FACULDADE · DE · CIÊNCIAS UNIVERSIDADE · DE · LISBOA

INTRUSION TOLERANCE BASED ON ARCHITECTURAL HYBRIDIZATION

Miguel Nuno Dias Alves Pupo Correia

Dissertação submetida para obtenção do grau de

DOUTOR EM INFORMÁTICA

Orientador:

Paulo Jorge Esteves Veríssimo

Júri:

Danny Dolev

José Manuel Esgalhado Valença

Paulo da Costa Luís da Fonseca Pinto

Maria Antónia Bacelar da Costa Lopes

Nuno Fuentecilla Maia Ferreira Neves

Julho de 2003

INTRUSION TOLERANCE BASED ON ARCHITECTURAL HYBRIDIZATION

Miguel Nuno Dias Alves Pupo Correia

Dissertação submetida para obtenção do grau de
DOUTOR EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Orientador:

Paulo Jorge Esteves Veríssimo

Júri:

Danny Dolev

José Manuel Esgalhado Valença

Paulo da Costa Luís da Fonseca Pinto

Maria Antónia Bacelar da Costa Lopes

Nuno Fuentecilla Maia Ferreira Neves

Julho de 2003

Resumo

A segurança de sistemas informáticos distribuídos tem sido em grande parte baseada na ideia de prevenção. Têm-se tentado conceber e projectar sistemas "perfeitos", sem vulnerabilidades que possam ser exploradas por potenciais atacantes. A realidade mostra que isso é impossível e que os sistemas vivem num ciclo permanente: vulnerabilidade descoberta – sistemas atacados – remendo publicado – alguns sistemas remendados – nova vulnerabilidade descoberta – etc.

A tolerância a faltas ou, de forma mais genérica, a confiabilidade, há muito que é encarada de uma perspectiva diferente. Nesta disciplina também se tentam construir sistemas tão fiáveis quanto o possível. No entanto, assume-se que os componentes na prática falham, e que esses componentes que falham têm de ser usados para construir sistemas que não falhem.

Se bem que as duas aproximações pareçam quase antagónicas, ataques e intrusões podem ser considerados como faltas. O problema da tolerância a estes tipos de faltas tem recebido alguma atenção ao longo dos anos mas recentemente recebeu um novo fôlego sob a designação de *tolerância a intrusões*.

A presente tese insere-se neste âmbito da investigação em tolerância a intrusões em sistemas distribuídos. Um dos problemas que se põe a esta aproximação, e que a tese aborda, é o da concepção de sistemas que sejam simultaneamente eficientes e eficazes (ou seja, seguros) dada a dificuldade em fazer hipóteses sobre os modos de falha causados pelos atacantes. A tese estuda esta problemática sob a perspectiva inovadora de um modelo de faltas arquitecturalmente híbrido. Este modelo considera que a maior parte do sistema pode falhar de forma arbitrária, até maliciosa, mas que existem componentes que são por construção seguros e tempo-real. O componente explorado nesta tese foi denominado *Trusted Timely Computing Base* (TTCB).

A TTCB é um componente com características inovadoras. Em primeiro lugar, é um subsistema distribuído com a sua própria rede segura. Em segundo lugar, é tempo-real, ou seja, é um subsistema síncrono, capaz de comportamento atempado. Em ter-

ceiro lugar, pode ser realizada usando apenas componentes COTS. A primeira parte da tese apresenta o modelo da TTCB, a sua concretização baseada em componentes COTS e a sua funcionalidade.

Uma vez introduzida a TTCB, é abordada a concepção de diversos componentes de *middleware* tolerantes a intrusões, no sentido de validar a aproximação proposta. Note-se que a TTCB é usada arquitecturalmente como um componente de suporte à execução (*runtime*), e não como um componente da usual pilha de camadas. Este facto dá uma grande versatilidade à arquitectura, uma vez que a TTCB pode ser utilizada indiscriminadamente por todos ou apenas por alguns níveis do sistema. Então é apresentado o primeiro protocolo baseado no modelo de faltas híbrido, um protocolo de difusão fiável. Este protocolo é eficiente e tolera qualquer número de processos maliciosos, ao contrário dos protocolos do mesmo tipo na bibliografia que toleram menos de um terço.

Um problema clássico de sistemas distribuídos, o consenso, é usado para apresentar outra forma de usar a TTCB para suportar protocolos tolerantes a intrusões. O protocolo apresentado é eficiente em termos de complexidade de mensagens e temporal. Serve também de oportunidade para mostrar como o resultado de impossibilidade FLP se relaciona com os sistemas baseados na TTCB.

A comunicação em grupo é um paradigma importante para a construção de sistemas distribuídos tolerantes a faltas. A parte final da tese apresenta um sistema de comunicação em grupo tolerante a intrusões. O sistema inclui um serviço de filiação e uma primitiva de difusão atómica. Este sistema tem um desempenho consideravelmente melhor do que sistemas semelhantes na bibliografia.

PALAVRAS-CHAVE: sistemas distribuídos, tolerância a intrusões, comunicação em grupo, confiabilidade, segurança, tolerância a faltas bizantinas

Abstract

Security in distributed computing systems is usually based on the idea of prevention. The usual approach consists in trying to design "perfect" systems, with no vulnerabilities to be exploited by potential attackers. Reality shows that this is impossible and that systems live in a permanent cycle: vulnerability discovered – systems attacked – patch published – some systems patched – new vulnerability discovered – etc.

Fault-tolerance or, more generically, dependability, takes a different approach. This discipline also tries to build systems as reliable as possible. However, components are assumed to fail, and systems that do not fail have to be built using fallible components.

Although the two approaches seem almost opposite, attacks and intrusions can be considered to be faults. The problem of tolerance of these kinds of faults has been receiving much attention in recent years, and gained a new momentum under the designation of *intrusion tolerance*.

This thesis appears in the context of research on intrusion tolerance in distributed systems. One of the problems with this approach, studied in the thesis, is the design of systems that are simultaneously efficient and secure, given the difficulty of making assumptions about the failure modes caused by the attacker. The thesis is based on an architectural-hybrid fault model. This model assumes that most of the system can fail arbitrarily, even maliciously, with the exception of a few components that are by construction secure and real-time. The component studied in depth in the thesis is called *Trusted Timely Computing Base* (TTCB).

The TTCB is a component with novel characteristics. In the first place, it is a distributed subsystem with its own secure network. Secondly, it is real-time, i.e., a synchronous subsystem capable of timely behavior. Thirdly, it can be implemented using only COTS components. The first part of the thesis presents the TTCB model, its implementation based on COTS components and its services functionality.

Once the TTCB introduced, the thesis describes the design of several intrusion-tolerant middleware components with the objective of validating the proposed ap-

proach. Note that the TTCB is used architecturally as a runtime support component, not as a layer of the usual stack of protocols. This makes the architecture very versatile since the TTCB can be used indiscriminately by all or just some of the system layers. Then, the thesis presents a first protocol based on the hybrid fault model, a reliable multicast protocol. This protocol is efficient and tolerates any number of malicious processes, contrary to similar protocols in the literature that tolerate less than one third.

A classical problem in distributed systems – consensus – is used to show another way of using the TTCB to support intrusion-tolerant protocols. The protocol is efficient in terms of message and time complexities. It also shows how the FLP impossibility result relates to systems based on the TTCB.

Group communication is an important paradigm for the implementation of fault-tolerant distributed systems. The final part of the thesis presents an intrusion-tolerant group communication system. The system includes a membership service and an atomic multicast primitive. This system has an arguably superior performance in relation to similar systems in the literature.

KEY-WORDS: distributed systems, intrusion tolerance, group communication, dependability, security, Byzantine fault tolerance

Acknowledgments

This thesis would not have been possible without the contribution, support and encouragement of many people.

In the first place, I thank my supervisor, Professor Paulo Veríssimo, for all he taught me through these years, for many great ideas that have made scientific research really exciting and for his constant support. This thesis would not exist without him.

In the second place, I thank Professor Nuno Ferreira Neves also for teaching me so much during these last years, for his constant attention to details, his encouragement and support. His contribution to this thesis was also invaluable. Several protocols in the thesis were developed in collaboration with him and, especially BRM-M, was also made in collaboration with Professor Lau Cheuk Lung, to whom I also extend my thanks for his constant support since we met a few years ago.

My sincerest gratitude goes also to my colleagues in the Navigators group and the LASIGE laboratory. Professor António Casimiro Costa was a constant encouragement and helped me in many ways. His work on the Timely Computing Base was an important starting point for this thesis. Dr. Pedro Martins did a great work with the TCB implementation and gave me an indispensable support for the implementation of the initial TTCB prototype. Dr. Nuno Miguel Neves was of permanent assistance and did a great job on the final TTCB prototype. Dr. Paulo Sousa and Dr. Luís Sardinha joined the 'boat' later but supported this work in several ways.

This thesis was developed in the Informatics Department at FCUL. My gratitude goes to all my colleagues and friends in the department, especially to the 'lunch' and

room colleagues, and also to the ex-Navigators, especially Professor Luís Rodrigues and Dr. Hugo Miranda, a constant inspiration.

The work in this thesis was developed in the context of the EU MAFTIA project. Many people in the project contributed directly and indirectly to this thesis, with suggestions, discussions, etc. With the risk of missing some, they were: Dr. Klaus Kursawe, Professor Ian Welch, Dr. William Simmonds, Dr. Cristian Stueble, Dr. André Adelsbach, Dr. Cristian Cachin, Professor Robert Stroud, Professor Brian Randell, Dr. Ives Deswarte and Dr. David Powell. My thanks go also to Professor Rachid Guerraoui, which has read and given feedback on the consensus protocol.

Two professors put me up into this in the beginning: Professor Vítor Vargas and Professor Paulo Pinto. My sincerest gratitude goes again to them, for too many reasons to write in this place. Thanks also to my ex-colleagues at INESC.

Many friends were a constant support and encouragement through these years. I do not mention any because I would forget many more. Thanks!

Last but not least, I must thank all my family. This thesis is dedicated to my wife and my daughters: it was intended for them since the beginning. My parents and the rest of my family were always a great support.

This work was partially supported by the European Community, through project IST-1999-11583 (MAFTIA), and by the Fundação da Ciência e Tecnologia, through the Large-Scale Informatic Systems Laboratory (LASIGE) and project POSI/1999/CHS/33996 (DEFEATS).

