a purposeful foundation as a human being, and, within their means, for always backing up my professional choices. In particular, my mother for imprinting me with a resolute assurance in my intellectual prowess, and my father, for providing me the environment, patience, and support by which I began osmosing what became my professional playground and a most important form of expression.

My advisors, Prof. Nuno Neves and Prof. Miguel Correia. Still to this day, I have no idea how they picked up me among the crowd. They invested in me, never failed to provide the space and tranquility I needed to develop my work, and always gave me a tremendous amount of trust and protection for which I am truly grateful. Along with Prof. Paulo Veríssimo, the support I received from them during my initial stages at the Navigators group was a fundamental encouragement for me to untangle the serious health problems I had then.

A special word for my ex-girlfriend, Ana Teresa, who stood next to me during most of this journey. Almost stoically, endured the side-effects of my personality, was the pillar of the emotional serenity I experienced dur-

ing most of these two years, and, ultimately, was the catalyst for the fierce personal growth I am going through in my life. Thank you for everything.

Finally, my friends, for their relenteless faith and inconditional support. I can't name them with the fear of leaving someone out, but they know who they are. During the rare moments I am able to rest my mind and truly contemplate what I have, you make me feel the most fortunate person on earth. I have no words to express the love I feel for all of you.

Lisboa, September 2006

Henrique Lícias Senra Moniz

To all who stand alone.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Impetus

Society has evolved increasingly dependent on networked computer systems over the past few decades. The Internet has become an hallmark of modern society, and the massification of networked information systems has brought forth a powerful cultural shift with an almost instantaneous sharing of information. The Internet also became a huge marketplace for companies, collaborative work has grown dramatically easier, file-sharing applications have become ubiquitous, and video and voice streaming are poised to boost another leap into personal communications. With this rapid shift in business and personal communications, the protection of this digital medium becomes one of the biggest priorities of modern society. The availability, confidentiality, and integrity of data and services is crucial for our everyday life to go smoothly.

The typical approach to secure these distributed systems has been one of almost complete prevention, i.e., to avoid successful attacks, or penetrations, at all cost. Once the breach occurred, there was not much that could

be done about it, except to declare all remaining data to lack in integrity (until proven otherwise, at least), and to reinstall the whole system from scratch. Recently, a new trend of dealing with computer (in)security has been emerging within the scientific community. The philosophy is simple. Instead of just trying to prevent every intrusion, one would also resort to some mechanisms that would provide the automatic tolerance of these intrusions. This means that the system as a whole can continue to provide a correct behavior even if some of its components fall under the control of an intelligent adversary. This area of study is called *intrusion tolerance* and arose from the intersection of two classical areas of computer science, *fault tolerance* and *security* (Avizienis et al., 2004; Fraga & Powell, 1985; Veríssimo et al., 2003).

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 they all agree in one of these values. Consensus has been shown to be equivalent to fundamental problems, such as state machine replication (Schneider, 1990), group membership (Guerraoui & Schiper, 2001), and atomic broadcast (Correia et al., 2006b; Hadzilacos & Toueg, 1994). Hence, the relevance of consensus is noteworthy because it is a building block for 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 (Fischer et al., 1985)). 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 (Dolev et al., 1987; Dwork et al., 1988)), or by augmenting the system with devices that hide in their implementation these assumptions (e.g., failure detectors (Chandra & Toueg, 1996; Malkhi & Reiter, 1997) or wormholes (Neves et al., 2005)). *Randomization* is another technique that has been around for more than two decades (Ben-Or, 1983; Rabin, 1983). One important advantage of this technique is that no additional timing assumptions are needed. It allows for the whole system to remain completely asynchronous. This is a vital feature if we are assuming the case where the system is deployed in a hostile environment and is subject to attacks that attempt to delay to execution of the protocols. Obviously, an attacker can launch an attack so strong that can stop the whole system. But this is usually easier to perform on a synchronous system, where such an attack can be conducted in a much more subtle, but efficient manner, and can even compromise the safety of the system, in addition to its liveness. 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 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[1]. 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 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 ad-

[1] There is an exception to this reasoning, which is the stack of randomized protocols proposed by Cachin et al. (Cachin et al., 2000; Cachin & Poritz, 2002). These protocols terminate in a low expected number of communication steps but they depend heavily on public-key cryptography which seriously affects their performance. This is discussed in more detail in the Related Work section.

versary 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 is one of the motivations for this thesis: to show that randomization can be efficient and should be regarded as a valid solution for practical intrusion-tolerant distributed systems. To the best of our knowledge, this is the first work that presents a thorough look into the effectiveness of randomization for practical systems.

1.2 Contributions of the Thesis

The work performed on this thesis involves the implementation of a stack of randomized intrusion-tolerant protocols for distributed systems, and the evaluation of their performance under different environmental settings and faultloads. This implementation is called RITAS which stands for *Randomized Intrusion-Tolerant Asynchronous Services* (Moniz et al., 2006). At the lowest level of the stack there are two broadcast primitives: *reliable broadcast* and *echo broadcast*. On top of these primitives, the most basic form of consensus is available, the *binary consensus*. This protocol lets processes decide on a single bit of information and is, in fact, the only randomized algorithm of the stack. The rest of the protocols are simply built on the top of this one. Building on the *binary consensus* layer is the *multi-valued consensus*, allowing 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 (Kent & Atkinson, 1998). All these protocols have been previously described in the literature (Bracha, 1984; Correia et al., 2006b; Reiter, 1994). However, 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's relative execution and communication times. This is important to prevent attacks against assumptions on the domain of time, a known problem in some intrusion-tolerant 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. This 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. This has an important impact 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). This avoids the costly operation of detecting the failure of a leader, an event that can considerably delay the execution.

The thesis has two main contributions: 1) it presents the implementation of a stack of randomized intrusion-tolerant protocols discussing sev-

eral optimizations – 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 6.3%) 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.

1.3 Organization of the Thesis

The thesis is organized as follows. Chapter 2 presents the related work by discussing the several techniques to circumvent the FLP result in consensus protocols - randomization in particular - and other implementations of related protocols. Chapter 3 thoroughly describes the stack of protocols implemented in RITAS: its architecture, the individual algorithms, and the modifications performed for optimization. The actual implementation details are presented in Chapter 4 where the design decisions and internal structure of RITAS are discussed. The performance evaluation of the RITAS protocol stack under different environmental settings and subject to different faultloads is described in Chapter 5. Finally, Chapter 6 presents the final conclusions of the thesis.

Chapter 2

Context and Related Work

This chapter provides the context in which this thesis is inserted. Since this work presents a stack of randomized agreement protocols, the chapter starts with an introduction to the consensus problem, its related problems, and the current solutions. The next section focuses on the evolution of randomization as a solution for consensus in asynchronous distributed systems. Finally, the last section describes implementations and evaluations of protocols that offer services similar to the ones provided by the protocol stack of this thesis.

2.1 Consensus

The consensus problem in distributed systems is defined in simple terms: given a set of processes where each one proposes a value, all must agree on one of these values. Behind this apparently simple problem lies the solution to important distributed system services. The consensus problem has been shown to be equivalent to several other fault-tolerant distributed systems problems, such as state machine replication, and atomic broad-

cast.

The solution to consensus seems trivial at a first impression. It is when one starts to consider the possibility of individual process failures, that things complicate. What happens if a process crashes? Do we take a decision without considering its proposal? But what if other processes have seen this proposal? And what if a process is intentionally sending different proposal values to different processes? Or omitting proposals to certain processes?

The consensus problem can be devised under several different system models. The system model can be seen as a set of parameters that abstract and dictate the behavior of the system. The two most important system parameters are the synchrony assumptions and the fault model.

About the synchrony, or timing assumptions, it is said that the system is *synchronous* if there is a known bound on the processes's relative communication and computation delays. On the other hand, the system is *asynchronous* if the relative communication and computation times are unknown, thus subject to arbitrary delays.

The fault model defines the types of faults that can exist in the system. Commonly used fault classes are: crash, omissive, and Byzantine (or arbitrary). The crash model states that processes simply stop taking any actions when they fail. In the omissive model, in spite of being subject to crash failures, the processes may exhibit occasional omissive failures (e.g., failing to send a message) which do not necessarily constitute a crash. Finally, the Byzantine model states that there are no assumptions on the types of faults the system can exhibit. This necessarily implicates that the system falls under the threat of any kind of faults, including those of malicious nature. In this thesis we will consider only Byzantine faults.

There is an additional important system parameter: the upper bound f on the number of processes that can fail. This is usually referred as the *resilience* of the system. This upper bound value is usually defined by the previous two parameters: the synchrony assumptions, and the fault model.

Besides the parameter f, there are three other parameters that model the behavior of a potential adversary. These are its scheduling capabilities, its resources, and its adaptiveness.

The scheduling capabilities may include the timing of message delivery, the individual rates of the internal clocks of the processes, and the order by which messages are delivered to the individual processes. These decisions do not need to be fixed a priori, the adversary may change their values dynamically as the protocol is executed.

The resources include both the computational and information resources the adversary may have access to. It is said that the adversary is computationally unbounded if it has access to unlimited computing power. Otherwise, it is said that the adversary is computationally bounded and there are certain actions it cannot perform (e.g., it cannot break certain types of cryptography).

The adaptiveness of the adversary is only relevant under the randomized model. When the protocols are deterministic, the adversary can determine its whole strategy a priori, thus not requiring to adapt its behavior while the protocol is being executed. On the other hand, in a randomized protocol, the adversary has no way to predict the outcome of certain steps of the protocol, thus adapting its behavior on-the-fly may be important for the adversary.

Considering that the scope of this thesis is to provide a set of proto-

cols that are tolerant to faults of malicious nature, the most interesting model is one where there are no synchrony or fault assumptions whatsoever. The asynchronous model is the most interesting because since we consider the system to be subject to malicious attacks, it allows the correctness of the protocols to be completely independent of timing constraints, making them more resilient, for instance, to denial-of-service attacks. The interest of lack of assumptions on the types of faults is obvious since we are considering an hostile environment. It is best to keep on the safe side and assume that any kind of abnormal behavior can happen.

The problem with this model is that it has been proven that consensus cannot be solved deterministically in an asynchronous system where a single process crash can occur. This result, named after its author's initials, is commonly referred as the FLP impossibility (Fischer et al., 1985).

Intuitively, the result comes from the fact that, given an asynchronous setting, it is impossible for a process that has not received a message from another process to know if that other process is faulty or if it is just slow. Since it is impossible to identify faulty processes, the correct processes can find themselves, for instance, in a state in which they are waiting indefinitely for messages from a crashed process but have no way to know that that process has crashed. By formalizing this intuition, Fischer, Lynch and Paterson proved that there is no deterministic consensus protocol that can guarantee termination in an asynchronous system where just one process can fail.

Since this surprising result, a large amount of research was made on the topic of devising alternative models in which consensus could be solved. Most of these solutions require timing assumptions, either explicitly (e.g., partial synchrony models (Dolev et al., 1987; Dwork et al., 1988)), or im-

plicitly (e.g., failure detectors (Chandra & Toueg, 1996; Malkhi & Reiter, 1997) or wormholes (Neves et al., 2005)). The only technique that requires no timing assumptions whatsoever is randomization, it rather uses a probabilistic approach to solve consensus.

2.1.1 Partial Synchrony

The work by Dwork and Lynch represents the first attempt to devise a time model to circumvent the FLP result (Dwork et al., 1988). Although consensus was proven to be impossible to solve in asynchronous systems subject to process failures, it does not necessarily mean that a fully synchronous system is needed to solve consensus. Their goal was to identify a set of minimal synchrony settings necessary to solve consensus.

The intuition of their work is that while the existence of bounds on communication and processing times is necessary to solve consensus, the knowledge of their exact values is not. They used the concept of partial synchrony in a distributed system which lies between the completely synchronous and completely asynchronous models. There are two cases where the communication (or the computation) can be partially synchronous. One is to assume that there exists an upper bound on communication (or computation) time, but its value is unknown. The other is to consider that exists an upper bound which is known but it only holds after some unknown time.

From these scenarios they devise three models of partial synchrony: partially synchronous communication and synchronous processors, partially synchronous communication and processors, and partially synchronous processors and synchronous communication. For each of these three models, they prove that maximum resilience is possible for four differ-

ent fault models: crash, omission, Byzantine and, authenticated Byzantine (i.e., Byzantine fault model with the processes fully-connected by authenticated channels).

For the crash and omission fault models the resilience is $n \geq 2f + 1$ for n processes of which up to f can be faulty. For the Byzantine and authenticated Byzantine fault models the resilience is $n \geq 3f + 1$.

Moreover, their protocols reach consensus in polynomial time in the number of processes and the upper bound on the communication time. The number of bits exchanged by the protocols is also polynomial in these parameters.

2.1.2 Failure Detectors

The attractiveness of the asynchronous system model has led researchers into developing alternatives to circumvent the FLP result while maintaining most of the system model as fully asynchronous. One of such alternatives is the failure detector model, introduced in (Chandra & Toueg, 1996). It was an important step in order to breed life into the asynchronous system model for consensus.

Their proposal is based on the *unreliable failure detector* abstraction. Since the impossibility to solve consensus in asynchronous systems arises from being unable to distinguish a crashed process from a very slow one, Chandra and Toueg augment the asynchronous model with a failure detection mechanism that can provide this information to processes. It is assumed that these failure detectors can make mistakes (i.e., they are not necessarily perfect), and that process only fail by crashing. Byzantine faults are not tolerated in the initial failure detector model.

In the failure detector model, each process has access to a local failure

detector module which is responsible for detecting and maintaining a list of processes suspected to have crashed. The failure detectors are specified according to two properties: *completeness* and *accuracy*. Completeness requires the failure detector to eventually suspect every process that crashes, and accuracy restricts the mistakes a failure detector can make when suspecting processes. More formally, two types of completeness properties and four types of accuracy properties were defined:

Strong completeness. Eventually every process that crashes is permanently suspected by every correct process.

Weak completeness. Eventually every process that crashes is permanently suspected by some correct process.

Strong accuracy. No process is suspected before it crashes.

Weak accuracy. Some correct process is never suspected.

Eventual strong accuracy. There is a time after which correct processes are not suspected by any correct process.

Eventual weak accuracy. There is a time after which some correct process is never suspected by any correct process.

By combining one of the two completeness properties with one of the four accuracy properties, a class of failure detectors is obtained. This gives a total of eight classes summarized in Table 2.1. Basically, while this model makes possible the use of asynchronous protocols, timing assumptions must still be made to implement the failure detector module.

An example implementation of a failure detector can be one where every process q sends a periodic "q-is-alive" message to all other

	Accuracy			
Completeness	Strong	Weak	Eventually Strong	Eventually Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

Table 2.1: The eight classes of failure detectors (Chandra & Toueg, 1996).

processes. When a process p does not receive this message after a given time, it adds q to a list of suspected processes. If later p receives a "q-is-alive" message, it is because it has erroneously suspected q. Process p can then remove q from the list of suspects and increasing the timeout for the 'q-is-alive' message to avoid making the same mistake in the future.

Although this example fails to implement a failure detector with even the eventual weak accuracy property, in practice it can hold this property for a "long enough" period of time. A "long enough" period of time here means the necessary time for the consensus protocol to reach a decision, or to execute whatever critical step that requires that property.

There were some attempts within the scientific community to extend the failure detection model to include Byzantine failures, but the results were not completely satisfactory (Doudou et al., 2002; Kihlstrom et al., 1997; Malkhi & Reiter, 1997). The problem is that, in the context of Byzantine failures, a complete separation between the failure detector module and the algorithm using it is impossible. This is because Byzantine failures are very specific to the algorithm since they potentially violate its semantics. Unlike crash failures, some action can be considered a Byzantine failure for some algorithm but not for another.

2.1.3 Wormholes

Despite their usefulness, failure detectors are not particularly adequate for systems subject to Byzantine faults. For such a fault to be detected it would be necessary for the failure detector to perfectly understand the semantics of whatever protocols are using its service. It would have to closely monitor every message exchange that occurs in the system and be aware of all the badness that could occur in these message exchanges. Additionally, there can be protocols were an invalid message does not necessarily translate into a malicious action. It would be extremely difficult to distinguish a corrupt process from a correct one that made a honest mistake in such cases.

Wormholes are another way of augmenting the basic system model in order to solve consensus (Correia et al., 2005; Neves et al., 2005; Veríssimo, 2002). For agreement problems, one can say that wormholes pick up where failure detectors left off, providing a useful abstraction for solving these kinds of problems in malicious environments. They are the materialized through architectural hybridization (Veríssimo et al., 2003). That is the notion that certain parts of the system can be enhanced with stronger properties otherwise not guaranteed by the 'normal' environment. In such a setting, the problem of solving consensus can be tackled by having a few of its critical steps executed inside the wormhole which, by design, can be immune to Byzantine faults and/or offer timely behavior.

Neves et al. solved the consensus problem (in its vector variant) with wormholes (Neves et al., 2005). In their model the system is divided in two parts: a payload system which is the normal environment where processes carry out their execution, and a wormhole which is a distributed component with local parts on each node and a private network with enough

synchrony to ensure that its services eventually terminate. While the payload system is subject to Byzantine failures, the wormhole is subject only to crash failures. The wormhole provides a small set of simple services to protocols or applications such as a block agreement protocol.