À João, à Madalena e à Joaninha.

Table of Contents

List of Figures

List of Tables

1

Introduction

The generalized use of computer networks for communication, access to commercial services, research, or simply for entertainment is no longer a distant goal but became a fact during the last decade. "Internet" is no longer a word reserved for computer scientists and engineers, it became part of common people vocabulary. The explosion on the number of users generated a multitude of paid services and therefore a demand for quality of service. This quality of service includes service reliability, a property impaired by a wave of constantly increasing malicious activity: viruses, worms, hacker attacks...[1]

This scenario is causing a renewed interest in distributed systems security. The classical approach to Security has mostly consisted in trying to prevent bad things from happening. In other words, the objective has been to try and develop "perfect" systems, systems without vulnerabilities, and to patch the vulnerabilities when they are eventually discovered. Dependability has been defined as the property of a computer system such that reliance can justifiably be placed in the service it provides to other systems or humans. The fault tolerance of critical systems, a main paradigm in Dependability, has taken an approach almost opposite to Security. Systems are realistically assumed to be built with components that can fail, but they have mechanisms that tolerate these faults when they happen. The objective is therefore to avoid the system failure, i.e., its departure from the specified behavior, despite faults.

Applications of the fault tolerance paradigm to security have been rare until some years ago when the approach raised attention under the designation of Intrusion Toler-

[1] See, e.g., the CERT Coordination Center statistics at http://www.cert.org/stats/.

ance. The term Intrusion Tolerance appeared in a paper by Fraga and Deswarte (Fraga & Powell, 1985) but was not intensely used for several years. In the meantime a few intrusion-tolerant systems were developed (Deswarte *et al.* , 1991; Reiter, 1995; Kihlstrom *et al.* , 2001; Dutertre *et al.* , 2002; Malkhi & Reiter, 1998; Castro & Liskov, 1999), and the beginning of this decade witnessed a great development in the area, for example with the EU MAFTIA project and the US DARPA OASIS programs [2]. The idea is basically the following (Veríssimo *et al.* , 2003; Adelsbach *et al.* , 2002):

- to assume and accept that the systems remain to some extent vulnerable;
- to assume and accept that attacks on components can happen and some will be successful;
- to ensure that the overall system nevertheless remains secure and operational.

Intrusion Tolerance should not be considered to be a substitute for classical Security techniques and mechanisms, mainly based on prevention, but a complementary approach.

1.1 Objectives

This thesis appears in the context of Intrusion Tolerance, i.e., it applies the Tolerance paradigm to Security. The thesis was developed within the Navigators group of the Large-Scale Informatic Systems Laboratory (LASIGE), at the Informatics Department of FCUL. More specifically, the thesis is part of a research effort of the Navigators group in the above-mentioned EU MAFTIA project and the DEFEATS project [3].

Malicious faults –attacks and intrusions– are always the direct or indirect action of a human being, since "malicious" is an assertion on the intent of that being. Since the agent is intelligent, it is hard (or risky) to make assumptions on malicious faults. This

[2] See http://www.maftia.org and http://www.tolerantsystems.org.
[3] See http://defeats.di.fc.ul.pt/

causes most intrusion-tolerant systems to assume an arbitrary (also called Byzantine) fault model, i.e., to assume that faults can occur both in the domain of time (early or delayed interactions, omissions, crashes) and in the domain of value (interactions with wrong data or semantics).

This thesis considers a different fault model, of the class of hybrid fault models. We consider a system composed of hosts interconnected by a network (a LAN, the Internet) where malicious faults do occur, therefore we also do not make assumptions on these faults. However, we consider the existence of a distributed subsystem that is always correct, i.e., which is not affected by malicious faults. This subsystem is called the Trusted Timely Computing Base (TTCB). Applications run in the normal system, prone to malicious faults, but occasionally they can use the services provided by the TTCB. Because hybrid behavior is enforced, rather that assumed as usually done, the fault model itself is innovative, and we call it *architectural-hybrid*. The first and important consequence of this is that one can assume certain restrictions to malicious behaviors as a natural outcome of the system's structure and thus with high coverage.

Another facet of our work concerns the time or synchrony model. Intrusion-tolerant systems in the literature usually consider an asynchronous time model. The reason is that time assumptions can be hard to substantiate in highly distributed systems like the Internet. Furthermore, these assumptions can often be attacked. This thesis is based on a different time model, a partially-synchronous model. We consider most of the system to be asynchronous, therefore we do not make time assumptions that may affect the safety of the applications (we make some for liveness). However, the TTCB is synchronous, i.e., it is a real-time subsystem that executes its services in a limited and known interval of time. The innovation is that partial synchrony is based on the architectural hybridization of the system: the TTCB is the support of our partially-synchronous model.

Group communication is a well known paradigm for data transmission among distributed sets of hosts or processes (Birman & Joseph, 1987b). This paradigm has been successfully used to support a large range of fault-tolerant applications, from

database replication to highly available web servers (see, e.g., (Birman, 1997)). Recently a few intrusion-tolerant group communication systems appeared (Reiter, 1995; Kihlstrom *et al.* , 2001; Moser *et al.* , 2000). As a final contribution, this thesis proposes a set of protocols for an intrusion-tolerant group communication system based on our hybrid fault model and on our partially-synchronous model. The basic aim was to contribute proof-of-concept demonstrators of the ideas just described and the innovation lied on two aspects: (1) new algorithm design methods on hybrid distributed system models, namely encompassing the difficulty of dealing with dual space-time realms (the "normal" or payload system's and the TTCB's); (2) new protocols with Byzantine resilience exhibiting high performance.

1.2 Contributions

Summarizing, the contributions of this thesis are the following:

- principles of modeling distributed systems subject to both accidental and malicious faults, under a hybrid perspective with regard to faults and synchrony.
- The concept of architectural hybridization and its illustration through the architecture of a distributed secure and real-time subsystem, the Trusted Timely Computing Base.
- New algorithm and protocol design methods using architectural-hybrid system models, illustrated through the design of intrusion-tolerant versions of two classical distributed systems protocols, reliable multicast and consensus.
- The design of an intrusion-tolerant group communication system based on the fault and time models mentioned. More precisely, the thesis complements the above-mentioned reliable multicast and consensus protocols with a membership service and a view-synchronous atomic multicast primitive, which are the main components of that system from the user perspective.

- A proof-of-concept implementation of the TTCB subsystem using COTS components[4], and its evaluation supporting the protocols mentioned above.

1.3 Structure of the thesis

Chapter 2 gives the context in which the thesis appears and presents related work in the literature.

Chapter 3 presents the hybrid fault model on which the TTCB is based. The chapter presents the TTCB model, architecture and services. It also presents the design of a TTCB based on COTS components.

Chapter 4 shows the implementation of a classical distributed systems protocol – reliable multicast– using our hybrid fault model. This is the first illustration of how our model can be used to implement intrusion-tolerant protocols. The TTCB is basically used to deliver a reliable digest of a message.

Chapter 5 gives the implementation of another classical distributed systems protocol, consensus. This is a more complex protocol that is normally affected by the FLP impossibility result (Fischer *et al.* , 1985). The chapter shows the relation between FLP and our system, using the results in (Dolev *et al.* , 1987).

Chapter 6 takes another step in terms of complexity. It presents the design of an intrusion-tolerant membership service. This is a core service of any group communication service, since it supplies communication protocols with the information about who are the members of a group at an instant. Failure detection is also discussed.

In Chapter 7 the thesis takes a final step towards the definition of an intrusion-tolerant group communication system. The chapter augments the membership service with a view-synchronous atomic multicast protocol.

[4]Commercial-off-the-shelf (COTS) components are hardware and software components available commercially. These components usually do not have the dependability required by the systems that use them.

Chapter 8 concludes the thesis and discusses some future work.

2

Context and related work

This chapter presents the context in which the thesis appears. The document proposes an architecture and protocols for Intrusion Tolerance, therefore the chapter starts with an introduction to this discipline. The relations with Dependability and Security are discussed. The next two sections introduce two central ideas of the thesis: hybrid failure assumptions and time models. Then, context about intrusion-tolerant communication protocols is provided. Details on specific protocols, like reliable multicast and consensus, are left to the chapters where protocols of those types are presented. Finally, the middleware architecture of project MAFTIA is presented, since it provides a framework in which the protocols of the thesis fit.

2.1 Intrusion tolerance concepts and mechanisms

Three concepts stand in the core of Dependability: fault, error and failure (Laprie, 1991; Avizienis *et al.* , 2001). A system provides a service which is *correct* if it follows the system specification. Dependability aims at avoiding the system *failure*, i.e., a behavior of the system that deviates from its specification. The failure is caused by one (or more) remote event(s) called *fault(s)*, e.g., a bug in a program or a configuration problem. An activated fault leads to an *error*, an erroneous state of the system which is liable to lead to failure. If nothing is done the error can become visible at the system interface, a system failure.

A fault can be internal, i.e., the failure of a component of the system, or external,

caused by the system environment (e.g., electromagnetic radiation, human interference). Faults can be classified in several types according to different criteria but one is especially important here: in relation to intent, faults can be either *accidental* or *intentional*. In the context of this thesis we consider only *malicious* intentional faults.

The way in which a component fails is designated its *failure mode* (Powell, 1992). Failures can be omissive (the component does not do an action when it was supposed to do) or assertive (the component does an action in a manner not specified). *Omissive failures* can be classified in timing failures (late or early interactions), omission failures (missing interactions), and crash failures (stop interacting). *Assertive failures* can be syntactic (incorrect format) or semantic (incorrect meaning). *Arbitrary failures* are the encompassing category: failures that can be simultaneously omissive and assertive.

From the point of view of a system, the failure of one of its components is a fault. Therefore, we can also talk about arbitrary faults, omissive faults, etc. Malicious faults can be simultaneously omissive and assertive. It is hard to predict or to put constrains on malicious faults, therefore they are usually described as arbitrary faults. A particular kind of arbitrary faults are *Byzantine faults* after a classical paper that studied the 'Byzantine generals problem' (Lamport *et al.* , 1982). In consequence, *tolerance to Byzantine faults* and *Byzantine resilience* are expressions with a similar meaning in Intrusion Tolerance: countering a certain hardness of behavior, prototypical of a hacker, such as sending different messages with the same identifier and/or forging messages. In this thesis we use interchangeably the terms Byzantine and arbitrary fault/failure.