2.1.4 Randomization: the non-deterministic solution

Randomization is fundamentally different from the other solutions to circumvent the FLP result. Other solutions rely essentially, either implicitly or explicitly, on incorporating timing assumptions into the system model in order to guarantee termination. Those protocols are deterministic: given a certain input, the same output will always be produced at a given step of the protocol. Rather than deterministic, randomized protocols are probabilistic. There are certain steps of the protocol in which the produced result may be a random value, chosen according to a probability distribution. This means that the adversary cannot determine the outcome of any disruptive strategy since that outcome is affected by a random element. The ending result is that any strategy the adversary may employ is dependent on 'luck' in order to prevent correct processes from reaching agreement. The same reasoning applies for correct processes. They also rely on a 'lucky' event to reach a decision. The difference is that in a multi-round protocol, the correct processes only need for that lucky event to happen once, while the adversary needs its lucky event to happen an infinite number of times to prevent correct processes from reaching a decision. If the correct processes are unable to reach agreement in a given round, they just need to carry out the execution for another round. No matter how small the probability for correct process to reach agreement on any individual round may be, given an infinite number of rounds, the

probability that correct processes make a decision is 1.

The biggest advantage of randomization is that the system model requires no additional timing assumptions. The only change is a slight modification to the termination property of consensus. Instead of stating that the correct processes must reach termination, it is stated that the correct processes reach termination with probability 1. This allows the system to remain completely asynchronous and safeguards the protocols to attacks based on the domain of time.

Randomized intrusion-tolerant protocols have been around since Ben-Or's and Rabin's seminal consensus (or Byzantine agreement) protocols (Ben-Or, 1983; Rabin, 1983) (an excellent survey of early work is in (Chor & Dwork, 1989)). 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 to ensure that all processes see the same coins. Ben-Or-style protocols theoretically take many communication steps to terminate. Rabin-style protocols terminate in fewer communication steps but there can be exhaustion of the pre-distributed shares. More recent papers following Rabin's approach have solved this limitation by distributing the coins in run-time (Cachin et al., 2000; Canetti & Rabin, 1993).

Bracha's randomized binary consensus protocol is a Ben-Or-style protocol that exchanges $O(n^3)$ point-to-point messages per round and the expected number of rounds until termination is 2^{n-f} under the *strong adversary* model[1] (Bracha, 1984). The algorithm itself does not use any kind of

[1]In the strong adversary model it is assumed that the adversary completely controls the network scheduling, having the power to decide the timing and the order in which the

cryptographic operations, albeit its dependence on a reliable communication channel implies the use of a relatively inexpensive cryptographic hash function.

The ABBA randomized binary consensus protocol is a Rabin-style protocol that exchanges $O(n^2)$ point-to-point messages per round and reaches a decision in 1 or 2 rounds with high probability (Cachin et al., 2000). The protocol makes extensive use of asymmetric cryptography to ensure the correctness of the execution. For the coin-tossing scheme, the protocol relies on a novel cryptographic technique called a (n, k, f) *dual-threshold signature scheme*. In such a scheme, there are n processes and at most f of them may be corrupt. Processes hold shares of an unpredictable function F that maps the coin name C to a binary value $F(C) \in \{0, 1\}$. The processes can generate shares of the coin and k of those shares are both necessary and sufficient to assemble the function F. The implemented threshold coin-tossing scheme is the Diffie-Hellman based solution in (Cachin et al., 2000).

2.2 Related Implementations

This section describes previous implementations of protocols, and their respective performance evaluation, that offer similar services to the ones provided by the protocol stack of this thesis. Five systems are described: Rampart, SecureRing, BFT, Worm-IT, and SINTRA.

messages are delivered to the processes.

2.2.1 Rampart

Rampart is a protocol stack that provides a set of services to the application programmer for the implementation of high integrity services using state machine replication (Reiter, 1995, 1996a).

All protocols are tolerant to Byzantine faults and achieve optimal resilience. The protocols make extensive use of asymmetric cryptography which dominates the execution time of the protocols. The system model of Rampart makes no assumptions about time, meaning that it is asynchronous. Rampart assumes that each process has an associated private/public key pair. The private key is known only to the associated process, while all public keys are known to all processes. The initial key distribution is done outside the scope of Rampart, but can be achieved, for example, by manual distribution or with the aid of some key distribution infrastructure. An additional assumption is the existence of a reliable, authenticated, point-to-point channel between every pair of processes.

From bottom to top, the stack provides a group membership service, a reliable multicast protocol, and an atomic broadcast protocol. It also provides an output voting service to ensure that the replies received by clients correspond to the correct values.

The group membership protocol provides a service by which processes can request their addition or removal from the group, and ensures that malicious processes cannot force unwanted changes to the group. The membership protocol, in its essence, solves a consensus problem since all correct processes must agree on a view of the group. In Rampart, this means that its termination cannot be guaranteed because it assumes an asynchronous model.

The reliable multicast primitive is implemented on top of the mem-

bership protocol. Rampart uses this primitive to either deliver messages reliably multicasted by other group members, or to deliver group views received locally from the membership protocol. The progression of this protocol is dependent on the liveness of the membership protocol. If a failed process is not removed from group, the reliable multicast protocol can become halted until the group view is correctly updated.

The atomic multicast primitive is implemented on top of the reliable multicast primitive. This primitive is used to either deliver group views or messages in the same order to all group members. The order of the messages are decided by a designated process called the sequencer. Clients outside the group can atomically multicast messages by sending them to any of the group members which, in turn, forwards the request by atomically multicasting it to the group.

The output voting service exists in two flavors. The first is a simple voting protocol where each server sends its output to the client, which performs the voting based on the received values. In the second method, the voting is performed at the servers, which authenticate the response with a (k,n)-threshold signature scheme. The client then verifies the threshold signature associated with the received value in order to accept the response.

The performance evaluation of the protocols in the Rampart Toolkit is scattered across several papers (Reiter, 1994, 1995, 1996b). Although they do not provide much information about the testbed used for the experiments, it is assumed that they were conducted under roughly the same environmental settings: moderately loaded SPARCstations 10s running SunOS 4.1.3 and spanning several networks.

The group membership protocol is based on the concept of a manager.

A process that is responsible, based on recommendations from the other group members, for suggesting and carrying on changes to the group. There are two cases on this type of protocol, one where the manager is correct, and another where the manager is faulty. For the faulty case, the protocol has an added complexity inherent to the cost of electing another process responsible for carrying out the necessary group changes. In the Rampart Toolkit, this process is called the deputy. The performance assessment evaluates both cases using RSA for the digital signature scheme, with 512-bit keys.

For the case where the manager is correct, the results present the average time necessary to remove a non-manager process from the group. For the non-manager processes, the results were 215 ms, 250 ms, and 283 ms for a group size of 4, 7, and 10 processes, respectively. For the manager process the results were 205 ms, 236 ms, and 265 ms for a group size of 4, 7, and 10 processes, respectively.

For the case where the manager is faulty, the results show the average time to remove the manager from the group, which includes electing a deputy and having it carrying out the group change. For the non-deputy processes the results were, respectively, 295 ms, 330 ms, and 378 ms for a group size of 4, 7, and 10 processes. For the deputy, the results were 282 ms, 320 ms, and 360 ms for a group size of 4, 7, and 10 processes, respectively.

The reliable and atomic multicast protocols are evaluated in (Reiter, 1994). The chosen RSA key size for these experiments was 300 bits. For the reliable multicast, the average latency was measured with varying group sizes and different message payload sizes. For a 0 KB payload, the results were 42.5 ms, 55 ms, and 67.5 ms for a group size of 4, 7, and 10 processes,

respectively. For 1 KB payloads, the results were, respectively, 45 ms, 58 ms, and 68 ms for 4, 7, and 10 processes. For 4 KB payloads, the results were 51 ms, 67.5 ms, and 85 ms, for 4, 7, and 10 processes, respectively. The performance results also show the sustainable throughput for the reliable multicast protocol with 0 KB messages and two cases: only one process multicasting and all processes multicasting. With one process multicasting the results were, for a group size of 4, 7, and 10 processes, 27 msgs/s, 20 msgs/s, and 16 msgs/s, respectively. With all the processes multicasting, the results were 26 msgs/s, 18 msgs/s, and 14 msgs/s for a group size of 4, 7, and 10 processes, respectively.

Finally, for the atomic multicast protocol the available results show the sustainable throughput for 0 KB message payloads and two multicasting scenarios: one process multicasting, and all processes multicasting. For the scenario were only one process multicasts, the results were, respectively, 22 msgs/s, 18 msgs/s, and 16 msgs/s for 4, 7, and 10 processes. For the second scenario, the results were 18 msgs/s, 14 msgs/s, and 11 msgs/s for 4, 7, and 10 processes, respectively.

2.2.2 SecureRing

SecureRing provides a set of intrusion-tolerant protocols that offer services such as reliable ordered message delivery and group membership (Kihlstrom et al., 1998, 2001).

The philosophy of SecureRing is to optimize the protocols for fault-free executions, and assume a performance degradation whenever Byzantine faults are detected (which are assumed to be rare). The protocols are built upon an asynchronous system model. Processes are assumed to be fully connected, and the communication channels to be unreliable and unau-

thenticated. Every process has a private/public key pair and has access to the public key of other processes. The resilience of the protocols is optimal in the sense that they can tolerate up to f faulty processes out a total of $n = 3f + 1$ processes.

The protocol stack provided by SecureRing is organized as follows. At the top of the stack is the Reliable Totally Ordered Message Delivery Protocol. This protocol in essence guarantees the same properties of atomic broadcast and it is built on top of a group membership protocol, and a Byzantine fault detector (which is also used by the membership protocol). The protocol relies on a logical token-passing ring arrangement of the processes. The total order of messages is derived from a sequence number present in the token. The membership protocol receives information from the Byzantine fault detector about process failures and installs new membership views accordingly.

The Byzantine failure detector provides the strong Byzantine completeness and the eventual strong accuracy properties. It does so by assuming the existence of periods of stability in the system with bounds on computation and communication speeds, and that the digital signatures are unforgeable. Besides detecting process crashes, the fault detector also detects possible malicious actions such as the transmission of messages with invalid signatures.

At the bottom of the SecureRing protocol stack is a Message Diffusion Protocol. This protocol is used by both the membership protocol and the fault detector, and in some instances by the Reliable Totally Ordered Message Delivery protocol. In the latter, it is used only when process failures occur during the recovery phase of the protocol when a new view is installed. The Message Diffusion protocol is essentially a reliable broadcast

protocol and ensures that, upon a broadcast, every correct process receives the same message or no message at all.

The performance tests on SecureRing were performed on a testbed consisting of eight UltraSparc 2 workstations with clock speeds varying from 168 MHz to 200 MHz. The Operating System used was Solaris, and the machines were connected by a 100 Mbps Ethernet.

For the total order delivery protocol, the measurements were taken by varying three parameters: the message size, the RSA key size, and the number of processes. The throughput values were more or less the same independently of the number of processes, showing only a slight decline on certain occasions with 7 and 8 processes. To make the discussion concise, we only discuss the values obtained for a 4 process group size. Using 300-bit keys the measured thoughput was 5500 msgs/s for 200-byte messages, 1900 msgs/s for 600-byte messages, and 1000 msgs/s for 1000-byte messages. For 512-bit keys, the results were 2700 msgs/s for 1000-byte messages, 900 msgs/s for 600-byte messages, and 500 msgs/s for 1000-byte messages. Finally, for 768-bit keys, the results were 1080 msgs/s for 200-byte messages, 350 msgs/s for 600-byte messages, and 200 msgs/s for 1000-byte messages.

For the membership protocol, experiments were made with the goal of measuring average time to remove a process from the group. The varying parameters were the number of processes and the RSA key size. For 300-bit keys, the average removal time was 50 ms for 4 processes, and 100 ms for 8 processes. For 512-bit keys, the removal time was 140 ms for 4 processes, and 390 ms for 8 processes. Finally, for 768-bit keys, the time was 400 ms for 4 processes, and 980 ms for 8 processes.

2.2.3 BFT

Unlike the previous described implementations, BFT is not a protocol stack, but an algorithm that provides Byzantine-fault-tolerant state machine replication (Castro & Liskov, 1999).

BFT has a set of characteristics that make it an efficient solution for the implementation of Byzantine-fault-tolerant services using state machine replication. It has optimal resilience, tolerating up to $\lfloor \frac{n-1}{3} \rfloor$ faulty processes out of a total of n processes. During fault-free operation, BFT does not resort to public-key cryptography for its normal execution, it relies instead on message authentication codes. BFT does not rely on synchrony for safety, but it requires it for liveness. The required assumption is that the network delays do not grow exponentially.

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. The servers are either primary or backup. There is only one primary at any given moment in the system. The client requests are issued directly to the primary, which in turn multicasts the request to the backups. The replies are transmitted to the client by all servers. The 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 the servers must agree on a new primary. View changes are triggered by timeouts. A consensus algorithm is run to elect a new primary. This is the only time that BFT resorts to public-key cryptography. After a view change the service resumes to its normal operation.

The performance experiments on BFT were carried on a testbed consisting of four DEC 3000/400 Alpha workstations with a 133 MHz Alpha

21064 processor and 128 MB of RAM, running Digital Unix 4.0. The machines were connected by a 10 Mbps DEC EtherWorks 8T/8X Ethernet switch. Two main experiments were performed. A micro-benchmark that measures the latency to invoke a null operation, and an Andrew benchmark that emulates a software development workload and measures a replicated file system that uses BFT.

The micro-benchmark evaluates a simple request-reply service and compares one that is replicated using BFT with another that is unreplicated and uses UDP. The results are presented for different payload sizes for the request and reply messages. For the replicated service there is also a distinction between read-write and read-only operations. The read-only operations do not require total order since they do not modify the system state. For 0 KB payload for both the request and reply messages the results were 3.35 ms for the replicated read-write operation, 1.62 ms for the replicated read-only, and 0.82 ms for the unreplicated operation. For 4 KB requests and 0 KB replies, the results were 14.19 ms, 6.98 ms, and 4.62 ms, for the replicated read-write, replicated read-only, and the unreplicated operation, respectively. Finally, for 0 KB requests and 4 KB replies, the results were 8.01 ms for the replicated read-write operation, 5.94 ms for the replicated read-only, and 4.66 ms for the unreplicated operation.

The Andrew benchmark was used to evaluate BFT by comparing the performance of the standard NFS file system with a replicated NFS version using BFT. On average the BFT-replicated NFS completed the benchmark in 64.48 seconds, while the standard NFS took 62.52. The implies that the replicated version only took an additional 3% of time to complete the benchmark.

2.2.4 Worm-IT

Worm-IT is a group communication system that uses the wormhole abstraction to provide a membership service and a view-synchronous atomic multicast (VSAM) primitive (Correia et al., 2006a).

Worm-IT is designed under an hybrid system model where a small subset of the system is assumed to be secure (i.e., not subject to Byzantine failures) and synchronous, whereas the rest is assumed to be unreliable in the sense that completely asynchronous and subject to all kinds of failures. The secure part of the system (or the *wormhole*) is implemented by a special secure distributed component called *Trusted Timely Computing Base* (TTCB) (Correia et al., 2002). Critical steps of the protocols that require stronger environmental properties (such as agreement tasks) can be executed inside the *wormhole*. As such, the TTCB provides a small number of services useful for protocols and applications, most notably, it provides a *Trusted Block Agreement Service* (TBA) which allows processes to agree on binary blocks of data with a limited size.

This architecture allows the protocols provided by Worm-IT to be efficient since they do not require any kind of public-key cryptography, and to be totally decentralized which makes the protocols more resilient to denial-of-service attacks because there is no notion of primary replica.

A performance evaluation of Worm-IT was conducted on a LAN environment with 6 Pentium III PCs. The PCs were connected by two 100 Mbps Ethernet switches. One switch was used for the payload network, where most of the protocol execution took place, and another for the control network, where the wormhole (i.e., TTCB) was deployed.

The experiments for the membership service were performed with a number of nodes ranging from four to six, with f always set to 1. Three

scenarios were measured: the time for a crashed process to be removed from the group, the time for a process to join, and the time for a process to leave. For the first scenario, the measurements registered approximately 20 ms to remove a crashed process independent of the group size. In the second scenario, the time to join the group was around 9 ms for both 4 and 5 processes, with the latter taking slightly longer. For the leave operation, the time was around 10 ms for 4, 5, and 6 processes.

For the VSAM primitive, experiments were performed by varying three parameters: the delivery watermark, the message size, and the number of processes. This delivery watermark (WM) is the number of delivered messages that cause an agreement round to start in order to decide the ordering of the messages received so far. The message size is in respect to the payload and disregards the headers. The number of processes is the total number of nodes participating in the protocol. The metrics used for these experiments were the average latency per message

For the first experiment of the VSAM protocol the number of processes was fixed to four and the message payload to 100 bytes. There was only one sender, and none of the processes failed. The WM was varied between 1 and 25. The observed average latency for WM=1 was around 12 ms and increased linearly up to around 80 ms for WM=25. With increasing WM values, the throughput stabilized at around 175 messages per second.

The second experiment measured the impact of the payload message size. The number of processes was fixed to four, and the watermark to WM=10. There was a single sender and no process failures were considered. The payload size was varied between 0 and 1000 bytes. Regardless of the message size the performance results were always the same. The average latency was around 37.5 ms and the throughput around 170 mes-

sages per second.

The last experiment measured the protocol performance with a varied number of processes in different situations: (1) one sender and no failures, (2) all senders and no failures, and (3) one sender and one crashed process. For the average latency, the results were the same for scenarios (1) and (3): around 37.5 ms regardless of the number of processes. For scenario (2), the results were 25 ms for 4 processes, and roughly 45 ms for 5 processes, and 50 ms for 6 processes. For the throughput, the results were again similar for (1) and (3): 175 msgs/s. For (2), it was around 360 msgs/s for 4 processes, 400 msgs/s for 5 processes, and 415 msgs/s for 6 processes.