2.1.1 AVI composite fault model

Security hazards are normally classified in three broad categories –vulnerability, attack and intrusion– which are related to the notions of fault-error-failure through the AVI composite fault model, developed in the MAFTIA project (Veríssimo *et al.* , 2000a; Adelsbach *et al.* , 2002; Veríssimo *et al.* , 2003). The model is presented in Figure 2.1. A *vulnerability* is a fault in the system that can be exploited with malicious intent. A

vulnerability can be a design fault (created during the system development) or an operational fault (introduced during system execution). A typical example is a software bug, like a missing array boundary test, which can be exploited using a buffer overflow attack [1]. Another example is a privileged account with a guessable password. These faults are usually accidental but can also be malicious, like in the case of a trapdoor left with malicious intention.

An *attack* is a malicious operational interaction fault performed with the objective of exploiting one or more vulnerabilities. Examples are port scans and hacker attempts to guess passwords.

The event of an attack managing to exploit a vulnerability is called an *intrusion*, a malicious operational fault. An example intrusion happens when a hacker manages to guess a privileged account password, and then penetrates the system. This intrusion causes an *error* (the hacker logged in) which can later cause the system *failure* (e.g., a server in the host ends-up with defaced web pages).

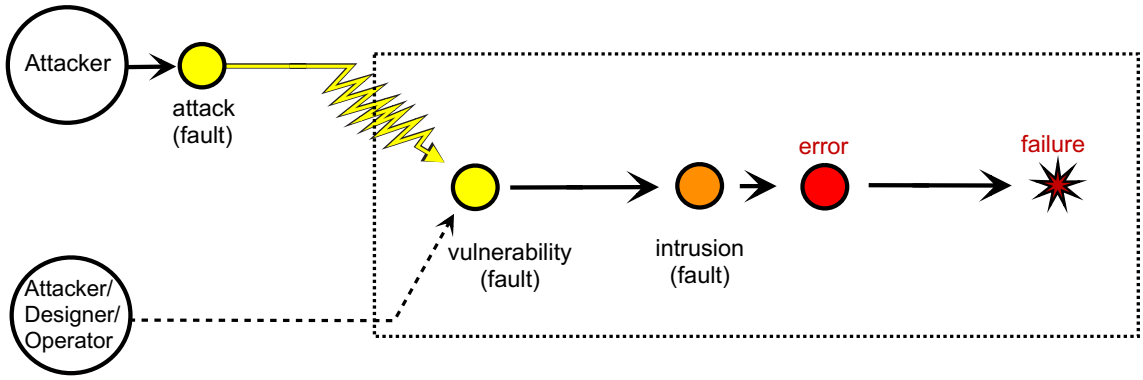

Figure 2.1: The AVI composite fault model.

The concepts of error and failure are related to the notion of *security policy*. This

[1] The thesis mentions a few well-known attacks. No references are provided since the former are not the outcome of scientific work, there are no obvious references and descriptions can be found in many reference books on security.

notion, which does not have a consensual definition, is central in security (Adelsbach *et al.* , 2002). Informally, the security policy of a system can be a statement of the security requirements of a system. Examples of items in a security policy are the minimum length of passwords and the machines that can be remotely accessed by a class of users. The notion of security policy can be refined in terms of security goals and security rules (Adelsbach *et al.* , 2002). *Security goals* are high-level statements of the security properties that the system must guarantee. The violation of a security goal is a failure of the system. *Security rules* are lower level statements of constrains that the system should satisfy so that the security goals are not violated, therefore the rules are usually stronger than the goals. The violation of a security rule is an error, which can directly or indirectly lead to the failure of the system.

The AVI composite fault model is a specialization of the fault → error → failure sequence. It describes the mechanism of intrusion in terms of three kinds of faults and the mechanism that can lead to failure: vulnerability + attack → intrusion → error → failure. The model gives insight into the mechanisms that can be used to tolerate these faults avoiding failure, i.e., on how to ensure the security properties.

2.1.2 Properties

Computing systems can be characterized in terms of several properties like functionality, usability, performance, cost and dependability (Avizienis *et al.* , 2001; Adelsbach *et al.* , 2002). Dependability itself has several attributes or properties: availability, reliability, safety, confidentiality, integrity and maintainability. Security shares three of these properties: availability, confidentiality and integrity (Adelsbach *et al.* , 2002). These dependability and security properties are also intrusion tolerance properties so we define them here. Special emphasis is put on the security properties.

Availability is the readiness of the system to provide a correct service. Attacks against availability are often designated Denial of Service attacks (DoS). This property is about 'something good' happening, on the contrary to the other security properties,

which are about 'bad things' not happening.

Confidentiality (or secrecy) is the absence of disclosure of information by unauthorized users. Disclosure of information is hard to detect so it is usually prevented/tolerated using *cryptography* (Menezes *et al.* , 1997, Chapter 1).

Integrity is the absence of invalid system state alterations. The expression "system state" should be taken very generically. It may include data and code in a host, messages in the network, and hardware configuration. Invalid alteration can be due to accidental faults (e.g., electromagnetic noise corrupting a network packet) or malicious faults (e.g., a hacker corrupting the packet).

Authenticity is the property of data being "genuine". If a document or a network message identifies its author/sender, it is authentic if it was truly authored/sent by that entity.

Several other properties are defined in the context of security. For instance, *non-repudiability* is the property of the author/sender of a piece of data not being able to deny that he authored/sent it.

Some additional dependability properties are the following. *Reliability* is the property of the service delivered being correct. *Safety* is the absence of catastrophic consequences of a failure of the system to its users or environment. *Maintainability* is the property of a system being able to undergo repair and reconfiguration.

2.1.3 Mechanisms

Building a dependable computing system, as also an intrusion-tolerant system, requires the combination of four types of techniques and mechanisms: fault prevention, fault tolerance, fault removal and fault forecasting (Laprie, 1991; Avizienis *et al.* , 2001). The threats against dependability (its impairments), the dependability attributes or properties, and the mechanisms or means, are depicted in the classical *dependability tree* in Figure 2.2.

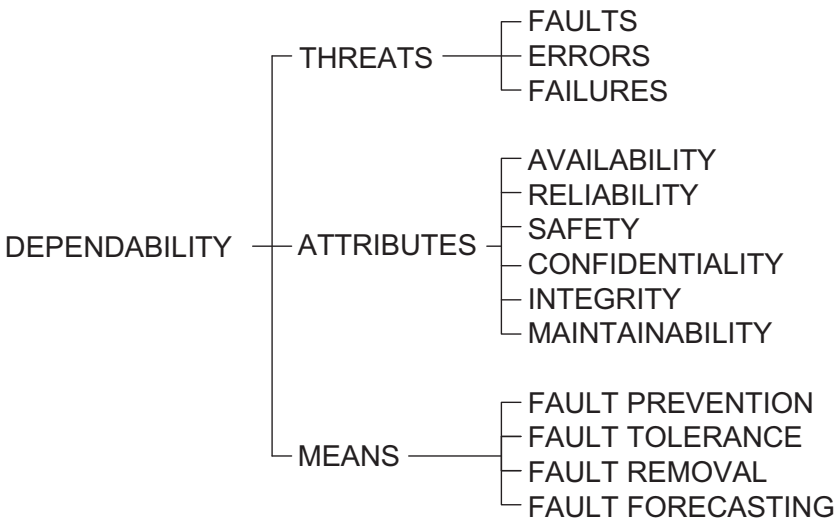

Figure 2.2: The dependability tree (Adelsbach *et al.* , 2002; Avizienis *et al.* , 2001).

Fault prevention aims to impede the occurrence or introduction of faults in the system. Quality control techniques in the design of hardware and software are typical fault prevention examples. Firewalls are fault prevention devices which aim to block intrusions in subnetworks.

Fault tolerance intends to make a system continue to deliver a correct service despite the actual presence of faults. Fault tolerance involves both error processing and fault treatment. *Error processing* can be performed with error detection and recovery (e.g., intrusion detection and countermeasures) or with error compensation (e.g., error masking using majority voting). *Fault treatment* aims to avoid that faults are reactivated. It encompasses fault diagnosis (identifying the cause) and fault isolation (exclusion of faulty components).

Fault removal during the system development involves verification of dependability properties (including validation of the specification) and the diagnosis and correction of problems. During the operational life, fault removal is made in the context of system

maintenance.

Fault forecasting involves the evaluation of the history of fault occurrence and activation in the system. This evaluation can be qualitative (identification of the faults, components where they occur, environmental causes) and quantitative (probability, how dependability attributes are affected).

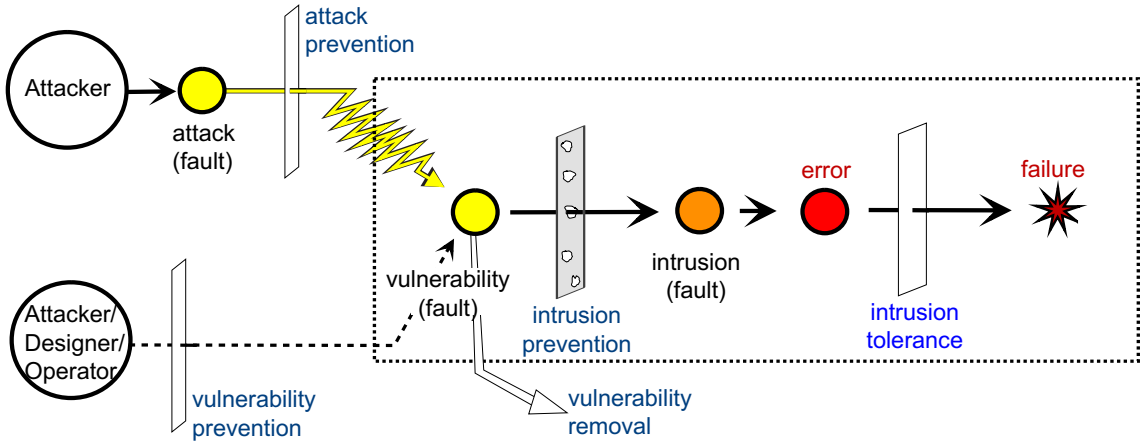

Figure 2.3: Intrusion tolerance mechanisms and the AVI model.