2.2.5 SINTRA

SINTRA is a protocol stack that provides a number of communication primitives for the construction of intrusion-tolerant distributed services (Cachin & Poritz, 2002). SINTRA uses a completely asynchronous system model, not relying on any time assumptions for either liveness or safety. It uses a static group, and attains optimal resilience in the presence of arbitrary faults. To solve consensus in such a setting, SINTRA uses randomization. The protocols in SINTRA also rely heavily in public-key cryptography, more specifically in threshold signatures and a threshold coin-tossing scheme, which has a significant negative impact on their performance.

At the bottommost of the SINTRA protocol stack there are two broadcast primitives, reliable and consistent broadcast, and a binary agreement protocol. Above this layer, there are three other layers, each one implementing one protocol, from bottom to top: multi-valued agreement, atomic broadcast, and secure causal atomic broadcast.

The reliable broadcast primitive is the protocol proposed in (Bracha & Toueg, 1985) and ensures that all correct processes deliver the same message or no message at all. The consistent broadcast is the one proposed by Reiter for Rampart. It ensures that only those processes that deliver the message, do it so for the same value.

The binary agreement protocol of SINTRA is the ABBA protocol described above and is, in fact, the only one that uses randomization in the stack (Cachin et al., 2000). The multi-valued agreement protocol is built on top of the binary agreement and allows processes to agree on values with an arbitrary size.

The atomic broadcast primitive ensures that messages are delivered by total order to all correct processes, while the secure causal atomic broadcast extends this primitive with confidentiality for the payload of the messages until their order of delivery is determined.

Performance experiments were conducted in SINTRA in both LAN and WAN environments. The LAN setup consisted of four servers running different operating systems connected by a 100 Mbps Ethernet switch. The WAN setup consisted of four Linux servers deployed on the Internet, each located in a different continent. The average network latency between these hosts on the Internet varied around 100 to 400 ms.

The first experiment evaluated the Atomic Broadcast protocol. The way the experiment was conducted was with three of the servers broadcasting a total of 1000 messages concurrently. The performance metric used was the time between successive deliveries. The protocol proceeded in rounds and batched two messages per round. The results for the LAN setup were around 0.5-1 seconds between batches. For the WAN setup, the results were around 2.5-3.5 seconds between batches.

The second experiment evaluated the latency of the broadcast protocols. The experiment was conducted with one server sending 500 messages. The performance metric was once again the time between successive deliveries. For the LAN setup, the average results were 0.11s for the consistent broadcast, 0.13s for the reliable broadcast, 1.07s for the secure causal broadcast, and 0.69s for the atomic broadcast. For the WAN setup, the average results were 0.83s for the consistent broadcast, 0.72s for the reliable broadcast, 3.61s for the secure causal broadcast, and 2.95s for the atomic broadcast. An additional setup was used in this experiment were the four servers from the LAN were combined with three others from the WAN to form a group of seven nodes. The results for this setup were 0.64s for the consistent broadcast, 0.6s for the reliable broadcast, 3.79s for the secure causal broadcast, and 2.74s for the atomic broadcast.

Chapter 3

The Protocol Stack

This chapter describes the protocol stack that was implemented in the context of this thesis. It is divided in two main sections. First, it lays out the system model assumed by the protocol stack. Second, it describes in detail each protocol in the stack, its function, algorithm, and formal correctness proof (where relevant).

3.1 System Model

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

There are no constrains 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 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 ei-

ther 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 for total number of n processes.

The system is assumed to be completely asynchronous. There are no assumptions whatsoever about bounds on processing times or communications delays.

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 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*.

All the described protocols preserve their correctness under the presence of an adversary with complete control of the network scheduling, having the power to decide the timing and the order by which the messages are delivered to the processes. Despite this, the presence of such an adversary is not very realistic in practice since a malicious attacker who has the power to control the network scheduling usually has the power to perform much more severe damage such as halting the communication between the processes altogether.

3.2 Protocol Stack

The RITAS protocol stack, depicted in Figure 3.1, provides a set of useful distributed system services. A first version of this protocol stack was presented in (Correia et al., 2006b). All protocols in the stack rely on two standard Internet services: the IPSec Authentication Header protocol (AH) and the Transmission Control Protocol (TCP). These two protocols provide authenticated reliable communication channels for the rest of the stack. At the bottom there are the broadcast primitives: echo broadcast and reliable broadcast. On top of the broadcast primitives is the most basic flavor of consensus: binary consensus. This is the only randomized protocol in the stack. On top of binary consensus there is the multi-valued consensus protocol, which allows the proposal of arbitrary values. Finally, at the top of the stack there are two protocols: vector consensus, and atomic broadcast. Each one of these protocols is throughly described in the next subsections.

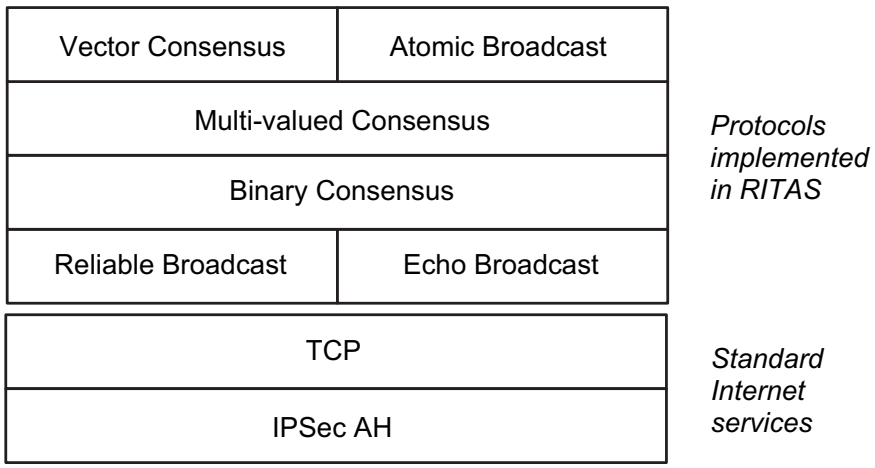

Figure 3.1: The RITAS protocol stack.

3.2.1 Reliable Channels

The two layers at the bottom of the stack implement a reliable channel (see Figure 3.1). 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.

Formally, such a channel follows two properties:

RC1 Reliability. If processes p_i and p_j are correct and p_i sends a message m to p_j, then p_j eventually receives m.

RC2 Integrity. If p_i and p_j are correct and p_j receives a message m with $sender(m) = p_i$, then m was sent by p_i and m was not modified in the channel.[1]

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 (Kent & Atkinson, 1998).

TCP establishes a point-to-point two-way communication channel between a pair of processes and guarantees reliable and FIFO delivery of sender to receiver data.

The IPSec AH protocol guarantees connectionless integrity and data origin authentication of IP datagrams. It protects all fields of an IP datagram except those that are mutable during the transmission of an IP packet on the network (e.g., the TTL field). The IPSec AH protocol requires that every pair of processes p_i and p_j share a secret symmetric key k_{ij}, which is already assumed in the system model.

[1] The predicate $sender(m)$ returns the process identifier of the sender of message m.

This layer provides a primitive *RC_Broadcast(m)* that allows the broadcasting of a message m to all processes. In practice this is done by sending m to each channel that connects to any other process.

3.2.2 Reliable Broadcast

The *reliable broadcast* protocol ensures that all correct processes eventually receive the same set of messages. No constrains are placed on the relative delivery order of messages.

It is defined formally as follows:

RB1 Validity. If a correct process p_i broadcasts a message m, then p_i eventually delivers m.

RB2 Agreement. If a correct process p_i delivers a message m, then all correct processes eventually deliver m.

RB3 Integrity. For any message m, every correct process delivers m at most once, and only if m was previously broadcasted by $sender(m)$.

These properties basically ensure that all correct processes deliver the same messages, and that, upon a broadcast, if the sender is correct, then the message is eventually delivered by all correct processes. In the case the sender is corrupt, the protocol guarantees that either all correct processes deliver the same message, or no message is delivered at all.

The implemented reliable broadcast protocol was originally proposed in (Bracha, 1984), and it is presented in Algorithm 1. An instance of the protocol, identified by *rbid*, starts with the sender broadcasting a message (INITIAL, m, rbid) to all processes. Upon receiving this message a process sends a (ECHO, m, rbid) message to all processes. It then waits for at least

$\lfloor \frac{n+f}{2} \rfloor + 1$ (ECHO, m, rbid) messages or $f + 1$ (READY, m, rbid) messages, and then it transmits a (READY, m, rbid) message to all processes. Finally, a process waits for $2f + 1$ (READY, m, rbid) messages to deliver m. Figure 3.2 illustrates the three communication steps of the protocol. The broadcasts inside the protocol are made via the reliable channels.

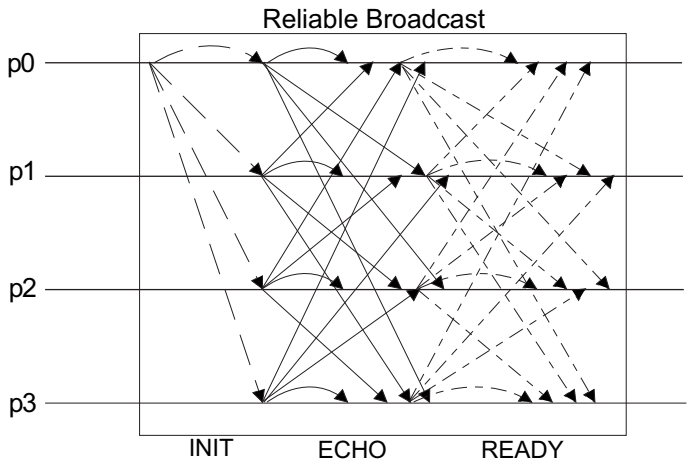

Figure 3.2: Messages exchanged during a reliable broadcast execution with four processes.

Correctness Proof

The protocol is the one described in (Bracha, 1984) so the correctness proof remains unchanged.

3.2.3 Echo Broadcast

The *echo broadcast* protocol is a weaker and more efficient version of 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 (Toueg, 1984). In this case, the protocol only ensures that

Algorithm 1 Reliable Broadcast protocol (for process p_i).

Function `R_Broadcast` (v_i, rbid)

INITIALIZATION:

1: **activate task** (T0); {sender only}
2: **activate task** (T1);

TASK T0 (SENDER ONLY):

1: `RC_Broadcast` (⟨INITIAL, v_i, rbid⟩);

TASK T1:

1: **wait until** have been delivered at least one ⟨INITIAL, v, rbid⟩ or $\frac{n+f}{2}$ ⟨ECHO, v, rbid⟩ or $f+1$ ⟨READY, v, rbid⟩) messages;
2: `RC_Broadcast` (⟨ECHO, v, rbid⟩);
3: **wait until** have been delivered at least $\frac{n+f}{2}$ ⟨ECHO, v, rbid⟩ or $f+1$ ⟨READY, v, rbid⟩ messages;
4: `RC_Broadcast` (⟨READY, v, rbid⟩);
5: **wait until** have been delivered at least $2f+1$ ⟨READY, v, rbid⟩ messages;
6: **return** v;

the subset of correct processes that deliver will do it for the same message.

Formally, we define *echo broadcast* with the following properties:

EB1 Validity. If a correct process p_i broadcasts a message m, then p_i eventually delivers m.

EB2 Agreement 1. If the sender is correct and a correct process p_i delivers a message m, then all correct processes eventually deliver m.

EB3 Agreement 2. If the sender is corrupt and a correct process p_i delivers a message m, then no correct process delivers $m' \neq m$.

EB4 Integrity. For any message m, every correct process delivers m at most once, and only if m was previously broadcasted by $sender(m)$.

The implemented *echo broadcast* primitive was originally proposed in (Toueg, 1984), and is a variant of the previously described *reliable broadcast* protocol. It is presented in Algorithm 2.

The protocol is essentially the described *reliable broadcast* algorithm with the last communication step omitted. An instance of the protocol identified by *ebid* 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. Figure 3.3 illustrates the communication steps of the protocol. The broadcasts inside the protocol are made using the reliable channels.

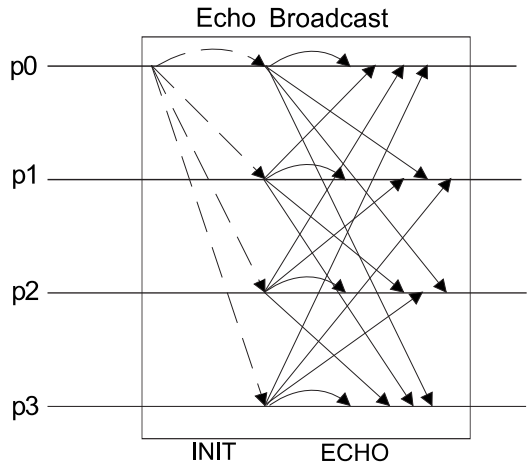

Figure 3.3: Messages exchanged during an echo broadcast execution with four processes.

Correctness Proof

The protocol is the one described in (Toueg, 1984) so the correctness proof remains unchanged.

Algorithm 2 Echo Broadcast protocol (for process p_i).

Function `E_Broadcast` (v_i, ebid)
INITIALIZATION:
1: **activate task** (T0); {sender only}
2: **activate task** (T1);

TASK T0 (SENDER ONLY):
1: `RC_Broadcast` (⟨INITIAL, v_i, rbid⟩);

TASK T1:
1: **wait until** have been delivered at least one ⟨INITIAL, v, rbid⟩ or $\frac{n+f}{2}$ ⟨ECHO, v, rbid⟩
2: `RC_Broadcast` (⟨ECHO, v, rbid⟩);
3: **wait until** have been delivered at least $2f + 1$ ⟨ECHO, v, rbid⟩
4: **return** v;

3.2.4 Binary Consensus

A *binary consensus* allows correct processes to agree on a binary value. 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.

Binary consensus is formally defined by the following properties:

BC1 Validity. If all correct processes propose the same value b, then any correct process that decides, decides b.

BC2 Agreement. No two correct processes decide differently.

BC3a Termination. Every correct process eventually decides.

Given the FLP impossibility result, there is no deterministic algorithm that can guarantee the termination property of consensus in our system model, which is completely asynchronous. The solution is to resort to a randomized model that guarantees the termination in a probabilistic way

(as opposed to a deterministic way). As such, the termination property is changed to the following:

BC3 Termination. Every correct process eventually decides with probability 1.

The implemented protocol is adapted from a randomized algorithm previously presented in (Bracha, 1984). 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 main advantage of this algorithm is that is does not use any cryptography whatsoever (altough its dependence on a reliable communication channel, in practice, implies the use of a relatively cheap cryptographic hash function of some sort).

The protocol, which is presented in Algorithm 3, proceeds in 3-step rounds, running as many rounds as necessary for a decision to be reached. The first step (lines 2-9) of an execution of the protocol identified by *bcid* starts when each process p_i (reliably) broadcasts its proposal v_i. Then waits for $n - f$ *valid* messages and changes v_i to reflect the majority of the received values. In the second step (lines 10-17), 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 (lines 18-27), 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.

The validation of the messages is performed as follows. 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.

Correctness Proof

The protocol is the one described in (Bracha, 1984) so the correctness proof remains unchanged.

3.2.5 Multi-valued Consensus

The *multi-valued consensus* builds on top of the *binary consensus* protocol. It allows for processes to propose and decide on values with an arbitrary domain $\mathcal{V}$. Depending on the proposals, the decision is either one of the proposed values or a default value $\perp \notin \mathcal{V}$.

Formally, it is defined as follows:

MVC1 Validity 1. If all correct processes propose the same value v, then

Algorithm 3 Binary Consensus protocol (for process p_i).

Function `B_Consensus` (v_i, bcid)

1: **repeat**
2: `R_Broadcast` (⟨S1, v_i, bcid, i⟩);
3: **wait until** ($(n-f)$ *valid* S1 messages have been delivered);
4: $\forall_j$: **if** (⟨S1, v_j, bcid, j⟩ has been delivered) **then** $V_i[j] \leftarrow v_j$; **else** $V_i[j] \leftarrow \perp$;
5: **if** $(\#_1(V_i) \geq \lceil \frac{n-f}{2} \rceil)$ **then**
6: $v_i \leftarrow 1$;
7: **else**
8: $v_i \leftarrow 0$;
9: **end if**
10: `R_Broadcast` (⟨S2, v_i, bcid, i⟩);
11: **wait until** ($(n-f)$ *valid* S2 messages have been delivered);
12: $\forall_j$: **if** (⟨S2, v_j, bcid, j⟩ has been delivered) **then** $V_i[j] \leftarrow v_j$; **else** $V_i[j] \leftarrow \perp$;
13: **if** $(\exists_v : v \neq \perp$ and $\#_v(V_i) > \frac{n}{2})$ **then**
14: $v_i \leftarrow v$;
15: **else**
16: $v_i \leftarrow \perp$;
17: **end if**
18: `R_Broadcast` (⟨S3, v_i, bcid, i⟩);
19: **wait until** ($(n-f)$ *valid* S3 messages have been delivered);
20: $\forall_j$: **if** (⟨S3, v_j, bcid, j⟩ has been delivered) **then** $V_i[j] \leftarrow v_j$; **else** $V_i[j] \leftarrow \perp$;
21: **if** $(\exists_v : \#_v(V_i) > 2f+1)$ **then**
22: **return** v;
23: **else if** $(\exists_v : \#_v(V_i) > f+1)$ **then**
24: $v_i \leftarrow v$;
25: **else**
26: $v_i \leftarrow 1$ or 0 with probability $\frac{1}{2}$;
27: **end if**
28: **until**

any correct process that decides, decides v.

MVC2 Validity 2. If a correct process decides v, then v was proposed by some process or $v = \perp$.

MVC3 Validity 3. If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.

MVC4 Agreement. No two correct processes decide differently.

MVC5 Termination. Every correct process eventually decides.

The implemented protocol is adapted from the multi-valued consensus proposed in (Correia et al., 2006b). It uses the services of the underlying *reliable broadcast*, *echo broadcast*, and *binary consensus* layers. The main difference from the original protocol is 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. These changes grant greater efficiency to the protocol without compromising its correctness. The protocol is presented in Algorithm 4.

The protocol starts when every process p_i announces its proposal value v_i by reliably broadcasting a (INIT, v_i) message (line 1). The processes then wait for the reception of $n - f$ INIT messages and store the received values in a vector V_i (lines 2-3). 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 (lines 4-9). The next step is to wait for the reception of $n - f$ *valid* VECT messages (line 10). 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 (lines 11-16). 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) with the same value v and then it decides on that value (lines 17-22).

Algorithm 4 Multi-valued Consensus protocol (for process p_i).

Function `M_V_Consensus` (v_i, cid)

1: `R_Broadcast` ($\langle$INIT, v_i, cid, i$\rangle$);
2: **wait until** (at least $(n - f)$ INIT messages have been delivered);
3: $\forall_j$: **if** ($\langle$INIT, v_j, cid, j$\rangle$ has been delivered) **then** V_i[j] $\leftarrow v_j$; **else** V_i[j] $\leftarrow \perp$;
4: **if** ($\exists_v^1 : \#_v(V_i) \geq (n - 2f)$) **then**
5: $w_i \leftarrow v$;
6: **else**
7: $w_i \leftarrow \perp$;
8: **end if**
9: `E_Broadcast` ($\langle$VECT, w_i, V_i, cid, i$\rangle$);
10: **wait until** (at least $(n-f)$ *valid* messages $\langle$VECT, w_j, V_j, cid, j$\rangle$ have been delivered);
11: $\forall_j$: **if** ($\langle$VECT, w_j, V_j, cid, j$\rangle$ has been delivered) **then** W_i[j] $\leftarrow w_j$; **else** W_i[j] $\leftarrow \perp$;
12: **if** ($\forall_{j,k}$ W_i[j] $\neq$ W_i[k] $\Rightarrow$ W_i[j] $= \perp$ or W_i[k] $= \perp$) and ($\exists_w$: $\#_w(W_i) \geq (n - 2f)$) **then**
13: $b_i \leftarrow 1$;
14: **else**
15: $b_i \leftarrow 0$;
16: **end if**
17: $c_i \leftarrow$ `B_Consensus`(b_i, cid);
18: **if** ($c_i = 0$) **then**
19: **return** $\perp$;
20: **end if**
21: **wait until** (at least $(n - 2f)$ *valid* messages $\langle$VECT, v_j, V_j, cid, j$\rangle$ with $v_j = v$ have been delivered);
22: **return** v;

Correctness Proof

Lemma 1 *If a message (VECT, w_i, V_i, cid, i) is echo-broadcasted by a correct process p_i, then eventually all correct processes will consider it valid.*

Proof: This Lemma is similar to the Lemma 1 found in (Correia et al., 2006b). The difference is that here the VECT message is echo-broadcasted instead of reliably broadcasted. This does not affect the correctness of the lemma since it is assumed that the VECT message is broadcasted by a correct process, and in this case the echo broadcast protocol ensures the same agreement property as the reliable broadcast protocol. □

Theorem 1 *(Validity 1) If all correct processes propose the same value v, then any correct process that decides, decides v.*

Proof: The proof is similar to the proof of the original protocol (Correia et al., 2006b). □

Theorem 2 *(Validity 2) If a correct process decides v, then v was proposed by some process or $v = \perp$.*

Proof: The proof is obtained with a trivial inspection of the protocol. □

Theorem 3 *(Validity 3) If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.*

Proof: The proof is similar to the proof of the original protocol (Correia et al., 2006b). □

Theorem 4 *(Agreement) No two correct processes decide differently.*

Proof: The proof is similar to the proof of the original protocol (Correia et al., 2006b). □

Theorem 5 *(Termination) Every correct process eventually decides.*

Proof: The difference from this proof to the proof for the original protocol is that we must prove that the protocol makes progress at the execution of the echo broadcast protocol (lines 9-10). Termination of the echo broadcast protocol is guaranteed if the sender is correct. The protocol must deliver $n - f$ VECT messages to make progress, and since by definition, there are at least $n - f$ correct processes, then at least $n - f$ VECT messages were sent by a correct process which implies that progress is ensured. □

3.2.6 Vector Consensus

Vector consensus allows processes to agree on a vector with a subset of the proposed values. It ensures that every correct process decides on the same vector V of size n; if a process p_i is correct, then the vector element $V[i]$ is either the value proposed by p_i or the default value $\perp$, and at least $f + 1$ elements of V were proposed by correct processes.

This problem is adapted from the problem of *interactive consistency*, defined for synchronous systems, to asynchronous systems (Pease et al., 1980). While in interactive consistency the problem requires that the decision vector is composed by the values proposed by all correct processes, in vector consensus the requirement is that the decision vector is formed by a majority of values proposed by correct processes.

Vector consensus is formally defined by the following properties:

VC1 Vector Validity. Every correct process that decides, decides on a vector V of size n:

- $\forall p_i$: if p_i is correct, then either $V[i]$ is the value proposed by p_i or $\perp$.

- at least $(f+1)$ elements of V were proposed by correct processes.

VC2 Agreement. No two correct processes decide differently.

VC3 Termination. Every correct process eventually decides.

The implemented protocol is the one described in (Correia et al., 2006b), which uses *reliable broadcast* and *multi-valued consensus* as underlying primitives. The protocol, which is presented in Algorithm 5, 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 proceeds 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.

Algorithm 5 Vector Consensus protocol (for process p_i).

Function `Vector_Consensus` (v_i, vcid)

1: $r_i \leftarrow 0$; {round number}
2: `R_Broadcast` ($\langle$VC_INIT, v_i, vcid, i$\rangle$);
3: **repeat**
4: **wait until** (at least $(n - f + r_i)$ VC_INIT messages have been delivered);
5: $\forall_j$: **if** ($\langle$VC_INIT, v_j, vcid, j$\rangle$ has been delivered) **then** W_i[j] $\leftarrow v_j$; **else** W_i[j] $\leftarrow \perp$;
6: $V_i \leftarrow$ `M_V_Consensus` (W_i, (vcid,r_i));
7: $r_i \leftarrow r_i + 1$;
8: **until** ($V_i \neq \perp$);
9: **return** V_i;

Correctness Proof

The protocol is the one described in (Correia et al., 2006b) and the correctness proofs remain unchanged.

3.2.7 Atomic Broadcast

The *atomic broadcast* protocol delivers messages in the same order to all processes and it is on the genesis of many important distributed system services. One can see atomic broadcast as a reliable broadcast protocol plus the total order property.

Formally, atomic broadcast is defined by the following set of properties:

AB1 Validity. If a correct process broadcasts a message m, then some correct process eventually delivers m.

AB2 Agreement. If a correct process delivers a message m, then all correct processes eventually deliver m.

AB3 Integrity. For any identifier ID, every correct process p delivers at most one message m with identifier ID, and if $sender(m)$ is correct then m was previously broadcasted by $sender(m)$.

AB4 Total order. If two correct processes deliver two messages m_1 and m_2, then both processes deliver the two messages in the same order.

The implemented protocol was adapted from a proposal in (Correia et al., 2006b). The main difference from the original protocol 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, presented in Algorithm 6, is conceptually divided in two tasks: (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 (A_MSG, i, $rbid$, m) message where $rbid$ is a local identifier for the message (lines 7-8). 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 A_MSG messages to arrive. When such a message arrives, p_i constructs a vector V_i with the identifiers of the received A_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 (lines 10-11). 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 (lines 12-13). 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 (lines 14-16). Figure 3.4 illustrates the protocols involved in the agreement task.

The protocol applies a window of messages to be delivered. Its purpose is to impose a limit on the identifiers that can be proposed to the multi-valued consensus primitive (line 14). This serves to ensure that processes will not indefinitely propose more identifiers while the messages with the identifiers within the window are not delivered by the

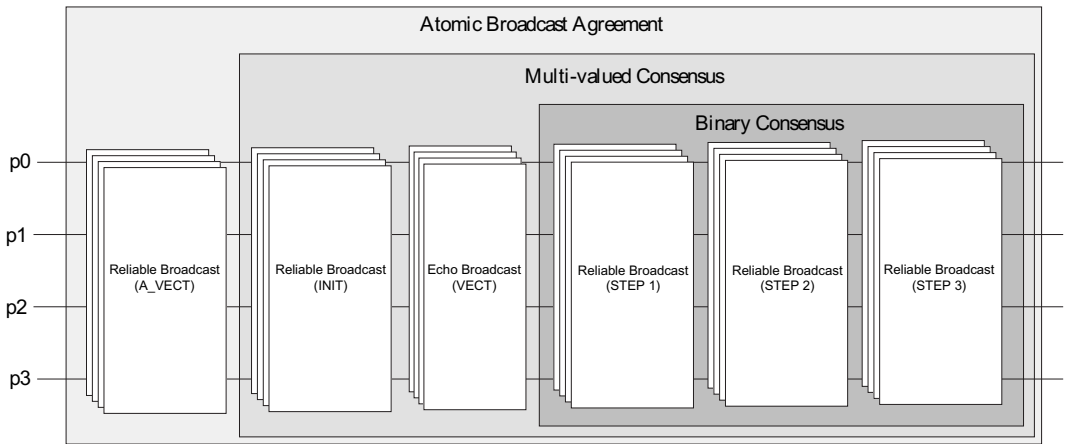

Figure 3.4: Protocols involved in an agreement task of the atomic broadcast protocol with four processes.

atomic broadcast protocol. The variable B_j indicates the beginning of the window for process j and L is the window size. So, for example, if $B_j = 10$ and $L = 50$, task T2 will only consider reaching agreement on the order of messages whose identifier (j, num) has $10 \leq num < 50$.

Correctness Proof

Theorem 6 *(Validity) If a correct process broadcasts a message m, then some correct process eventually delivers m.*

Proof: A correct process broadcasts a message M by calling A_Broadcast(M). Then, the atomic broadcast protocol adds a header to the message and broadcasts it using the reliable broadcast protocol. The properties of the reliable broadcast protocol ensure that all correct processes eventually receive M (properties RB1-RB3). This implies that all correct processes will eventually reliably broadcast an A_VECT message with a vector V_j containing ID(M). Consequently, all correct processes will deliver at least f+1 A_VECT messages with a vector V_j containing ID(M). From this point on,

Algorithm 6 Atomic Broadcast protocol (for process p_i).

INITIALIZATION:

1: R_delivered$_i \leftarrow \emptyset$; {messages delivered by the reliable broadcast protocol}
2: A_delivered$_i \leftarrow \emptyset$; {messages delivered by the atomic broadcast protocol}
3: aid$_i \leftarrow 0$; {atomic broadcast identifier}
4: num$_i \leftarrow 0$; {message number}
5: $\forall_j$: B[j] $\leftarrow 0$; {window start}
6: **activate task** (T1,T2);

WHEN **Procedure** `A_Broadcast` (m) is called DO

7: `R_Broadcast` ($\langle$A_MSG, num$_i$, m, i$\rangle$);
8: num$_i \leftarrow$ num$_i$ + 1;

TASK T1:

9: **when** R_delivered$_i \neq \emptyset$ **do**
10: V$_i \leftarrow$ {IDs (j, num$_j$) of the messages in R_delivered$_i$ where B[j] $\leq$ num$_j <$ L};
11: `R_Broadcast` ($\langle$A_VECT, V$_i$, aid$_i$, i$\rangle$);
12: **wait until** ($n - f$ or more $\langle$A_VECT, V$_j$, aid$_i$, j$\rangle$ messages have been delivered);
13: W$_i \leftarrow$ IDs (j, num$_j$) that appear in $f + 1$ or more vectors V$_j$ and B[j] $\leq$ num$_j <$ L;
14: W $\leftarrow$ M_V_Consensus(W$_i$, aid$_i$);
15: **wait until** (all messages with IDs in W are in R_delivered$_i$);
16: Atomically deliver messages with IDs in W in a deterministic order;
17: A_delivered $\leftarrow$ A_delivered $\bigcup$ W;
18: **while** message with ID (j, B[j]) $\in$ A_delivered$_i$ **do**
19: B[j] $\leftarrow$ B[j] + 1;
20: **end while**
21: aid$_i \leftarrow$ aid$_i$ + 1;
22: **end when**

TASK T2:

23: **when** $\langle$A_MSG, num$_j$, m, $j\rangle$ is delivered by the reliable broadcast protocol **do**
24: R_delivered$_i \leftarrow$ R_delivered$_i \bigcup \{\langle$A_MSG, num$_j$, m, $j\rangle\}$;
25: **end when**

all correct processes will propose a vector W for the multi-valued consensus protocol that contains ID(M) until a decision not $\perp$ is reached.

There is one execution of the multi-valued consensus protocol that reaches a decision because correct processes will not try to reach agreement on messages whose ID is equal or greater than ID(M)+B while M is not delivered. There must be an execution of MVC where all correct processes propose a vector W containing the same set of values W = {ID(M),ID(M)+1,...,ID(M)+B}. The decision of the MVC is guaranteed by the protocol property MVC1.

The protocol might block in line 15 where it waits until all messages that have to be delivered by the atomic broadcast protocol (those with IDs in vector W) are in R_delivered. A message with an ID in vector W must have already been delivered by the reliable broadcast protocol to at least one correct process. Because of properties RB1-RB3, this message will eventually be delivered to all correct processes, so no correct process blocks on line 15. □

Theorem 7 *(Agreement) If a correct process delivers a message m, then all correct processes eventually deliver m.*

Proof: The protocol assumes that a given correct process, say p_i, delivers M. Therefore: (1) the multi-valued consensus in line 14 decides on a vector W containing ID(M); and (2) the reliable broadcast protocol delivers M to p_i, therefore it delivers M to all correct processes (properties RB1-RB3). All correct processes get the same result from the multi-valued consensus, so all eventually deliver M. □

Theorem 8 *(Integrity) For any identifier ID, every correct process p delivers at most one message m with identifier ID, and if $sender(m)$ is correct then m was*

previously broadcasted by $sender(m)$.

Proof: The proof of the first assertion is trivial from the inspection of the algorithm. The proof of the second assertion follows directly from the properties of the reliable communication channels (RC1-RC2). □

Theorem 9 *(Total order) If two correct processes deliver two messages* m_1 *and* m_2*, then both processes deliver the two messages in the same order.*

Proof: Any correct process delivers messages only after an execution of multi-valued consensus (line 14-16). All correct processes execute the same instances of the multi-valued consensus protocol, identified by *aid*. The messages which are delivered are all those whose IDs are in vector W returned by multi-valued consensus and the order of delivery is deterministic (line 16). Therefore, all processes deliver the same messages in the same order. □

Chapter 4

RITAS: The Implementation

RITAS is the implementation of the protocol stack described in Chapter 3. This chapter 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. The chapter is organized as follows. Section 4.1 is concerned with the design considerations of RITAS. Section 4.2 describes the internal architecture and data structures of the implementation. Finally, Section 4.3 focuses on the external interface of the protocol stack.

4.1 Design Considerations

This section discusses the several design considerations that had to be taken into account before the actual implementation took place. These considerations provided a point from which the conception of the whole internal structure of RITAS could be driven.

4.1.1 Single-threaded vs. Multi-threaded Operation

One of the design options with significant influence on the efficiency of RITAS was the single-threaded operation of the protocol stack. When developing a software component such as a protocol stack, there are two possible options regarding its operation: multi-threaded or single-threaded. The RITAS protocol stack runs in a single thread, independent of the application thread. The rationale for this choice is explained below.

In a typical multi-threaded 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 responsible for reading messages from the network. The multi-threaded approach may be simpler to implement since there is practically no need to synchronize the different threads (each one deals with a separate protocol instance), each context is self-contained in a given thread, and there is virtually no need for protocol demultiplexing since the messages can be addressed directly to the threads handling them. The most important advantage of this approach is that it allows for cleaner implementations (i.e., more verbatim translations from pseudocode) because the protocol code has only to deal with one protocol instance (the context is implicit). Nevertheless, as new messages are received, more and more threads are created to handle the different protocol instances. This leads to a situation where the constant context switching between the various threads - and a loaded system can easily be in a situation where it needs several hundreads of threads to deal with all protocol instances - 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 such an 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 (each one representing a different protocol instance).

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 outweighted by its potential performance advantages. The same rationale was used for the programming language of choice. All of the protocol stack was implemented using the C language, while resorting to third-party libraries whenever possible (e.g., *openssl* was used for all cryptographic operations).

4.1.2 Message Management

Messages are the central element of any network protocol stack. Everything is built around them. Protocol layers are really there to process messages. So, if there is a single feature of a protocol stack that determines its efficiency, that must be the way messages are managed by the protocol stack. More specifically, the way they are created, destroyed, recycled, modified, and passed along the various protocol layers is of vital impor-

tance to an efficient stack operation.

When dealing with a multi-layered network protocol stack, messages need to be passed back and forth the network stack. A certain degree of flexibility is needed to manipulate the buffers which hold the messages, data may need to be prepended or appended to these buffers, existing data may need to be transformed or deleted, and the amount of operations that actually copy data needs to be kept to a minimum in order to maintain efficiency in the stack. Additionally, messages may need to be tagged with several meta-information such as their size, where they came from, or the bounds of certain headers inside the message.

This implies the need for some kind of data structure, preferably hidden behind a simple interface that abstracts most of the complex operations needed for its manipulation. Examples of such operations are initialization and destruction of messages, and the prepending and appending of data to the message. Besides being able to hold a message, this data structure should have the fields to store all the necessary meta-information for a correct and efficient message management.

In RITAS, such functionality is implemented by a data structure which we call *mbuf*. It was inspired by the TCP/IP implementation in the Net/3 Operating System kernel (Wright & Stevens, 1995), and it is described in detail later in this chapter.