The AVI composite fault model in Figure 2.3 shows how a combination of these techniques can be used to handle security hazards (Veríssimo *et al.* , 2000a; Adelsbach *et al.* , 2002; Veríssimo *et al.* , 2003). Let us take as an example a distributed service implemented by a set of servers interconnected by a LAN. Let us assume also that clients make requests to the service through the Internet.

A first step to protect this setting is to prevent some attacks from occurring. *Attack prevention* includes inserting a firewall between the system and the Internet (to filter incoming traffic) and physically locking the system in a room. These techniques are not enough since, e.g., the firewall does not obstruct attacks directed to the ports of the service themselves.

On the side of vulnerabilities, the first set of techniques to apply is *vulnerability*

prevention using adequate design practices. *Vulnerability removal* includes 'patching' the system (introducing software corrections), disabling unused network services, and shadowing password files.

All these mechanisms do *intrusion prevention* but some attacks may still be able to activate vulnerabilities causing an intrusion, therefore we still need *intrusion tolerance*. All the fault tolerance techniques indicated above can be used. Error detection can be performed by an Intrusion Detection System (Denning & Neumann, 1985; Denning, 1987). Countermeasures include disabling ports or filtering the malicious traffic in a firewall. Fault treatment might include removing corrupted servers or creating new firewall rules. Error masking can also be made using state machine replication.

Intrusion Detection Systems (IDSs) (Denning, 1987; Debar *et al.* , 1999; McHugh, 2001) are security devices much used in private networks nowadays. IDSs have generically the objective of detecting attacks/intrusions, although some also perform countermeasures. IDSs can be classified in knowledge-based and behavior-based. The former have a database that describes known attacks and intrusions, and use that information to perform the detection. The latter have data about what is normal or expected behavior, and detect deviations from it. IDSs can also be host-based (detect intrusions in the host) or network-based (detect malicious activity in a network). IDSs suffer from two problems: false negatives and false positives. A false negative (or miss) is the omission of the detection of an intrusion, e.g., because there was no information about it in the (knowledge-based) IDSs database. False positives (or false alarms) are alarms that do not correspond to a real attack/intrusion. Current IDSs typically generate large quantities of these alarms with the hope of not missing real malicious activity, but with the inconvenient of making the life of the system manager difficult.

The *state machine replication approach* is especially important in distributed systems fault tolerance (error masking) so let us describe it briefly (Lamport, 1978; Schneider, 1990). The approach is based on a client-server model in which the server has several replicas. All replicas start with the same state, and the state changes are deterministic. The clients' requests are delivered to all (non-failed) replicas in the same order. There-

fore, after executing the same request all replicas have the same state and provide the same result if requested. This trivially tolerates replica crash faults and a voting of the results returned can be used to handle arbitrary faults. Some systems based on this approach are presented later.

Fault forecasting is not shown in the figure because it can and should be done for all the faults considered. For instance, the incidence of attacks and intrusions, and their effect on the correctness of the service, should be assessed periodically. The results of this analysis could be used, for instance, to reconfigure the system with more servers.

Cryptography has been used in the last few thousand years to protect information from disclosure (Menezes *et al.* , 1997, Chapter 1). In modern computer security, cryptography is an indispensable omnipresent building block. Classical cryptography, or *symmetric-key cryptography*, uses a key to encrypt and decrypt data. If the key is known only by who encrypts and who decrypts the data, it is 'hard' for a third entity to discover the text (the data) having only the ciphertext (encrypted data) and the encryption/decryption algorithm. Cryptography was revolutionized in the 70s with the notion of *public-key cryptography* (Diffie & Hellman, 1976), and the first encryption algorithm of this new kind, RSA (Rivest *et al.* , 1978). Public-key cryptography uses pairs of keys *(private, public)*. The process that generates a pair never discloses the private key, but provides the public key to other processes. Only the process with the private key can: (1) decrypt ciphertext encrypted with the public key; and (2) encrypt data that can be decrypted with the public key.

2.1.3.1 Fault independence

State machine replication is a technique for error masking in distributed systems. If one third of the replicas (or more) fail in a Byzantine way in a system with unbounded communication delays, the service can fail, i.e., deliver incorrect results (Castro & Liskov, 1999). This maximum number of faulty replicas (one third) can be increased by proactively recovering their state, e.g., by periodically resetting some replicas (Castro & Liskov, 2000). However, even using proactive recovery the limit of corrupted repli-

cas between recoveries remains the same. This bound requires fault independence, i.e., that intrusions in replicas occur independently. Achieving this independence is far from trivial because replicas may have the same vulnerabilities that may be exploited almost simultaneously with identical attacks.

The AVI model helps our reasoning here: independent intrusions should normally be the consequence of independent vulnerabilities, since it is difficult to substantiate assumptions about the independence of attacks. An approach to independent vulnerabilities is to have different replicas: all replicas should have different code, run on different operating systems, have different root and user passwords, etc. A solution for the implementation of different replica code is software diversity obtained using N-version programming (Avizienis, 1985; Lyu *et al.* , 1992). The approach consists in making independent implementations, perhaps in different languages, of software with the same functionality. Code obfuscation is a technique which tries to make reverse engineering of code hard for the attacker (Hohl, 1998; Collberg *et al.* , 1998b; Collberg *et al.* , 1998a; Wang *et al.* , 2001). Code obfuscation can be parameterized giving different implementations of the same program. A powerful solution for vulnerability independence would be to have encrypted executable code, however only a preliminary solution for polynomial functions is available (Sander & Tschudin, 1998).

2.1.4 Coverage and trust

A system is dependable if it does not fail. However, in practice the probability of failure cannot be reduced to zero. This reality is caught by the notion of *assumption coverage* (Powell, 1992; Veríssimo *et al.* , 2003).

A given system, its environment, and the accompanying mechanisms and protocols, imply the set of *possible* failures ($\mathcal{P}$), in the universe of *all* failures ($\mathcal{U}$): all that can possibly go wrong, even if with the tiniest probability (Figure 2.4). On the other hand, by assuming a given behavior of the system we stipulate *assumed* failure modes ($\mathcal{A}$).

We would wish that $\mathcal{A} = \mathcal{P}$. Unfortunately, this may be too expensive to accom-

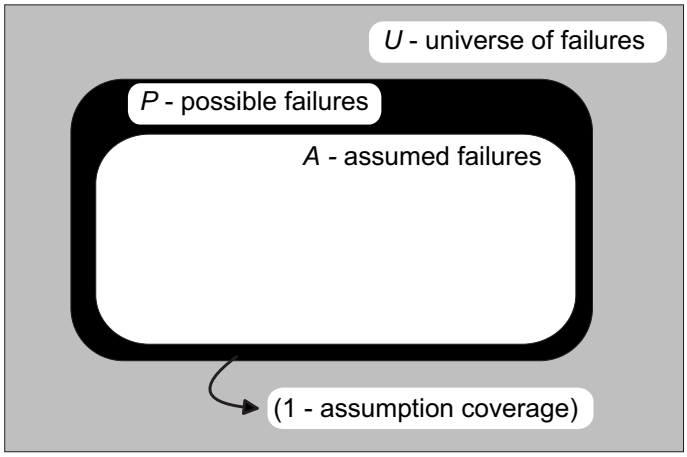

Figure 2.4: System/component assumption coverage.

plish, apart from the fact that $\mathcal{P}$ is seldom well defined. That is, we have in fact strictly $\mathcal{A} \subset \mathcal{P}$. This is normally satisfactory, if $\mathcal{P} - \mathcal{A}$ has an acceptably low probability of occurring. As shown in the figure, the assumption coverage is the complement of that probability: the probability of, given a failure, it not occurring in the region $\mathcal{P} - \mathcal{A}$. Assessing the assumption coverage of a design consists of verifying whether this probability is acceptably low. This assessment follows a separation of concerns between environmental and operational assumptions (Powell, 1992).

Environmental assumptions are assumptions about the expected behavior of the environment where the system will run (network, hardware, attackers,...). The *environmental assumption coverage* (Pr_e) is the probability that the environmental assumptions $\mathcal{H}$ hold in the presence of failures, i.e., it is the conditional probability of the set of assumptions $\mathcal{H}$ holding when any failure f occurs: $Pr_e = Pr(\mathcal{H}|f)$.

On the other hand, *operational assumptions* concern the behavior of the system, i.e., how the system will run given the environmental assumptions. The *operational assumption coverage* (Pr_o) gives the probability of the system properties $\mathcal{S}$ holding, considering that the environmental assumptions hold: $Pr_o = Pr(\mathcal{S}|\mathcal{H})$. In the context of intrusion

tolerance, Pr_o gives the probability that the system resists the fault it is assumed to handle (e.g., it tolerates the faults it should tolerate). If the algorithm(s) and its implementation are proven correct we expect a coverage $Pr_o = 1$, except when the algorithms themselves are probabilistic.

The overall *assumption coverage* gives the probability that the system tolerates a fault:

$$Pr(\mathcal{S}|f) = Pr_o \times Pr_e = Pr(\mathcal{S}|\mathcal{H}) \times Pr(\mathcal{H}|f) \tag{2.1}$$

This formula assumes that any violation of the environmental assumptions leads to the system failure, since it takes into account the complete term Pr_e. This is not always true but is, nevertheless, a common assumption (Babaoğlu, 1987; Powell, 1992). It makes even more sense in the context of malicious faults, which are caused by an intelligent being with the intention of provoking the system failure.

In security, the notion of *trustworthiness* gives the extent to which a component or system meets a set of security properties. If we generalize this notion to the other dependability properties then trustworthiness becomes essentially synonymous of dependability.

The notions of *trust* and trustworthiness are related. Trust is the accepted dependence of a component on the properties of another component or system. A component A *trusts* a component B if A relies on a set of properties of B in order to provide a correct service. For instance, A provides a correct service if the cryptographic primitives provided by B satisfy the property of integrity. This notion can be generalized to include not only malicious but also accidental faults. A trusts B if A makes a set of assumptions about the way in which B can fail generically, not only in terms of security properties.