4.1.3 Multiple Protocol Instances

Since we would like to evaluate the performance of the RITAS protocols, the ability to execute multiple instances of the same protocol is a strict requisite. This implies a need to have and efficiently manage several 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 multiple protocol instances: the RITAS Channel, and Control Block Chaining. Both are detailed later in this chapter.

4.1.4 Protocol Demultiplexing

Protocol demultiplexing is a problem that presents itself naturally in a stack such as the proposed one. It occurs when one or more protocols have to deliver messages to multiple protocols (or layers). In RITAS, this problem is present, for instance, in the broadcast primitives. For example, since the service provided by the reliable broadcast protocol is used by all the protocols above it in the stack, the reliable broadcast protocol needs to know which layer it should deliver its messages to.

The problem can be more tricky than what it may seem at a first glance if one wishes to keep the layers as transparent as possible from one another. The option usually used is to have a field in the lower-layer protocol header that gives indication of the higher-layer protocol to which the message should be delivered. The implication of this solution is that protocol transparency is lost. This field in the lower-layer protocol header must either be written directly by the higher-level protocol, or some sort of extra communication between the two layers is required.

The latter option seems to be the less stringent to layer transparency. It is the one used by RITAS, and is implicit in its management of control

blocks which is described in later sections.

4.1.5 Header Construction

When dealing with a layered protocol stack, it is important to keep each layer as separated as possible. As a message is passed along the stack, a protocol layer should not have to worry about making any special mangling to the message depending on which layer it is going to pass it.

Following this principle, when a layer receives an inbound message, its contents should remain untouched except for the protocol header at the beginning of the message. After processing the message, the header should be stripped off before passing the message to the upper layer protocol. When the message is outbound, the same principle applies. The contents of the message should be completely opaque to the protocol layer receiving the message, and a protocol header should be prepended at the beginning of the message.

4.1.6 Storage of Values

All protocols in RITAS have a similar structure. They perform their actions based on the values received from other processes. An action is triggered usually when some value reaches a given threshold. Protocols need to keep track of the values received and their respective count. Except for the binary consensus protocol, all protocols accept values with an arbitrary length. This implies a need to efficiently compare blocks of data with an arbitrary length, some possibly large.

RITAS makes extensive use of hash tables to deal with this issue. This makes possible the quick retrieval of the count associated with each value.

4.1.7 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 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. There must be a way to store these messages as they arrive, and to efficiently retrieve them when a context that can handle them is created. The protocol stack must be able to handle this issue in an efficient way.

4.2 Internals

Internally, three major data structures form the core of the RITAS operation: the RITAS context, the message buffers, and the protocol control blocks and their respective protocol handlers. There are other data structures and algorithmic operations that have a key role in the protocol stack: the RITAS channel, control block chaining, and out-of-context message handling. This section will describe these subjects. Additionally, several data structures and functions provide ancillary common operations for general use: hash tables, lists, FIFO queues, and vectors are the most useful. Those will not be subject of further study. Because of their general nature, it is assumed that the reader is already familiarized with these concepts.

4.2.1 The RITAS context

The operation of RITAS revolves around a monolithic data structure called `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. Every internal and external function of RITAS needs to take a pointer to this data structure as a parameter.

Figure 4.1 shows the `ritas_t` data structure. There are additional fields to `ritas_t` that are not represented in the figure. Those are mostly variables used for internal synchronization (e.g., mutexes, etc.) and were purposedly left out for simplification.

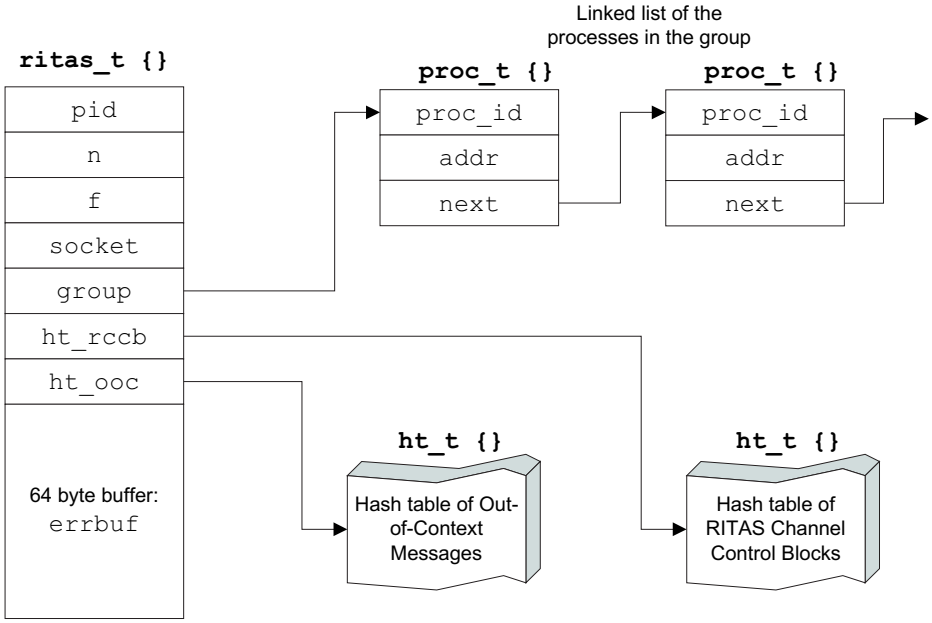

Figure 4.1: The *ritas_t* structure.

The `pid` field is an integer between 0 and $n-1$ that uniquely identifies the process inside the group. The elements `n` and `f` are self-explanatory, they indicate the number of processes n in the group, and the maximum

number of processes f that can fail. The field `socket` is a socket descriptor where the process receives incoming connections. The pointer `group` points to a linked list of `proc_t` data structures. These hold the information for each process in the group such as the identifier, network address, etc. The `ht_rccb` element is a pointer to a hash table that holds the RITAS Channel control blocks, while `ht_ooc` points to a hash table that hold the incoming out-of-context messages.

4.2.2 Message Buffers

In RITAS, 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 (Wright & Stevens, 1995). The *mbuf* is used to store messages and several metadata related to their management. One instance of *mbuf* can only hold a single message. 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.

Figure 4.2 shows the structure of an *mbuf*. Each field is explained below:

prev Pointer to the previous *mbuf* in a *mbuf* list.

next Pointer to the next *mbuf* in a *mbuf* list.

head Pointer to the beggining of the message held by the *mbuf*. This is some memory address inside the *data* buffer, lower or equal than the *tail* pointer.

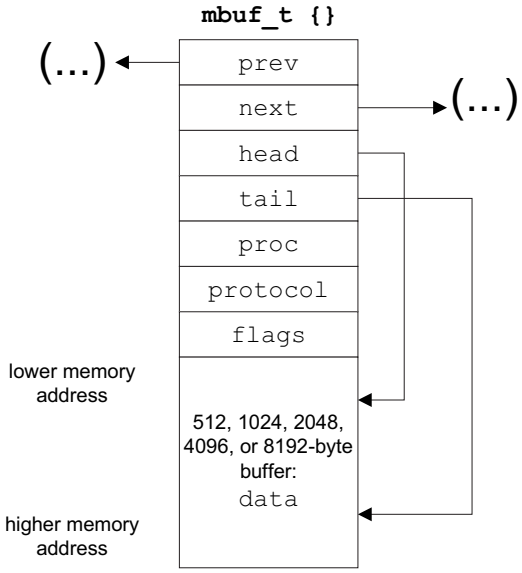

Figure 4.2: The *mbuf* structure.

`tail` Pointer to the end of the message held by the *mbuf*. This is some memory address inside the *data* buffer, higher or equal than the *head* pointer.

`proc` Identifier of the process that sent the message held by the *mbuf*. Only relevant if it is an inbound message.

`protocol` Indicates which was the latest protocol layer to process the message.

`flags` Special flags. These indicate to the `mbuf` manipulation functions if some sort of special behavior is necessary when processing the `mbuf`.

`data` Buffer where the actual message held by the *mbuf* is stored.

A *mbuf* is usually created when a new message arrives from the network. The RITAS network scheduler, prior to reading data from the socket,

creates a *mbuf*, then it reads the message from the socket directly into the *mbuf* `data` buffer, and passes the *mbuf* to the appropriate protocol layer. A *mbuf* can also be created by any specific protocol layer, for instance if it needs to send a message to other processes, but every *mbuf* is reutilized as much as possible within the protocol stack.

Regarding *mbuf* destruction, there are also specific rules as to when a *mbuf* should be destroyed. For an outbound *mbuf*, the *mbuf* should be destroyed immediatly after its message is sent to all relevant processes. The exception is when the `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* being processed is relevant.

4.2.3 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. The protocol handler is the set of functions that implement the operation of the protocol.

The control block data structure is responsible for holding all the nec-

essary context for the execution of a specific instance of the protocol. It keeps tracks of things like the instance identification, the current protocol step, the values received so far, etc. The code block below presents the skeleton of a typical control block using the binary consensus protocol as an example.

```
/**
 * General Control Block Information. In one form  or another, all control blocks in RITAS maintain
 * this information.
 */

/* Identifier of the specific protocol instance. */
u_short id;

/* Indicates if this control block is chained to an upper-layer control block. */
u_char upper;

/* If so, this pointer points to it. */
void *parent;

/* A linked list of the control blocks used by this protocol instance as primitives. In this case, the
 * binary consensus protocol needs several reliable broadcast control blocks in order to communicate. */
ritas_rbcb_t **rbcb;


/**
 * Now we're entering more protocol-specific state information. This can be anything the protocol
 * needs in order to maintain  the logical state of the protocol instance. Below are a few examples.
 */

/* The current round the protocol is in. */
u_short round;

/* The current step the protocol is in. */
u_short step;

/* The binary majority value of the specific steps of the protocol. */
u_char majority_value[3];

/* (...) */
```

Protocol handlers are formed by a 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 struc-

tures 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 code block below depicts the skeleton of a typical protocol handler using the binary consensus protocol as an example. The communication between the protocols is depicted in Figure 4.3.

```
/*
 * Initialization function. Creates a binary consensus control block.
 */
ritas_bccb_t *ritas_bc_init(ritas_t *ctx, u_short id, void *parent);

/*
 * Destruction function. Destroys an existing binary consensus control block.
 */
void ritas_bc_destroy(ritas_t *ctx, ritas_bccb_t *bccb);

/*
 * Input function. Used by the lower-level layers to pass an mbuf to a binary consensus specific
 * protocol instance referenced by the respective control block.
 */
int ritas_bc_input(ritas_t *ctx, ritas_bccb_t *bccb, ritas_mbuf_t *m);

/*
 * Output function. Used by the upper-level layers to pass an mbuf to a binary consensus specific
 * protocol instance referenced by the respective control block.
 */
int ritas_bc_output(ritas_t *ctx, ritas_bccb_t *bccb, ritas_mbuf_t *m);

/*
 * Function that exports the binary consensus protocol functionality to applications.
 */
int ritas_bc(ritas_t *ctx, u_short bcid, u_char proposal);
```

4.2.4 The RITAS Channel

The *RITAS channel* 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 Figure 4.3). It is placed in the stack such that 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.

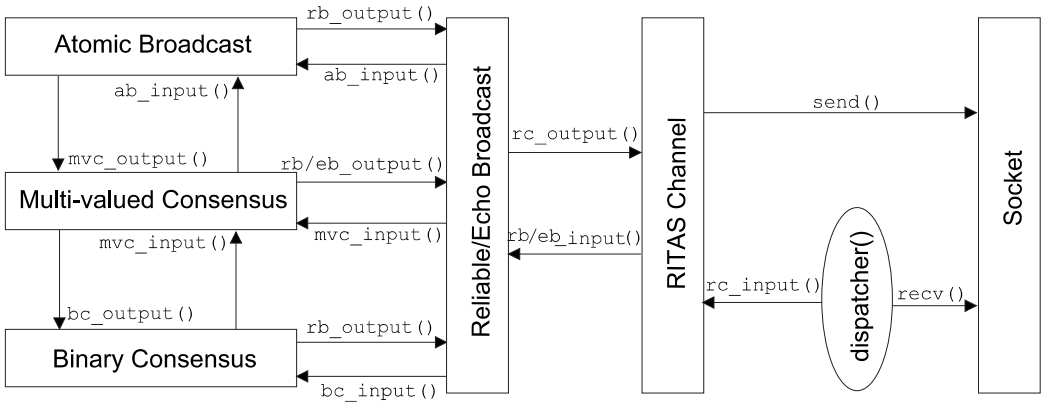

Figure 4.3: Communication flow between the various protocol layers.

The purpose of the RITAS channel 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.

4.2.5 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 identify all messages, provides for seamless protocol demultiplexing, and facilitates control block management.

Control block chaining works in the following way. Suppose an application creates an atomic broadcast protocol instance. This creation 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 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. 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. Figure 4.4 illustrates this technique.

An 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 sub-tree) of control blocks is automatically destroyed when its root node is eliminated.

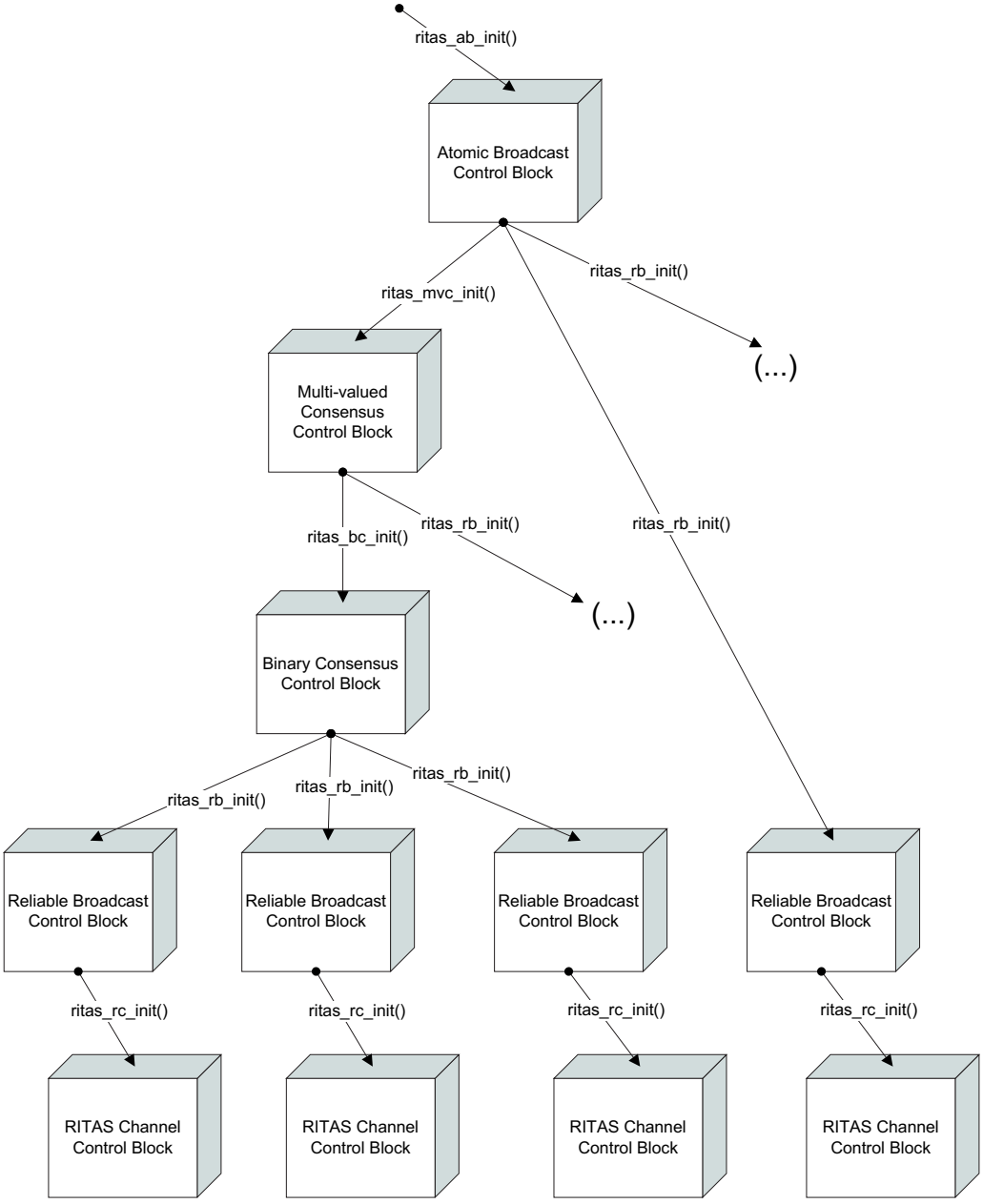

Figure 4.4: Initialization of a tree of control blocks.

4.2.6 Out-of-Context Message Handling

As indicated in our design considerations, out-of-context messages cannot simply be discarded since they are potentially necessary for the correct operation of future protocol instances. What is done is that 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 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.

4.3 Interface

RITAS exports a simple API for applications who wish to access the protocols provided by the stack to build distributed systems services. The API revolves around the RITAS context `ritas_t`, however, this data type 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 4 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 protocols as many times as wished (however, functions are blocking and not thread-safe).