The notion of trust is a means of separation of concerns. When a programmer designs a component A relying on B (e.g., a runtime environment), he can *trust* B to have a certain set of properties and focus on the design of A. Building B to be

trustworthy/dependable is a problem that can be dealt with separately. In other words, A makes assumptions about B to substantiate A's correct operation. On the other hand, the way in which B is built to substantiate those assumptions concerns another design step, probably, and recursively, by trusting some other underlying component C. Or alternatively, B might be a COTS, built by other designers. Naturally, a component A should trust another component B only to the extent of B's trustworthiness. If B is not as trustworthy as A trusts it, the dependability of A that is, the trustworthiness of A, is compromised. This gap can be caught by the notion of assumption coverage.

2.2 Hybrid failure assumptions

A crucial aspect of any fault- or intrusion-tolerant system is the type of fault model (or set of failure assumptions) upon which the system architecture is conceived and component interactions are defined (Veríssimo *et al.* , 2000a; Adelsbach *et al.* , 2002).

Controlled failure assumptions specify qualitative and quantitative bounds on component failures. For example, the fault model can specify that there are only crash failures (e.g., crash of hosts) or omission failures (e.g., loss of packets in the network). This approach can be realistic since it represents well how common systems work under the presence of accidental faults only. However, it is very difficult to model the behavior of an attacker, so specifying bounds on component failures in the presence of malicious faults can create a problem of assumption coverage.

Arbitrary failure assumptions, on the other hand, specify no qualitative or quantitative bounds on failures of the components. However, bounds on the number of components of the system which can fail are inevitable, e.g., "only less than one third of the components can fail". Arbitrary failure assumptions are usually costly to handle, both in terms of performance and algorithm complexity.

Hybrid failure assumptions combine controlled and arbitrary failure assumptions. The idea is to allocate different assumptions to different subsets of components of the

system. A number of systems and protocols based on this type of model can be found in the literature (Meyer & Pradhan, 1987; Powell *et al.* , 1988; Lincoln & Rushby, 1993; Veríssimo *et al.* , 1997).

With hybrid assumptions some parts of the system can be assumed to exhibit fail-controlled behavior, whilst the remainder of the system is still allowed to fail arbitrarily. This is advantageous in modular and/or distributed system architectures subjected to malicious faults. However, this approach is only feasible when the fault model is well substantiated, that is, when the behavior assumed for every single subset of the system can be modeled and/or enforced with high coverage. As a matter of fact, a system normally fails by its weakest link, and naive assumptions about a component's behavior can be an easy target to attackers.

There is a body of research on hybrid fault models for consensus and diagnosis algorithms, assuming different failure type distributions for different participants (Meyer & Pradhan, 1987; Lincoln & Rushby, 1993; Walter *et al.* , 1994; Kieckhafer & Azadmanesh, 1995). For instance, some participants are assumed to behave arbitrarily while others are assumed to fail only by crashing. This kind of assumptions lies on evidence arising from statistical analysis of systems under accidental faults and as such is hard to substantiate under the presence of malicious faults like attacks performed by a hacker. The hybrid failure assumptions we are interested in, follow the lines of earlier works such as (Powell *et al.* , 1988; Veríssimo *et al.* , 1997). They might best be described as *architectural hybridization*, where failure assumptions are in fact *enforced* by the architecture and construction of the system components, and thus substantiated. This is case of the TTCB, the component that supports the architectural-hybrid model proposed in the thesis.

2.3 Time and synchrony models

Research in distributed systems algorithms has been based on several synchrony models (Veríssimo *et al.* , 2000a; Adelsbach *et al.* , 2002). In the two extremes of the

synchrony spectrum there are the asynchronous model and the synchronous model.

2.3.1 Asynchronous and synchronous time models

The *asynchronous model* is time-free, i.e., does not allow any time specifications and cannot address timed problems. Distributed systems based on this model typically have the following characteristics:

- Unbounded or unknown processing delays
- Unbounded or unknown message delivery delays
- Unbounded or unknown local clock drift rates

The *synchronous model*, on the other hand, assumes that the system has strict time properties:

- Known bound for processing delays by correct processors
- Known bound for message delay between correct processors
- Known bound for the difference and drift rate among local clocks

The synchronous model is not particularly useful for intrusion tolerance for two reasons. First, it is too strong for generic networks such as the Internet, especially the second property. Second, assumptions about time are frequently vulnerable to attacks, compromising the dependability of the system. All three properties can be vulnerable: processors, messages and clocks can, for example, be delayed by an attacker.

Although the asynchronous model is adequate to describe any network, its lack of time properties constrains its usability for many practical applications. Practical systems have some sort of limit of time to deliver their service.

2.3.2 Partial synchrony

In order to address the limitations of these two extremes, several intermediate models were developed. All these models consider a system that is essentially asynchronous but extend it in some way or make additional assumptions. The timed-asynchronous model (Cristian & Fetzer, 1998) extends the asynchronous model with local hardware clocks (with bounded drift) and assumes enough synchrony, e.g., to detect timing failures. The quasi-synchronous model (Veríssimo & Almeida, 1995) assumes that the synchrony bounds have a probabilistic nature, and is extended with a synchronous communication channel with limited bandwidth connecting the processors. The asynchronous model with failure detectors tries to provide the minimum degree of synchronism necessary to deterministically solve consensus in distributed systems with crash failures (Chandra & Toueg, 1996; Chandra *et al.* , 1996). This synchronism is abstracted by the notion of unreliable (crash) failure detectors.

These intermediate models can be designated as *partially-synchronous*. The expression was originally introduced in a study of the degree of synchronism necessary to solve consensus deterministically in the presence of crash failures (Dwork *et al.* , 1988). Communication was said to be partially-synchronous in two conditions: if there was a maximum but unknown communication delay Δ; or if there was a known Δ which would eventually hold from some instant on.

An important objective of these models is to circumvent the FLP impossibility result, which states that no deterministic protocol can solve consensus in an asynchronous system if a single process can fail by crashing (Fischer *et al.* , 1985). This result was further detailed to several situations (Dolev *et al.* , 1987): synchronous/asynchronous processors/processes, ordered/unordered message delivery, broadcast/point-to-point communication and atomic/not-atomic receive and send. This paper has shown that different degrees of synchronism allow deterministic protocols to tolerate different numbers of faults. FLP can also be circumvent by probabilistic protocols or using randomization (Rabin, 1983; Ben-Or, 1983; Bracha & Toueg, 1985) (see Chapter 5).

2.3.3 The Timely Computing Base

The Timely Computing Base (TCB) model provides architectural support for partial-synchrony models (Veríssimo *et al.* , 2000b; Veríssimo & Casimiro, 2002). The TCB is a synchronous distributed subsystem. Some hosts have a local TCB module and these modules are interconnected by a control channel. The TCB can be inserted in systems with unreliable timeliness, possibly asynchronous.

The TCB provides three services which can be used by the applications to perform time-related operations: timing failure detection, duration measurement and timely execution. The TCB assumes a benign fault model, i.e., accidental faults only. An implementation based on COTS components is available (Casimiro *et al.* , 2000).

The objective of the TCB is not to make applications synchronous or timely. Applications run mostly outside the TCB so their timeliness is constrained by the degree of synchronism of the 'outside' system. However, applications can use the TCB services to tolerate timing faults.

Several example applications supported by the TCB were proposed. A *fail-safe application* goes to a safe state when a fault occurs. The TCB supports both the detection of timing failures and the change to the safe state in a timely way, two operations that cannot be done in purely asynchronous systems (Veríssimo *et al.* , 2000b). There is a considerable literature on the adaptation of quality of service to a changing environment (Talley & Jeffay, 1994; Correia & Pinto, 1995; Bom *et al.* , 1998). These applications that adapt to timing failures have been designated *time-elastic* and the TCB can be used to guarantee that safety is satisfied when these failures occur (Casimiro & Veríssimo, 2001). Yet another class of applications – *time-safe* – can use the TCB to mask timing failures using replication (Casimiro & Veríssimo, 2002).

2.4 Intrusion-tolerant group communication

Group communication is a well known paradigm for fault tolerance in distributed systems (see, e.g., the recent survey (Chockler *et al.* , 2001)). This paradigm has been used successfully to support a large range of applications, from database replication to highly available web services (see, e.g., (Birman & Joseph, 1987a; Birman, 1997)).

Group communication systems (GCSs) provide reliable communication primitives between sets of hosts or processes. A GCS has two fundamental components: the membership service and the multicast (or communication) service.

The *membership service* is the component in charge of keeping a list of the group members (hosts or processes). This service processes request to join or leave the group, and removes faulty members.

The *multicast service* provides communication primitives. The basic multicast primitive is *reliable multicast*, also called *Byzantine agreement* in the context of malicious failures (Hadzilacos & Toueg, 1994; Lamport *et al.* , 1982). The problem can be informally defined in terms of two properties (Bracha & Toueg, 1985): all correct processes in the group deliver the same messages; if a correct sender transmits a message then all correct processes in the group deliver this message. Reliable multicast does not impose an order for messages to be delivered, therefore, there are several variants of it with different orders of delivery. *FIFO multicast* imposes FIFO order: if a process multicasts a message M before M' then all correct processes deliver M before M'. *Causal multicast* delivers messages according to the relation of potential causality. A message M precedes, or is potentially causally related to, a message M' (M $\rightarrow$ M') iff: (1) a process sends M' after M; or (2) M is delivered to the sender of M' before it sends M'; or (3) there is a message M'' such that M $\rightarrow$ M'' and M'' $\rightarrow$ M'. *Atomic multicast* delivers messages in *total order*: any two messages delivered to any two correct processes are delivered in the same order to both. These orders can be combined resulting in multicast primitives with stronger orders. An example is causal atomic multicast that delivers messages both in causal and total orders.

The *view-synchronous* semantics defines how the membership and the communication services interact. This semantics states, informally, that all correct group members deliver the same messages in the same group membership (Birman & Joseph, 1987b; Birman & Joseph, 1987a). Several variants of this semantics were defined, e.g., extended virtual-synchrony (Moser *et al.* , 1994) and weak and strong virtual-synchrony (Friedman & van Renesse, 1996) [2].

Most works in group communication consider only crash failures. A few examples based on the asynchronous model are Isis (Birman & Joseph, 1987b), Transis (Amir *et al.* , 1992), Totem (Moser *et al.* , 1996) and NavTech (Rodrigues & Veríssimo, 2000). An example for the synchronous model is (Cristian, 1991).