4. Destroy the RITAS context by calling `ritas_destroy()`.

4.3.1 Context Management Functions

The context management functions allow for the basic management of a communication session. This includes the initialization and destruction of a session context, and the addition of processes to the session. Since the notion of group in RITAS is static, the addition of processes can only be performed before any kind of communication takes place. There is no operation to remove processes from the group since this would be incongruent with the system model and break the correctness of the protocols.

```
ritas_t *ritas_init(
    u_short pid,
    u_short n,
    u_short f,
    u_short port,
    u_char *errbuf);
```

`ritas_init()` initializes a new RITAS context. It allocates the necessary memory space for the `ritas_t` data structure and initializes its internal variables and data structures. The main arguments are: a process identifier `pid`; the total number of processes `n`; the maximum number of corrupt processes `f`. In case of success the function returns a pointer to a freshly created RITAS context; otherwise, it returns `NULL` and an appropriate zero-terminated error message is copied to `errbuf`.

```
void ritas_destroy(
    ritas_t *ctx);
```

`ritas_destroy()` destroys a previously initialized RITAS context, `ctx`. The internal context data structures are freed from memory along with the context itself.

```
int ritas_proc_add_ipv4(
    ritas_t *ctx, u_short pid,
    u_char *ip,
```

```
    u_short port,
    u_char *key);
```

`ritas_proc_add_ipv4()` adds a process to the context, `ctx`. The functions takes as argument a pointer to the IPv4 address of the process, `ip`. In case of success, the function returns 1; in case of failure returns -1.

4.3.2 Service Request Functions

The service request functions give the application programmer access to the actual protocols provided by the stack. These functions can be divided in two groups, one for the broadcast primitives and another for the various consensus protocols. The service request functions can only be called after the relevant session context has been properly initialized and the individual processes added to the group. When a session context is destroyed, no service requests functions for that particular session can be called afterwards.

```
int ritas_rb_bcast(
    ritas_t *ctx,
    u_short rbid,
    u_char *buf,
    u_short buf_s);
```

`ritas_rb_bcast()` reliably broadcasts a message to the group. The function takes as arguments a pointer to the relevant session context `ritas_t`. An identifier for the broadcast `rbid`. A pointer to a buffer `buf` containing the message to be broadcasted. Finally, the size of message `buf_s` in bytes. In case of success, the function returns 1; in case of failure returns -1.

```
int ritas_rb_recv(
    ritas_t *ctx,
    u_short txid,
    u_short rbid,
    u_char *buf,
    u_short buf_s);
```

`ritas_rb_recv()` delivers a message that was reliable broadcasted by some process belonging to the group. The function blocks until it is able to deliver the relevant message. It takes as arguments a pointer to the session context `ritas_t`. The identifier of the sender process `txid`. An identifier for the broadcast `rbid`. A pointer to a buffer `buf` in which the delivered message should be stored. The maximum length `buf_s` in bytes that the buffer can hold. In case of success, the function returns the length of the message in bytes; otherwise it returns -1.

```
int ritas_bc(
    ritas_t *ctx,
    u_short bcid,
    u_char proposal);
```

`ritas_bc()` runs a binary consensus execution with identifier `bcid`. The `proposal` value is passed to the function as an argument, and the latter blocks until the processes reach a decision. In case of success the functions returns the decision value which is either 0 or 1; in case of failure the function returns -1.

```
int ritas_mvc(
    ritas_t *ctx,
    u_short mvcid,
    u_char *prop,
    u_short prop_size,
    u_char *decision,
    u_short decision_size);
```

`ritas_mvc()` runs a multi-valued consensus execution identified by `mvcid`. The pointer `prop` points to a buffer containing the proposal value, and `prop_size` is the size of this data. Another pointer `decision` is used to reference the memory location where the decision value should be stored. The maximum length of data that can be stored in this buffer is indicated by `decision_s`. In case of success, the function returns the length of the decision value in bytes; in case of failure returns -1.

```
int ritas_vc(
```

```
    ritas_t *ctx,
    u_short vcid,
    u_char *proposal,
    u_short prop_size,
    u_char *decision,
    u_short decision_size);
```

`ritas_vc()` runs vector consensus executions identified by `vcid`. The functions blocks until a decision is reached. The proposal value is passed as a pointer to a buffer `proposal` containing the value of length `prop_s`. The decision vector is stored in the buffer pointed by `vec`. The maximum length of data that this buffer can hold is indicated by `vec_s`. In case of success, the function returns the length of the decision vector in bytes; in case of failure returns -1. The decision vector can be extracted into a data structure `ritas_vector_t` that makes it easier to process using the ancillary function `ritas_vector_extract()`.

```
int ritas_ab_bcast(
    ritas_t *ctx,
    u_char *buf,
    u_short buf_s);
```

`ritas_ab_bcast()` atomically broadcasts a message to the group. The message is passed as a pointer to the buffer `buf` that holds it. The message length is indicated by `buf_s`. In case of success, the function returns 1; in case of failure returns -1.

```
int ritas_ab_recv(
    ritas_t *ctx,
    u_char *buf,
    u_short buf_s,
    ritas_ab_header_t *abh);
```

`ritas_ab_recv()` delivers a message that was atomically broadcasted by some process in the group. The functions blocks until a message is delivered. The message is stored in the buffer pointed by `buf`. The maximum length in bytes that the buffer can hold is indicated by `buf_s`. The function takes a pointer `abh` to a data structure `ritas_ab_header_t`-

`t` where it is stored some meta-information about the delivered message such as its total order number. In case of success, the function returns the length of the message in bytes; otherwise it returns -1.

Chapter 5

Performance Evaluation

This chapter describes the performance evaluation of RITAS in a local-area network setting (LAN). Two different performance analysis are made. First is presented a comparative evaluation in order to gain insight into the stack, and how protocols relate and build on one another performance-wise. Second is conducted an in-depth analysis of 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 (Castro & Liskov, 1999; Correia et al., 2006a; Reiter, 1995).

5.1 Testbeds

The experiments were carried out on two different testbeds. One with older hardware and only four hosts, and another with modern hardware and ten hosts.

The first, which will be refered as *tb-slow*, consisted on four Dell Pen-

tium 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 in full-duplex mode.

The second testbed, which will be referred as *tb-fast*, consisted of 10 Dell PowerEdge 850 servers. These servers have Pentium 4 CPUs with 2.8 GHz of clock speed, and 2GB of RAM. They were connected by a Dell PowerConnect 2724 network switch with 10/100/1000 Mbps of bandwidth capacity. The operating system was Linux 2.6.11. Bandwidth tests showed a consistent throughput of 1.16 MB/s for the 10 Mbps setting, 11.5 MB/s for the 100 Mbps setting, and 67.88 MB/s for the 1000 Mbps setting. All values were taken in full-duplex mode, which was used in the experiments.

In both testbeds the used IPSec implementation was the one available in the Linux kernel and the reliable channels that were established between every pair of processes employed the AH protocol (with SHA-1) in transport mode (Kent & Atkinson, 1998).

5.2 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 two second 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. 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. Outliers were identified and excluded from the sample.

	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 5.1: Average latency for isolated executions of each protocol (with IPSec and IP) in testbed *tb-slow* (100 Mbps).

The results for testbed tb-slow, shown in Table 5.1, 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 multi-valued consensus. For a multi-valued 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 multi-valued

consensus. The experiments also showed 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 header adds another 24 bytes, which accounts for an extra 30%.

	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	1204	-
	7	3521	192%
	10	7907	557%
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%

Table 5.2: Average latency and relative slowdown (w.r.t. to the four-process scenario) for isolated executions of each protocol (with IPSec) in testbed *tb-fast* in the 1000 Mbps setting.

Table 5.2 shows the performance results for testbed tb-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 striking conclusion that can be extracted from these results is a much better performance of the protocols when compared to the previous testbed. The use of more powerful hardware had a significant impact on the performance of all protocols. For instance, in the case of binary consensus, the performance was improved seven-fold, 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, 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 in the protocol stack performance.

Another interesting observation from the results in Table 5.2 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 with a 38% ($n = 4$) and 79% ($n = 7$) slowdown for echo broadcast, and 36% ($n = 4$) and 76% ($n = 7$) for reliable broadcast. The slowdown for the remaining protocols was considerably accentuated due to larger protocol headers.

5.3 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 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 by varying several system parameters: group size, network bandwidth, faultload, and message payload size.

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 assume three values: 10 Mbps, 100 Mbps, and 1000 Mbps.

The *faultload* defines the types of faults that are injected in the system during its execution. The measurements were taken under three different faultloads. In the failure-free faultload all processes behave correctly. In the fail-stop faultload 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 faultload 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 multi-valued 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 Kilobyte, and 10 Kilobytes.

5.3.1 Group Size and Faultload

The set of experiments described in this section had the objective of measuring the impact of both the group size and the faultload. The network bandwidth was fixed to 100 Mbps in testbed tb-slow, and to 1000 Mbps in testbed tb-fast. The message payload size was 100 bytes. The group size was tested for 4, 7, and 10 processes. All the three different faultloads were also tested: failure-free, fail-stop, and Byzantine.

Failure-free faultload. Figure 5.1 shows the performance of the atomic broadcast in testbed tb-fast when no faults occur in the system. Each curve shows the latency or throughput for a different group size n.

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}. The throughput stabi-

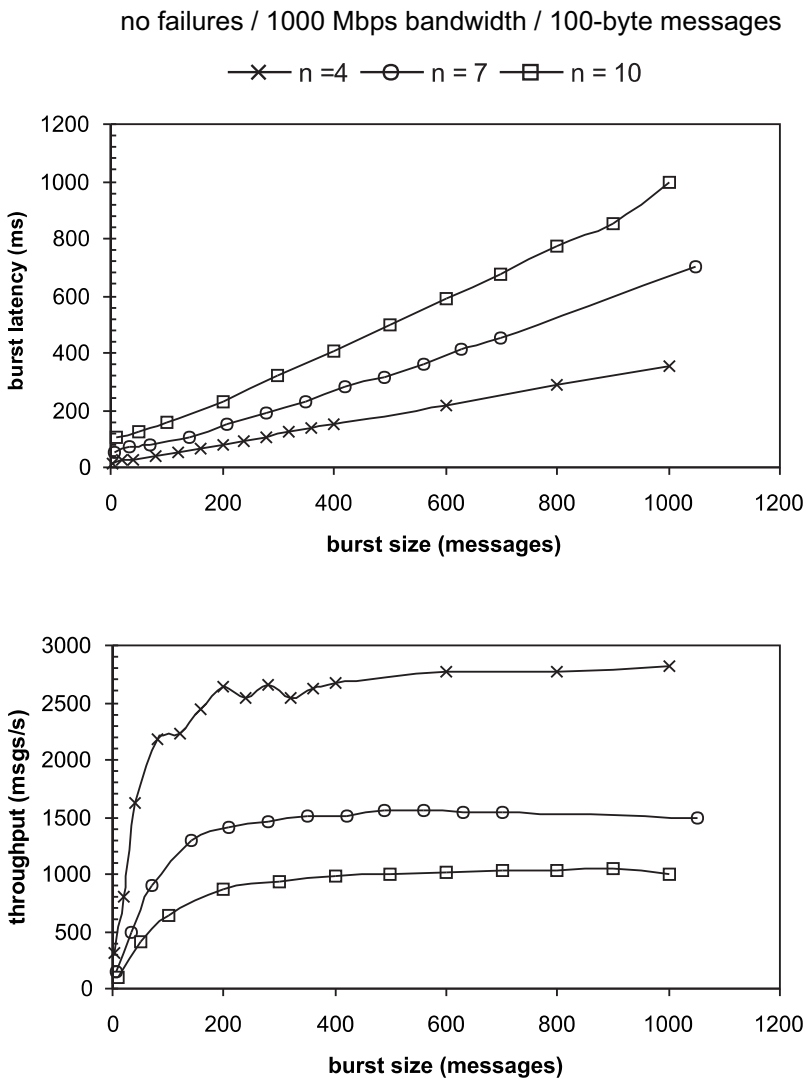

Figure 5.1: Latency and throughput for atomic broadcast with failure-free faultload, 1000 Mbps bandwidth, and 100-byte messages in testbed tb-fast.

lizes around 2800 messages/s for a group size of 4 processes, 1500 msgs/s for 7 processes, and 1000 msgs/s for 10 processes. The group size had a significant impact on the protocol performance. The maximum throughput dropped almost to half from the four-process to the the seven-process scenario, and then about one third from the seven-process to the ten-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 faultload. The performance of the atomic broadcast protocol for testbed tb-fast with f crashed processes is presented in Figure 5.2. In this faultload, each correct process sends a burst of $\frac{k}{n-f}$ messages. Each curve shows the latency or throughput for a different group size n.

Looking at the curves, it is possible to conclude that performance is noticeably better with f crashed processes than in the failure-free situation. This happens because with f less processes there is less contention in the network which allows operations to be executed faster. The maximum throughput T_{max} is around 3000 messages per second for a group size of 4 processes, 1700 msgs/s for 7 processes, and 1050 msgs/s for 10 processes.

Byzantine faultload. Figure 5.3 shows the performance of atomic broadcast with different group sizes in testbed tb-fast, with f processes trying to disrupt the 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, but

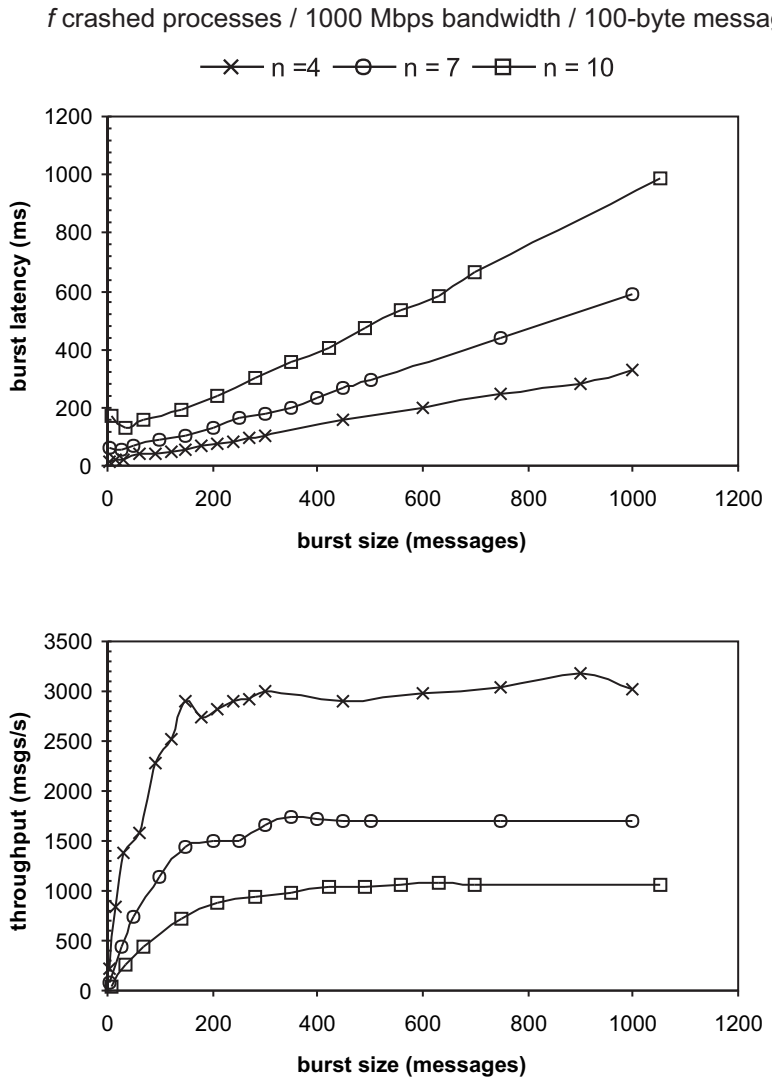

Figure 5.2: Latency and throughput for atomic broadcast with fail-stop fault-load, 1000 Mbps bandwidth, and 100-byte messages in testbed tb-fast.

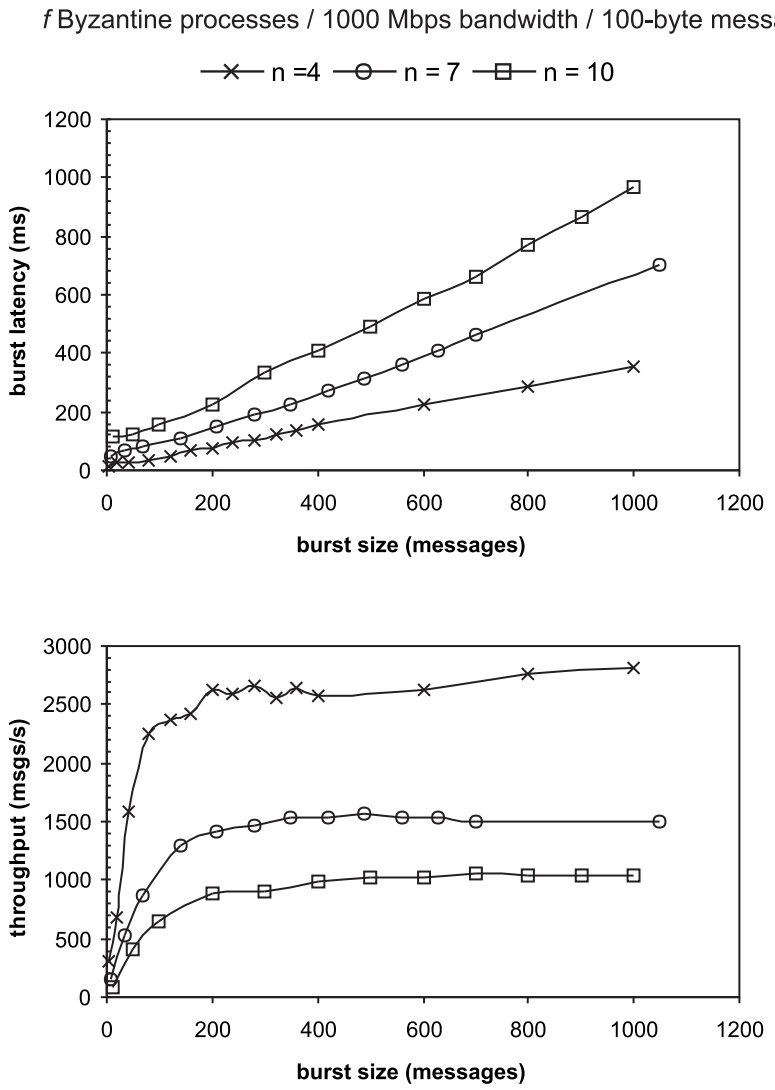

Figure 5.3: Latency and throughput for atomic broadcast with Byzantine faultload, 1000 Mbps bandwidth, and 100-byte messages in testbed tb-fast.

it 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 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.

In more detail, the maximum throughput T_{max} is around 2800 messages per second for a group size of 4 processes, 1500 msgs/s for 7 processes, and 1000 msgs/s for 10 processes.

Testbed tb-slow vs. tb-fast. Figure 5.4 compares the performance for the failure-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 failure-free scenario. The bandwidth for testbed tb-slow is 100 Mbps, and for tb-fast is set to 1000 Mbps.

Unsurprisingly, it can be observed that the performance is clearly superior in testbed tb-fast. The greater computational power and network capacity of tb-fast makes allows a maximum throughput about 4 times larger in the failure-free scenario (2800 msgs/s vs. 650 msgs/s), and 3

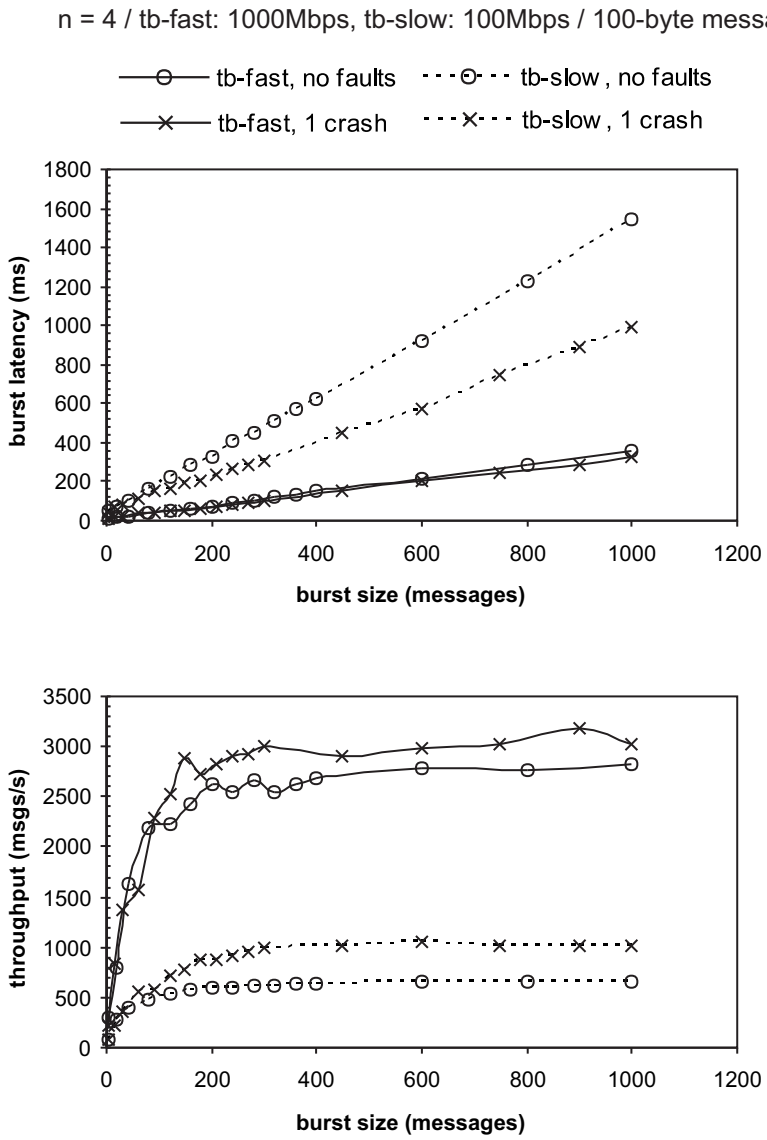

Figure 5.4: Latency and throughput for atomic broadcast with failure-free faultload, and 100-byte messages in both testbeds.

times larger in the fail-stop scenario (3000 msgs/s vs. 1000 msgs/s).

5.3.2 Network Bandwidth and Message Size

This section analyzes in greater detail the impact of network bandwidth and message payload size in the protocol performance. For all the experiments the faultload parameter was set to failure-free, and the group size to four processes. The network bandwidth was varied for the experiments in testbed tb-fast (1000 Mbps, 100 Mbps, and 10 Mbps), and fixed to 100 Mbps in testbed tb-slow. Four message payload sizes were used: 10 bytes, 100 bytes, 1 Kilobyte, and 10 Kilobytes.

Figure 5.5 shows the performance curves for testbed tb-fast with 10-byte 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 1000 Mbps scenario has 100 times more bandwidth than the 10 Mbps scenario, the maximum throughput is only about 1.6 higher in the 1000 Mbps case (2900 msgs/s vs. 1800 msgs/s). It seems that for small payloads (10 bytes), other than the bandwidth, there are more factors to consider when determining the performance bottleneck. The candidates are the processing power of the individual nodes and the network latency. Given the large number of messages that the protocol stack needs to exchange, it is very likely that, for small payload sizes, the latency restricts the performance. Nevertheless, this can only be confirmed by controlling the average network latency and performing the experiments with various latency values in order to measure its impact.

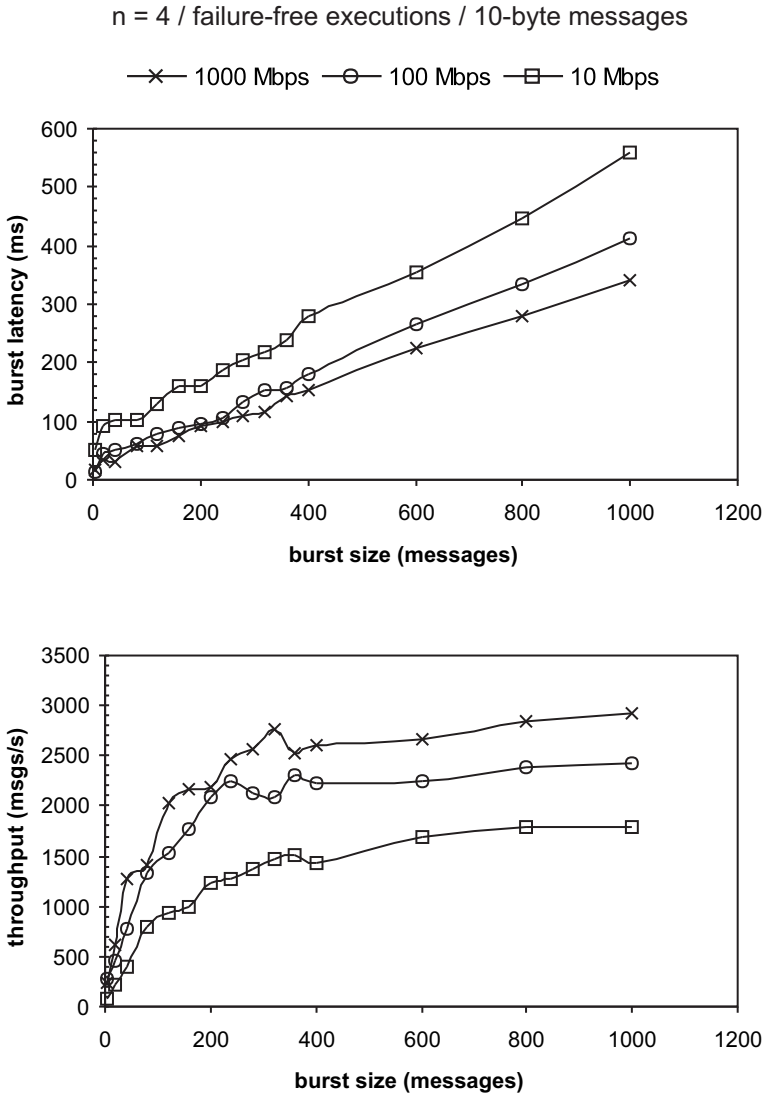

Figure 5.5: Latency and throughput for atomic broadcast with four processes, failure-free faultload, and 10-byte messages in testbed tb-fast.

Figures 5.6, 5.7, and 5.8 show the performance for testbed tb-fast with 100-byte, 1-Kbyte and 10-Kbyte message payloads, respectively. Besides revealing the obvious impact of greater payload sizes on the performance, these figures show that the bandwidth becomes increasingly important as the payload size becomes larger. For 100-byte messages, the maximum throughput decreases 1.15 times from 1000 Mbps (2800 msgs/s) to 100 Mbps (2400 msgs/s), and 2.66 times from 100 Mbps (2400 msgs/s) to 10 Mbps (900 msgs/s). For 1-Kbyte messages this drop is even more drastic. It decreases 1.8 times from 1000 Mbps (2200 msgs/s) to 100 Mbps (1200 msgs/s), and 10 times from 100 Mbps (1200 msgs/s) to 10 Mbps (120 msgs/s). Finally, for 10-Kbyte messages the throughput decreases 3.4 times from 1000 Mbps (440 msgs/s) to 100 Mbps (130 msgs/s), and 10 times from 100 Mbps (130 msgs/s) to 10 Mbps (13 msgs/s). It seems that the network bandwidth only becomes a serious performance bottleneck when the it approaches very small values (i.e., 10 Mbps) or the message payloads become relatively large (i.e., 1 KB and 10 KB).

Finally, Figures 5.9 and 5.10 compare the protocol performance on both testbeds with similar bandwidth values. The purpose is to solely compare the impact of the individual node computational power on the protocol performance. As can be easily observed, testbed tb-fast clearly outperforms tb-slow. There are other factors which can justify this difference. The network switch was an obvious candidate, but it must be ruled out since measurements with similar parameters were taken in testbed tb-fast using the network switch of tb-slow (HP ProCurve 2424M) and the results were actually just slightly better than with the network switch of tb-fast (Dell PowerConnect 2724) in the 100 Mbps mode. With the network switch ruled out, the remaining candidates to justify the performance difference

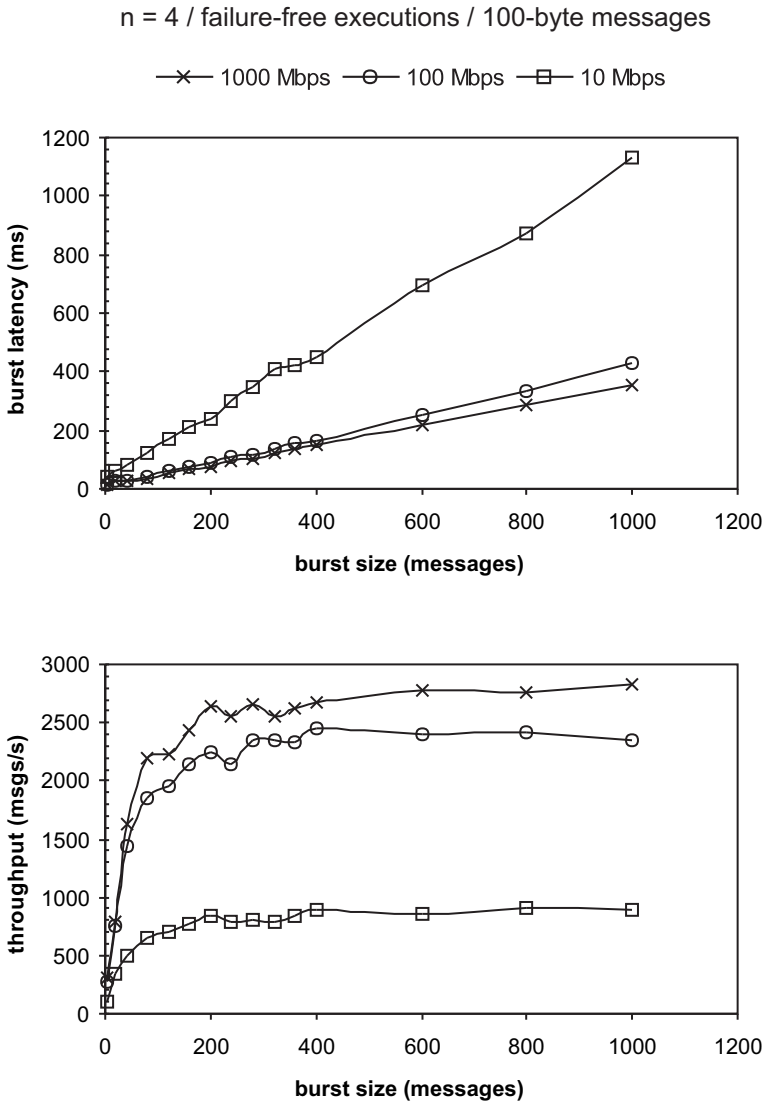

Figure 5.6: Latency and throughput for atomic broadcast with four processes, failure-free faultload, and 100-byte messages in testbed tb-fast.

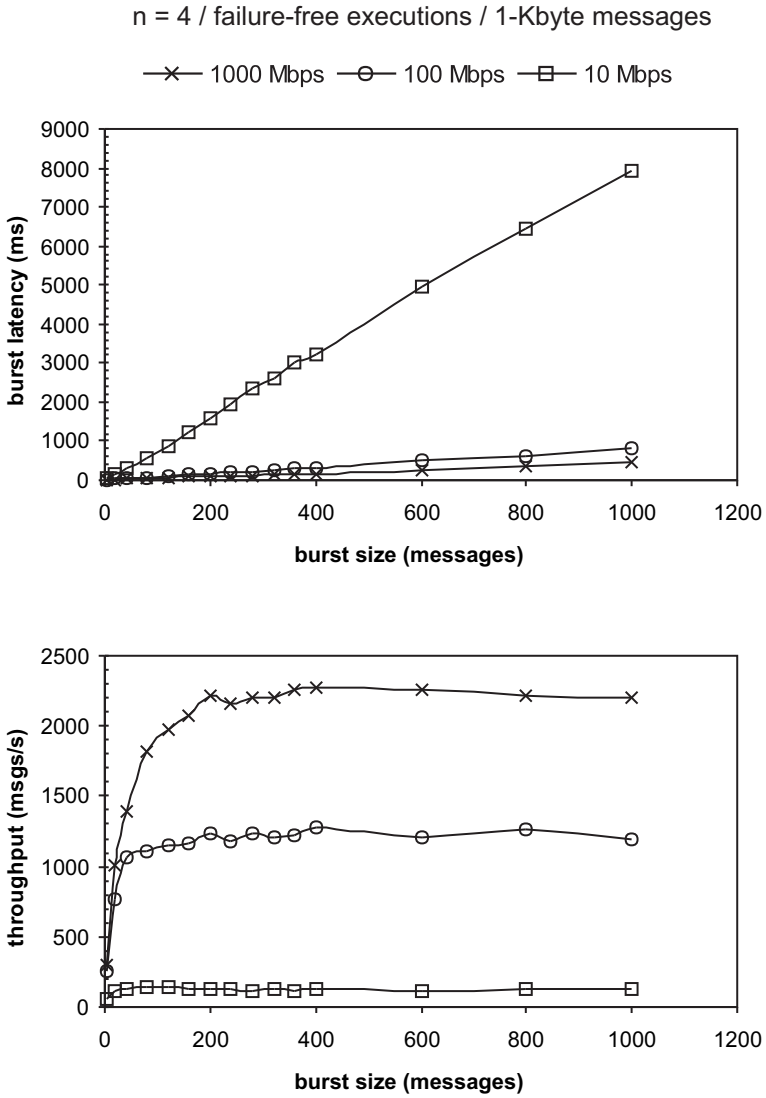

Figure 5.7: Latency and throughput for atomic broadcast with four processes, failure-free faultload, and 1-Kbyte messages in testbed tb-fast.

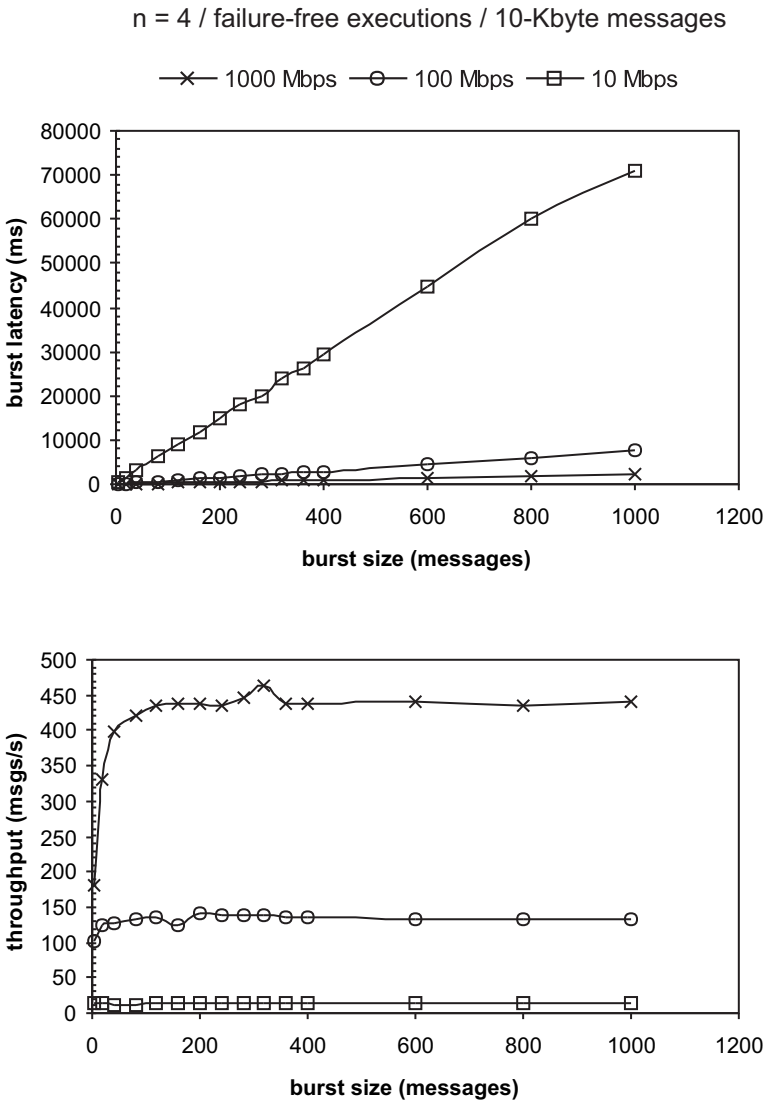

Figure 5.8: Latency and throughput for atomic broadcast with four processes, failure-free faultload, and 10-Kbyte messages in testbed tb-fast.

are the network interface cards and the computational power of the nodes. Both seem likely to have a determinant impact on the performance, however, the relative weight of each one remains unclear.