Some of these group systems have evolved to support a stronger model, which assumes that the communication can be attacked in the network, but hosts continue to be assumed secure. These systems provide a *secure islands* model (Reiter *et al.* , 1992) in which every process equally trusts another. This model is not intrusion-tolerant. It requires the operating system and the GCS software to be part of a Trusted Computing Base (National Computer Security Center, 1983), i.e., that intrusions in the hosts are prevented (Reiter *et al.* , 1994). If a process is malicious the secure group abstraction does not hold. For instance, the content of the communication can be disclosed, correct processes will not necessarily deliver the same messages, order properties can be violated, etc. Although not intrusion-tolerant, work based on the secure islands model handled important issues such as join authorization and key distribution, therefore we survey three of these systems below: Horus, Ensemble and Secure Spread.

More recently, interest emerged in GCSs for environments that might suffer arbitrary faults, i.e., intrusion-tolerant GCSs. The systems that we are aware of are only three: Rampart, SecureRing and SecureGroup. All these systems consider the asynchronous model. We present them below, without delving into the details of the pro-

[2] View synchrony was originally called virtual-synchrony since the objective was to give to some extent the idea that events occurred synchronously in all processes (Birman & Joseph, 1987a). This intuition did not fit well with a second generation of group communication systems that supported partitionable groups, therefore the more general definition and designation of view synchrony.

tocols that are left for the appropriate chapters of the thesis.

Finally we present a few intrusion-tolerant systems: Enclaves, BFT, and Phalanx/Fleet. They are not dynamic GCSs in the same sense as the others but they are distributed and therefore involve communication among groups of processes or hosts.

2.4.1 Horus

Horus, an extension of Isis, is the first work on security for GCSs that we are aware of (Reiter *et al.* , 1994; Reiter *et al.* , 1992). Horus assumes a *primary-partition model*, i.e., if two sets of processes become unable to communicate among them due to a network partition, one set is considered to be 'the group' while the processes in the other set are considered failed and removed. An attacker can view all network traffic and engage in any active network attack[3] and in any passive attack except traffic analysis[4]. This attack model in the network is virtually the same for all systems considered in the thesis. Horus makes two assumptions about the operating system: that it authenticates securely the user identifiers of local processes; and that it provides protected private address spaces and private authentic message passing between the local system and user processes. These assumptions are satisfied by common operating systems (OSs), like Unix, if the OS itself is not corrupted.

Horus security is implemented in two protocol layers: a transport layer (MUTS) whose objective is to provide reliable, sequenced multicast between sites; and a session layer (VSYNC) that implements the process group and view synchrony abstractions on the top of MUTS. Horus uses several cryptographic keys. The site keys are asymmetric key pairs associated to every host and used for initial key distribution. The group keys

[3] *Active attacks* are attacks in which the attacker modifies transmitted data. They can be of several kinds. In a *masquerade* attack the attacker impersonates at least one of the communication entities. In a *replay* attack the attacker retransmits a previously transmitted valid message. *Modification* attacks involve the modification of a valid transmitted message. A *denial of service* attack consists in disrupting the communication (Menezes *et al.* , 1997, Chapter 1).

[4] A *passive attack* is an attack that does not interfere with the communication. A *traffic analysis attack* consists in getting information from traffic even if not being able to read it, e.g., because it is encrypted. The attacker can see the source and destination of messages, how many messages are sent, patterns of traffic, etc. (Menezes *et al.* , 1997, Chapter 1).

are also asymmetric key pairs that are used to authenticate groups (the public key is incorporated into the group address). The communication keys are symmetric keys used by MUTS to distribute the connection keys. The latter are symmetric keys shared by every pair of hosts and used to secure their communication. Keys are always kept exclusively in volatile memory in order to be deleted in case the host crashes. Keys are always the same during the group lifetime; there is no rekey protocol. Communication and connection keys are generated in background by a user level service, the group key service, that stores them in the VSYNC layer for future use. This takes the costly generation of keys away from the group creation sequence of operations, reducing the delay of this operation.

The secure islands model requires that only trusted processes in trusted hosts join groups. This is achieved with a secure join protocol. The most important task of this protocol is to authenticate the process attempting to join, but the joining process has also to authenticate the group. Authentication is often used to distribute a symmetric key to encrypt the session communication but this is not the case with Horus. When a process tries to join a group it is identified by its site address and the process owner. Authorization to join is granted by the group members considering these two pieces of information. On the other hand, a joining process verifies the authenticity of the group by challenging the contact site to prove the possession of the private key of the group.

Horus uses an hierarchical key distribution scheme. It uses a trusted authentication service to distribute the public keys that correspond to each site's private key (site keys). A preliminary design of the authentication service was presented in (Reiter & Birman, 1994). A site in which a process creates a group uses those keys to distribute a communication key for the group and the private group key to the new members. Every site in which a process needs to send a message to the group creates a connection. The connection key is given to the other group sites using the communication key. The connection key is used to secure regular communication.

To secure the communication inside the secure island the first step needed is to authenticate messages sent. The objective is to detect attempts to insert, change and

replay valid group messages, or to impersonate a group member (there can also be the need for secrecy which is achieved with conventional message encryption). This detection can be used to discard messages modified or forged. In Horus message authentication and encryption is handled by the MUTS layer. Messages carry a message authentication code (MAC) obtained using a hash function whose result is encrypted with the connection key. A hash function is basically a one-way function that compresses its input and produces a fixed sized digest (e.g., 128 bits). Horus makes the usual assumption that an attacker is unable to subvert the cryptographic properties of the hash function, such as weak and strong collision resistance (Menezes *et al.* , 1997, Chapter 9). The message is also encrypted with the connection key, in case privacy is requested. Replays are detected because each message carries a unique sequence number.

2.4.2 Ensemble

Ensemble is an evolution of Horus, which also provides the secure islands abstraction (Rodeh *et al.* , 2001a). Ensemble security extends Horus mainly in two aspects: it handles group partitions and the associated security issues; and it has a rekeying protocol executed whenever a member joins or leaves a group. This protocol is necessary in order to guarantee the confidentiality of the communication before the join and after leaving, in relation to the new/past members. An efficient rekey protocol is described in (Rodeh *et al.* , 2001b).

In Ensemble, communication is mostly encrypted and signed using a symmetric encryption key shared by all group members, the group key. Every member has also an asymmetric key pair used for authentication and for signing and encrypting messages in special cases. A situation in which these pairs are used occurs when two partitions need to merge. They have to use these keys because they may not have the same group key.

Ensemble allows the definition of different communication semantics by com-

posing protocol layers in stacks. Ensemble security is implemented by three layers, from bottom to top: Encrypt, Rekey and Exchange. The Encrypt layer optionally encrypts/decrypts user messages using the group key. The Rekey layer handles the distribution of cryptographic keys. The Exchange layer handles security when there are several group components due to network partitions. When a partition occurs Ensemble elects a new leader for each "rebel" partition and each component starts operating independently. Ensemble tries to detect the reestablishment of communication using gossip messages, which are multicasted periodically. Whenever possible, the components are merged back into a single group. In the general case, different components have different group keys and so they cannot communicate (except with gossip messages). When a merge occurs, Ensemble rekeys some of the components so that they all start to share the same group key.

2.4.3 Secure Spread

Spread security (Amir *et al.* , 2000; Amir *et al.* , 2001) differs from Ensemble and Horus mostly in the following aspects:

- Security in Horus and Ensemble is integrated in the GCS stack of protocols. On the contrary, Spread security is implemented as a layer on the top of the other GCS layers. This allows some flexibility in the implementation, which can be integrated either in the Spread daemon or in a separate library (Amir *et al.* , 2000).
- Horus and Ensemble's key generation is centralized. Spread's is decentralized: all group members collectively generate the group key.

Secure Spread's key management uses project Cliques's protocol suite for group key agreement (Amir *et al.* , 2000; Steiner *et al.* , 2000; Ateniese *et al.* , 2000). This means that keys –shared secrets– are cooperatively generated by several group members as a function of information provided by each of these members. Cliques is contributory in the sense that each member equally contributes to the key. Every group member is

equally trusted. Cliques is an extension of the Diffie-Hellman key exchange technique in which two processes agree on a shared key, without sharing any previous secret and with an attacker possibly watching every message exchanged (Diffie & Hellman, 1979). Cliques provides initial key agreement and auxiliary key agreement protocols. The latter are more efficient but require a previously shared key. Communication in Secure Spread is protected in a similar way to Ensemble, i.e., using the group key.

There is some additional interesting work on group key management in the context of Internet multicast. Special emphasis is put in the scalability of key distribution for a 'large' number of processes (Wong *et al.* , 1998; Gong, 1994; Canetti *et al.* , 1999; Mittra, 1997; Hardjono *et al.* , 1999).

2.4.4 Rampart

Rampart is an intrusion-tolerant GCS (Reiter, 1995; Reiter, 1996a). It tolerates attacks and intrusions in the network and hosts. The motivation for Rampart was the need in Horus for high integrity services, which had also to be highly available (Reiter *et al.* , 1994). The objective of Rampart is precisely to support the implementation of this kind of services using the state machine replication approach.

Rampart has a membership protocol, which handles group joins and leaves, and removes failed processes from groups (Reiter, 1996b). The service is implemented by a three-phase commit style protocol. Processes in a group send failure suspicions to a leader that tries to change the membership when it received enough. The sender uses digital signatures (Menezes *et al.* , 1997, Chapter 11) to prove that it received the suspicions. These signatures are calculated with a public-key cryptography algorithm (e.g., RSA) which is the protocol performance bottleneck.

Rampart relies on two basic assumptions: (i) every server has a private key and all other servers the corresponding public key; (ii) there is a message transport facility that provides a point-to-point, reliable authenticated communication channel between each pair of servers. This channel guarantees that if both the sender and the recipient

are correct, the recipient eventually delivers the message sent. This transport layer is the Horus MUTS layer.

Rampart provides a reliable multicast primitive based on a simple echo protocol (Reiter, 1994). The sender multicasts the message which is 'echoed' by the recipients with their signature. When the sender has enough echoes is multicasts a commit message with the signatures it received. The objective of Rampart is to implement state machine replication, which requires atomic multicast. Rampart implements this protocol on the top of reliable multicast. Messages are ordered by a process of the group, the sequencer, which reliably multicasts the ordering to the others (Reiter, 1994).