5.3.3 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 behavior of the protocol has the interesting effect of diluting the cost of the agreements when the load increases.

Figure 5.11 shows the *relative cost* of the agreements with respect to the total number of (reliable and echo) broadcasts that was observed in the failure-free scenario with four processes and 100-byte messages in testbed tb-fast. This relative cost is referred 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 4 messages, it represents about 92% of all broadcasts. This number, however, drops exponentially, reaching as low as 6.3% for a burst size of 1000 messages.

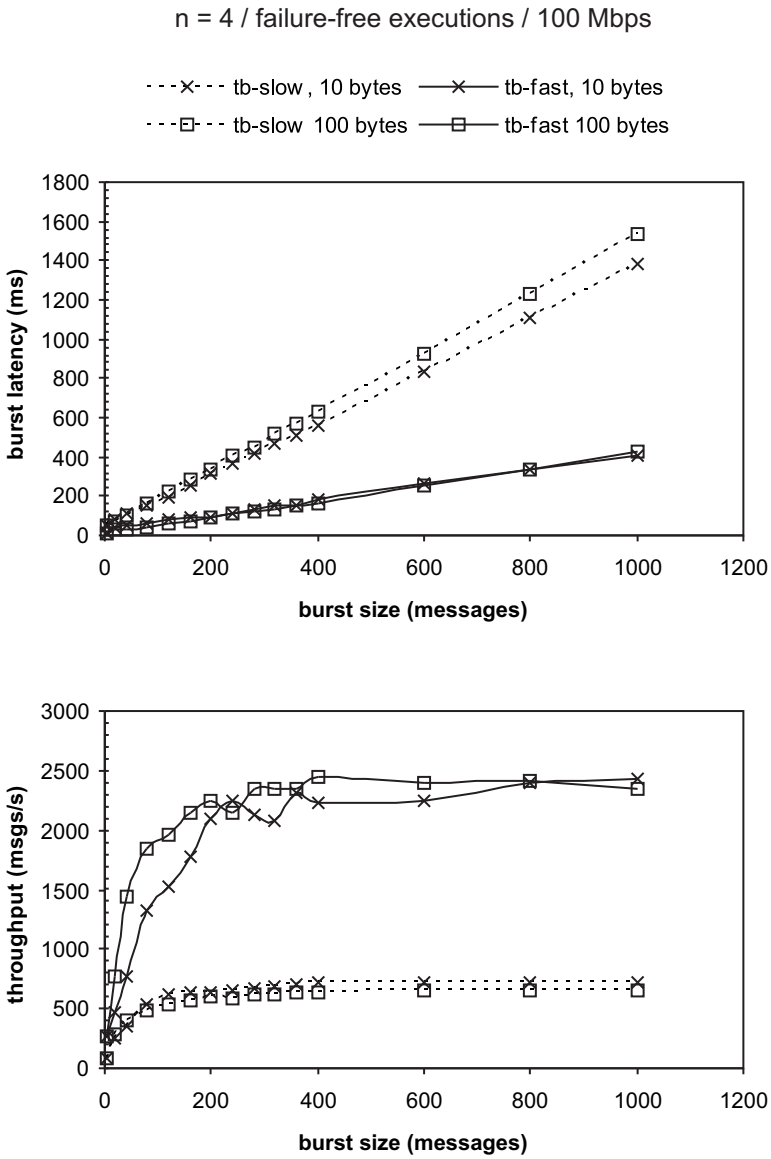

Figure 5.9: Latency and throughput for atomic broadcast with four processes, failure-free faultload, and 100 Mbps bandwidth in both testbeds (10-byte, and 100-byte messages).

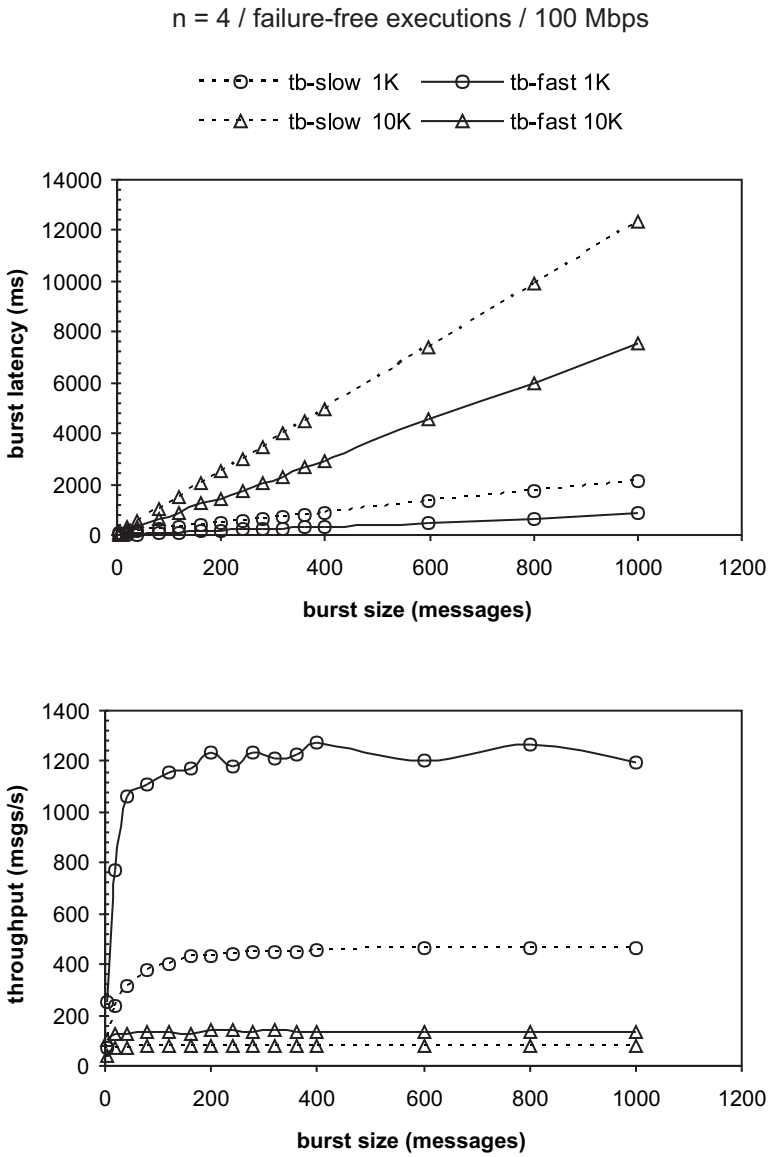

Figure 5.10: Latency and throughput for atomic broadcast with four processes, failure-free faultload, and 100 Mbps bandwidth in both testbeds (1-Kbyte, and 10-Kbyte messages).

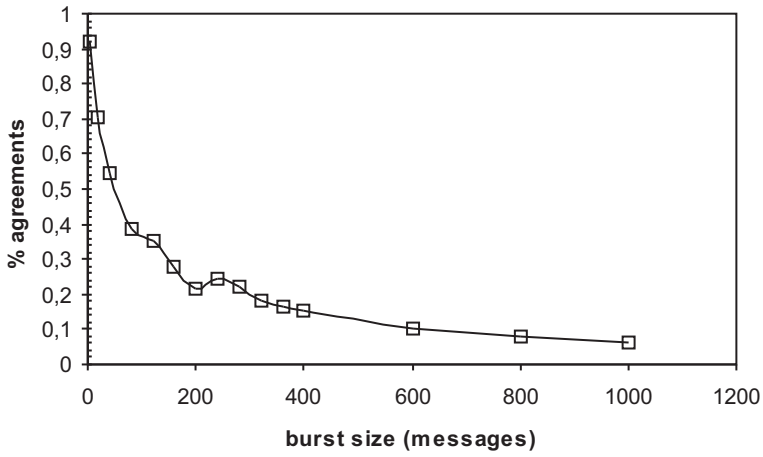

Figure 5.11: Percentage of (reliable or echo) broadcasts that are due to the agreements when a burst of messages is atomically broadcasted. Four-process, failure-free, 1000 Mbps, and 100-byte message scenario in testbed tb-fast.

5.4 Summary of Results

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

- The protocols are robust. Performance (and also 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 recovery actions when a failure occurs, crashes have the effect of making executions faster. Less

processes means less contention on the network.

- The network bandwidth only becomes a serious performance bottleneck when the it approaches small values (i.e., 10 Mbps) or the message payloads become relatively large (i.e., 1 KB and 10 KB).
- The computational capability of the individual nodes has a strong influence on the protocol stack performance.
- 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 6.3% of all (reliable or echo) broadcasts that were made.

Chapter 6

Conclusion

6.1 Conclusions

This paper presents the implementation and performance evaluation of a stack of intrusion-tolerant protocols for distributed systems. 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). A key protocol in the stack - binary consensus - employs randomization in order to avoid the use of timing assumptions in the system.

Most protocols are optimized versions of the previously available algorithms and were implemented in the C language. The protocol stack was packaged as a shared library, and provides a simple interface to applications wishing to use the protocols. The goal of efficiency drove the entire implementation process of the protocol stack.

The performance evaluation of the stack led to several insights about randomized protocols. 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, taking decisions in a distributed way and the absence of failure detectors are important to avoid performance penalties due to the existence of faults. Besides preventing attacks against time assumptions, randomization prevents these performance penalties. The protocols presented evidenced no performance degradation with Byzantine faults, and their execution actually got faster when some of the processes crashed. Third, with the right implementation, the impact of the agreement task in atomic broadcast protocols can be minimal. A high number of atomic broadcasts can be done with a small number of agreements.

Nevertheless, the main conclusion to retain from this work is that 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.

6.2 Future Work

The protocols presented in this thesis showed a good performance in a local-area-network (LAN). This type of environment has some characteristics that favor such protocols where a large number of messages are exchanged: it usually provides a large network bandwidth, and the latency is considerably lower than in a wide-area-network (WAN) environment.

It would be interesting to measure the performance of randomized protocols in other types of environments such as WANs and Wireless Networks, and, possibly, hybrid networks with wired and wireless nodes. Depending on the obtained results, it could be interesting to adapt the

protocols, or develop new ones to these environments.

Bibliography

AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., & LANDWEHR, C., 2004. Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

BEN-OR, M., 1983. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. In: *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30.

BRACHA, G., 1984. An Asynchronous $\lfloor (n-1)/3 \rfloor$-Resilient Consensus Protocol. In: *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162.

BRACHA, G. & TOUEG, S., 1985. Asynchronous Consensus and Broadcast Protocols. In: *Journal of the ACM*, 32(4):824–840.

CACHIN, C., KURSAWE, K., & SHOUP, V., 2000. Random Oracles in Contanstinople: Practical Asynchronous Byzantine Agreement Using Cryptography. In: *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132.

CACHIN, C. & PORITZ, J. A., 2002. Secure Intrusion-tolerant Replication on the Internet. In: *Proceedings of the International Conference on Dependable Systems and Networks*, pages 167–176.

CANETTI, R. & RABIN, T., 1993. Fast Asynchronous Byzantine Agreement with Optimal Resilience. In: *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 42–51.

CASTRO, M. & LISKOV, B., 1999. Practical Byzantine Fault Tolerance. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186.

CHANDRA, T. & TOUEG, S., 1996. Unreliable Failure Detectors for Reliable Distributed Systems. In: *Journal of the ACM*, 43(2):225–267.

CHOR, B. & DWORK, C., 1989. Randomization in Byzantine Agreement. In: *Advances in Computing Research 5: Randomness and Computation*, pages 443–497.

CORREIA, M., NEVES, N. F., LUNG, L. C., & VERÍSSIMO, P., 2005. Low Complexity Byzantine-Resilient Consensus. In: *Distributed Computing*, 17(3):237–249.

CORREIA, M., NEVES, N. F., LUNG, L. C., & VERÍSSIMO, P., 2006a. WormIT – A Wormhole-based Intrusion-Tolerant Group Communication System. In: *Journal of Systems and Software*. To appear.

CORREIA, M., NEVES, N. F., & VERÍSSIMO, P., 2006b. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. In: *The Computer Journal*, 41(1):82–96.

CORREIA, M., VERÍSSIMO, P., & NEVES, N. F., 2002. The Design of a COTS Real-Time Distributed Security Kernel. In: *Proceedings of the Fourth European Dependable Computing Conference*, pages 234–252.

DOLEV, D., DWORK, C., & STOCKMEYER, L., 1987. On the Minimal Synchronism Needed for Distributed Consensus. In: *Journal of the ACM*, 34(1):77–97.

DOUDOU, A., GARBINATO, B., & GUERRAOUI, R., 2002. Encapsulating Failure Detection: From Crash-Stop to Byzantine Failures. In: *International Conference on Reliable Software Technologies*, pages 24–50.

DWORK, C., LYNCH, N., & STOCKMEYER, L., 1988. Consensus in the Presence of Partial Synchrony. In: *Journal of the ACM*, 35(2):288–323.

FISCHER, M. J., LYNCH, N. A., & PATERSON, M. S., 1985. Impossibility of Distributed Consensus with One Faulty Process. In: *Journal of the ACM*, 32(2):374–382.

FRAGA, J. S. & POWELL, D., 1985. A Fault- and Intrusion-Tolerant File System. In: *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218.

GUERRAOUI, R. & SCHIPER, A., 2001. The Generic Consensus Service. In: *IEEE Transactions on Software Engineering*, 27(1):29–41.

HADZILACOS, V. & TOUEG, S., 1994. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Tech. Rep. TR94-1425, Cornell University, Department of Computer Science.

KENT, S. & ATKINSON, R., 1998. Security Architecture for the Internet Protocol. IETF Request for Comments: RFC 2093.

KIHLSTROM, K. P., MOSER, L. E., & MELLIAR-SMITH, P. M., 1997. Solving Consensus in a Byzantine Environment Using an Unreliable Fault

Detector. In: *Proceedings of the International Conference on Principles of Distributed Systems*, pages 61–75.

KIHLSTROM, K. P., MOSER, L. E., & MELLIAR-SMITH, P. M., 1998. The SecureRing Protocols for Securing Group Communication. In: *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, pages 317–326.

KIHLSTROM, K. P., MOSER, L. E., & MELLIAR-SMITH, P. M., 2001. The SecureRing Group Communication System. In: *ACM Trans. Inf. Syst. Secur.*, 4(4):371–406.

MALKHI, D. & REITER, M., 1997. Unreliable Intrusion Detection in Distributed Computations. In: *Proc.of the 10th Computer Security Foundations Workshop*, pages 116–124.

MONIZ, H., CORREIA, M., NEVES, N. F., & VERÍSSIMO, P., 2006. Randomized Intrusion-Tolerant Asynchronous Services. In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 568–577.

NEVES, N. F., CORREIA, M., & VERÍSSIMO, P., 2005. Solving Vector Consensus with a Wormhole. In: *IEEE Transactions on Parallel and Distributed Systems*, 16(12).

PEASE, M., SHOSTAK, R., & LAMPORT, L., 1980. Reaching Agreement in the Presence of Faults. In: *Journal of the ACM*, 27(2):228–234.

RABIN, M. O., 1983. Randomized Byzantine Generals. In: *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409.

REITER, M., 1994. 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.

REITER, M. K., 1995. The Rampart Toolkit for Building High-Integrity Services. In: *Theory and Practice in Distributed Systems*, vol. 938 of *Lecture Notes in Computer Science*, pages 99–110.

REITER, M. K., 1996a. Distributing Trust with the Rampart Toolkit. In: *Communications of the ACM*, 39(4):71–74.

REITER, M. K., 1996b. A Secure Group Membership Protocol. In: *IEEE Transactions on Software Engineering*, 22(1):31–42.

SCHNEIDER, F. B., 1990. Implementing Faul-Tolerant Services Using the State Machine Approach: A Tutorial. In: *ACM Computing Surveys*, 22(4):299–319.

TOUEG, S., 1984. Randomized Byzantine Agreements. In: *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178.

VERÍSSIMO, P., 2002. Traveling Through Wormholes: Meeting the Grand Challenge of Distributed Systems. In: *Proceedings of the International Workshop on Future Directions in Distributed Computing*, pages 144–151.

VERÍSSIMO, P. E., NEVES, N. F., & CORREIA, M. P., 2003. Intrusion-Tolerant Architectures: Concepts and Design. In: R. Lemos, C. Gacek, & A. Romanovsky, editors, *Architecting Dependable Systems*, vol. 2677 of *Lecture Notes in Computer Science*.

WRIGHT, G. R. & STEVENS, W. R., 1995. *TCP/IP Illustrated, Volume 2: The Implementation*.