A replicated service in Rampart is implemented as a group of servers (Reiter, 1995). Clients are outside the group so they cannot atomically multicast to the replicas directly. Therefore, clients send requests to one member of their choice, which forwards these requests to all members using atomic multicast. This allows a good throughput but a malicious server can deny the service to a client or corrupt the request. Both problems have to be detected by the client and a new server chosen. The first problem requires the use of timeouts, while the second can be solved by authenticating requests at the application layer.

The output of the service has to be voted so that the results provided by correct servers prevail over those returned by malicious servers. Rampart implemented two solutions. In the first, the client receives individual results from the servers and performs the voting. In the second, the voting is executed by the servers using a (k,n)-threshold signature scheme (Reiter & Birman, 1994; Desmedt & Frankel, 1989). This scheme generates a public key and n shares of the corresponding private key. Each share can be used to obtain a partial signature of a message and any k of those partial signatures form a full signature that can be verified using the public key. This scheme is intrusion-tolerant but has poor performance (Reiter, 1995).

Rampart's protocols were re-implemented more recently in the context of project ITUA with the objective of measuring the performance costs of intrusion tolerance (Ramasamy *et al.* , 2002).

2.4.5 SecureRing and SecureGroup

Two intrusion-tolerant GCSs were designed at UCSB Computer Networks & Distributed Systems Lab.: SecureRing and SecureGroup. SecureGroup has approximately the same performance in the presence and absence of attacks. SecureRing performs better when there are no attacks.

SecureRing provides reliable totally ordered delivery of messages in the presence of arbitrary faults (Kihlstrom *et al.* , 2001). The network is assumed not to partition so persistent disruption of the communication is handled as a host failure. SecureRing is inspired in the Totem single-ring protocols (Moser *et al.* , 1996).

SecureRing is designed for LANs and relies on a logical ring imposed on the communication medium, which controls the multicasting of messages. The system is composed of a message delivery protocol, a membership protocol, a Byzantine failure detector, and a message diffusion protocol. The message delivery protocol is used to deliver regular messages (application data) and configuration change messages (information about changes in the membership) in total order. Only the host with the token can multicast. The token, which is signed, is used to distribute digests of the messages (which therefore do not need to be signed, with performance benefits). The membership protocol reconfigures the system when one or more hosts exhibit detectable Byzantine failures. When failures are detected the protocol forms a new ring with the hosts that seem to be correct. Detectable Byzantine failures are detected using an unreliable Byzantine failure detector (Kihlstrom *et al.* , 2003). The message diffusion protocol is used to broadcast special messages, e.g., for a host that needs to join to send a request to the group. The protocol is basically a reliable broadcast protocol, i.e., it guarantees that if a correct process delivers a message then all correct processes deliver it.

SecureGroup is the other intrusion-tolerant GCS designed at UCSB (Moser *et al.* , 2000; Moser & Melliar-Smith, 1999), this one inspired in the crash-tolerant Trans/Total suite of protocols (Melliar-Smith *et al.* , 1990). SecureGroup is also suited for LANs and

provides totally ordered delivery of messages.

The system is composed of three protocol layers, from bottom to top: Secure Trans, Secure Total and Secure Membership. Secure Trans reliably broadcasts messages to all processors. It uses a combination of piggybacked positive and negative acknowledgments to avoid the need for a separate acknowledgment from every recipient. Messages are digitally signed. Secure Total totally orders the messages delivered by Secure Trans without additional transmission of messages. Secure Total goes on ordering and delivering messages even if there are failed, possibly malicious, processors. This is a difference in relation to SecureRing and Rampart, which have to remove failed processors from the membership. The Secure Membership protocol works on the top of Secure Total therefore it is simpler than other similar protocols.

2.4.6 Enclaves

Enclaves is a middleware for 'group-oriented' applications. Enclaves is not a typical GCS in the sense of the systems above because it does not provide strong communication semantics, like reliable multicast, atomic multicast and view synchrony. The multicast primitive provided is best-effort, i.e., there are no guarantees of delivery. The purpose of Enclaves is not to support reliable services but collaboration between humans, assuming that they can tolerate some message losses. The initial versions of Enclaves aimed to secure the communication but relied on a leader so they were not intrusion-tolerant (Gong, 1997; Dutertre *et al.* , 2001). The discussion below is about the current intrusion-tolerant version (Dutertre *et al.* , 2002).

Enclaves has special processes called leaders. These leaders communicate over an asynchronous network and can fail arbitrarily. The system tolerates f malicious leaders out of a total of $3f + 1$. The leaders are in charge of performing the group and key management operations. Any modification to the group membership requires an agreement between the nonfaulty leaders.

Normal members (non-leaders) request to join and are removed by the leaders. The

group members share a group key. A new group key is generated whenever the membership changes. Intrusion-tolerant key generation is obtaining making each leader generate a share of the group key, using a scheme that allows the reconstruction of the key with $f + 1$ shares. A member joins the group contacting $2f + 1$ leaders, to which it remains attached, for instance to receive key changes.

2.4.7 BFT

BFT is a distributed error masking algorithm (Castro & Liskov, 1999; Castro & Liskov, 2000; Castro & Liskov, 2001). The objective is to support the implementation of efficient intrusion-tolerant services using state machine replication. A service is implemented as a set of servers which process the requests from the clients. The service tolerates f malicious servers out of $3f + 1$. BFT is not a full-fledged GCS since it does not have a membership service and does not provide generic group communication primitives.

The algorithm runs basically this way: a client sends a request to the primary/leader; the primary atomically multicasts the request to the backups (the other servers); all replicas execute the request and send the result to the client; the client waits for $f + 1$ replies with the same result, which is the result of the operation. The service does not tolerate faulty clients. Therefore, clients are authenticated and that authentication is used to control the access to shared information.

BFT has better performance than Rampart and SecureRing because it does not use digital signatures in normal operation. Instead, it uses message authentication codes (MACs), which are obtained with symmetric-key cryptography (Castro & Liskov, 1999). Symmetric-key cryptography operations are usually about three orders of magnitude faster than public-key cryptography operations.

2.4.8 Phalanx and Fleet

Quorum systems are an alternative to the state machine replication approach to implement fault-tolerant systems. Malkhi and Reiter were, to the best of our knowledge, the first to present a study of their application to tolerate arbitrary faults (Malkhi & Reiter, 1997a). They proposed several variants of the scheme (Malkhi & Reiter, 1997a; Malkhi *et al.* , 1997a; Malkhi *et al.* , 1997b; Alvisi *et al.* , 1999) and used it to build a dependable data repository that supports shared data abstractions: Phalanx (Malkhi & Reiter, 1998). They assume asynchronous communication and that any two correct processes can communicate over an authenticated and reliable channel (this can be obtained with mechanisms described in Section 2.4.1).

Quorum systems are quite different from state machine replication. In quorum systems only a subset of the servers has to receive the requests so these systems are more scalable and available than systems based on state machine replication. However, the applications for quorum systems are not the same. Phalanx provides data stores (read/write operations) and locks, not a generic service as BFT. A generic service can be implemented using reads/writes/locks but the rate of locks will probably be high and the system performance arguably low.

A quorum system, for a universe of servers, is a set of subsets of servers – *quorums* – in which each subset intersects with all others. Operations are done in just one quorum. The intersection of quorums guarantees that operations performed in different quorums are consistently seen across the system. Quorum systems tolerant to arbitrary faults need a stronger intersection: the intersection of each pair of quorums has to contain a minimum number of correct servers in order to guarantee consistency of the replicated data as seen by clients.

Fleet builds on Phalanx but provides support for generic objects instead of just read/write operations on variables (Malkhi *et al.* , 2001). Fleet provides support for persistent objects, i.e., objects that tolerate crashes and Byzantine faults of some of the servers that keep their state.

2.5 MAFTIA middleware

Malicious- and Accidental-Fault Tolerance for Internet Applications (MAFTIA) is a recently finished EU IST project. The project had the objective of systematically investigating the 'tolerance paradigm' for constructing large-scale dependable distributed applications. The project had a comprehensive approach that included both accidental and malicious faults. This section introduces the middleware of MAFTIA since it is the framework in which the protocols presented in the thesis were developed. We start with a brief introduction about the project.

2.5.1 MAFTIA project

MAFTIA project followed three main lines of action[5]:

- definition of an architectural framework and a conceptual model;
- the design of mechanisms and protocols;
- formal validation and assessment.

The first line aimed to develop a coherent set of concepts for an architecture able to tolerate malicious faults (Adelsbach *et al.* , 2002). Work has been done on the definition of a core set of intrusion tolerance concepts, clearly mapped into the classical dependability concepts. The AVI composite fault model presented above was defined in this context. Other relevant work included the definition of synchrony and topological models, the establishment of concepts for intrusion detection and the definition of a MAFTIA architecture. This architecture includes components such as trusted and untrusted hardware, local and distributed trusted components, operating system and runtime environment, software, etc.

[5]The work briefly sketched here is reported in a large number of reports, so no exhaustive references are provided. All MAFTIA reports can be found at http://www.maftia.org .

Most MAFTIA work was done in the context of the second line of action, the design of intrusion-tolerant mechanisms and protocols. The design of the MAFTIA middleware was part of this line, but we leave it for the next section. *Intrusion detection* was considered as a mechanism for intrusion tolerance but also as a service that has to be made intrusion-tolerant. MAFTIA developed a distributed intrusion-tolerant intrusion detection system. Problems like handling high rates of false alarms and correlating the alarms generated by several IDSs were also explored. *Trusted Third Parties* (TTPs), such as certification authorities, are important building blocks in today's Internet. MAFTIA designed a generic distributed certification authority that uses threshold cryptography and intrusion-tolerant protocols in order to be intrusion-tolerant. Another TTP, the distributed optimistic fair exchange service, was also developed. MAFTIA defined an *authorization service* based on fine grain protection, i.e., on protection at the level of the object method call. The authorization service is a distributed TTP that can be used to grant or deny authorization for complex operations, combining several method calls. The service relies on a local security kernel, e.g., a JavaCard.

The third line of work was on the formalization of the core concepts of MAFTIA and verification of the dependable middleware. A novel rigorous model for the security of reactive systems was developed and protocols were modelled using CSP, and verified using FDR.

2.5.2 MAFTIA middleware

The work on MAFTIA middleware included the definition of its architecture and the design of two suites of protocols (Armstrong *et al.* , 2002; Cachin *et al.* , 2001).

MAFTIA middleware is group-oriented, i.e., it supports the communication among groups of participants. *Participant* is an abstract denomination for the application level software, i.e., the component that uses the services of the middleware.

The architecture of MAFTIA middleware in a host is depicted in Figure 2.5. The architecture has two levels: participant and site. The participant level handles the com-

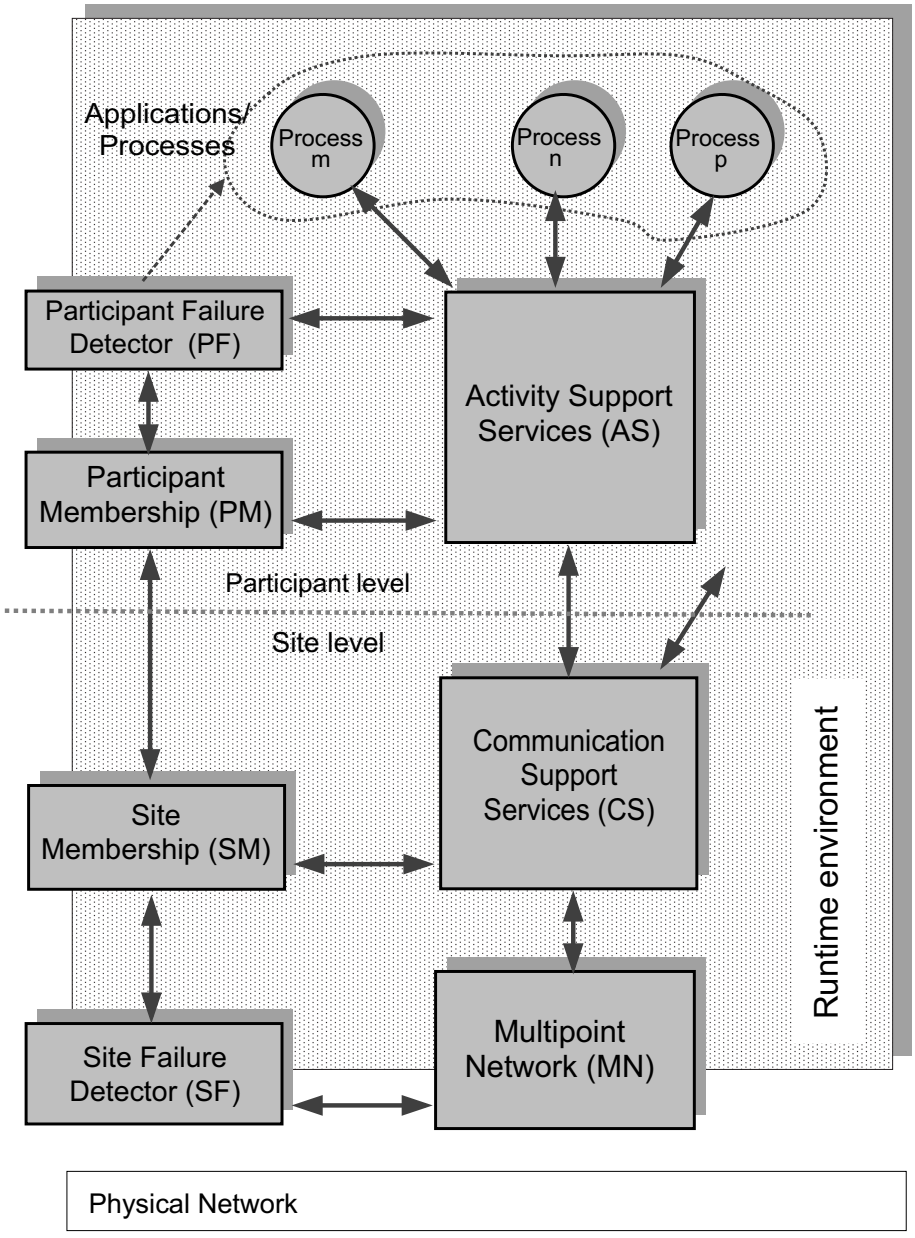

Figure 2.5: Architecture of the MAFTIA middleware in a node.

munication among participants, while the site level handles inter-host communication. A *participant-group* is multiplexed into a *site-group*, composed of all hosts where there are participants of the former group. This avoids, e.g., sending several copies of the same data to a host if there are several participants in it. Therefore, this division in participant and site levels is the materialization of a form of clustering used in MAFTIA: a site is a cluster of participants. This clustering improves the scalability of the middleware.

The backplane of the figure depicts the runtime environment. It includes the operating system and a protocol kernel, which simplifies the programming of protocols and supports their execution. The protocol kernel used in MAFTIA was Appia (Miranda *et al.* , 2001). The runtime environment includes also the TTCB (see Chapter 3).

The lowest layer of the architecture is the Multipoint Network module, MN, created over the physical network infrastructure. The objective is to provide some degree of abstraction of the specific underlying network(s) below. Its main properties are the provision of multipoint addressing and a moderate best-effort error recovery ability, both depending on topology and site liveness information.

In the site level, the Site Failure Detector module, SF, is in charge of assessing the connectivity and correctness of sites, and the MN module depends on this information. The Site Membership module, SM, depends on information given by the SF module. It creates and modifies the membership of site-groups. The Communication Support Services module, CS, implements basic cryptographic primitives, group communication with several reliability and ordering guarantees (e.g., reliable multicasts, atomic multicast), and other core services. The CS module depends on information given by the SM module about the composition of the groups, and on the MN module to access the network.

In the participant level, the Participant Failure Detector module, PF, assesses the liveness and the correctness of local participants. The Participant Membership module, PM, performs operations similar to the SM, but on the membership of participant-groups. The PM module monitors all groups with local members, depending on in-

formation propagated by the SM and by the PF modules, and operating cooperatively with the corresponding modules in the concerned remote sites. The Activity Support Services module, AS, implements building blocks that assist participant activity, such as replication and transactional management.

Two suites of site-level protocols were developed in the project. One suite is 'time-free', i.e., it is asynchronous. This suite includes a binary Byzantine-agreement protocol (Cachin *et al.* , 2000), reliable and atomic multicast protocols (Cachin & Poritz, 2002). The other suite is 'timed' and is composed essentially by the protocols reported in this thesis (Chapters 4-7). At application-level, an intrusion-tolerant transaction service with support for multiparty transactions was designed (Armstrong *et al.* , 2002).

Note

The author participated in the development of some of the ideas and concepts presented in this chapter in the context of the Navigators group involvement in project MAFTIA. This work is reported in (Veríssimo *et al.* , 2000a; Cachin *et al.* , 2001; Correia *et al.* , 2001a; Armstrong *et al.* , 2001; Armstrong *et al.* , 2002; Veríssimo *et al.* , 2003).

3

Trusted Timely Computing Base

This chapter describes the model and the design of a security kernel called Trusted Timely Computing Base (TTCB)[1]. A security kernel (Ames *et al.* , 1983) is a fail-controlled subsystem trusted to execute a few functions correctly, albeit immersed in an environment subjected to malicious faults. In the past, security kernels have mainly been used as *intrusion prevention devices*, by supporting the mediation/protection of all system interactions, and/or all accesses to system resources. The reference monitor paradigm is such an example (Lampson, 1974). Alternatively, we argue that a security kernel can be used as an *intrusion tolerance device*. The idea is to consider that most of the system runs in an environment prone to attacks, but there is a secure subsystem that is used to run crucial phases of execution allowing a collection of processes to *tolerate* intrusions in some of them. Think for example in a web server with several replicas. A security kernel can run some steps of an intrusion-tolerant protocol that provides correct results, even if some replicas are intruded and behave maliciously (i.e., try to break the protocol).

The TTCB has some innovative features. Firstly, it is a *distributed* subsystem with its own secure channel/network – the control channel/network (see Figure 3.1). A distributed security kernel represents a "hard-core" component, offering trusted services to a collection of processes [2], despite the fact that the latter reside in different nodes, and that their normal communication is through an insecure network – the payload

[1] A prototype of the TTCB is available at http://www.navigators.di.fc.ul.pt/software/ttcb/.

[2] Throughout the chapter we use the word *process* to denominate any software component that uses the TTCB services, e.g., an operating system process, a thread, or a Java applet.

network (see figure). In consequence, the collection of processes can achieve some degree of distributed trust, for low-level facts reported to/by the TTCB for/to all (and thus agree on them), without having to explicitly communicate. That is, protocol processes essentially exchange their messages in a world full of threats, some of them may even be malicious and cheat, but there is an oracle that correct processes can trust, and a channel that they can use to get in touch with each other, even if for rare moments. Moreover, this oracle also acts as a checkpoint that malicious processes have to synchronize with, and this limits their potential for Byzantine interactions (inconsistent value faults).

Secondly, the TTCB is synchronous (or real-time), in the sense of having reliable clocks and being able to execute timely functions, and obviously do it in a distributed way: the control channel provides timely (synchronous) inter-module communication. As such, it is capable, for example, of telling the time, measuring durations of distributed operations, and detecting timing failures.

Thirdly, the TTCB can be implemented using only COTS components, hardware and operating system. In consequence, all the design guidelines and the mechanisms we describe in the chapter are reproducible and useable in open settings. As a matter of fact, a prototype of the TTCB that runs in common PCs with RTAI (Cloutier *et al.* , 2000) [3], a real-time brand of Linux similar to RT-Linux (Barabanov, 1997; Yodaiken & Barabanov, 1999), currently available for free non-commercial use.

Networked systems like the Internet suffer from a problem of unpredictability, notably in terms of time and security. Recently, the *wormholes* metaphor was proposed as a way to handle this unpredictability (Veríssimo, 2003). The idea is to extend a distributed system with a wormhole, i.e., with a privileged component which provides services with some degree of predictability. The concept is very generic and has several possibilities for application. The TTCB is an example of a distributed wormhole that can be used to achieve predictability in terms of time and/or security.

The chapter discusses essentially three things about our distributed wormhole.

[3] More information at http://www.rtai.org

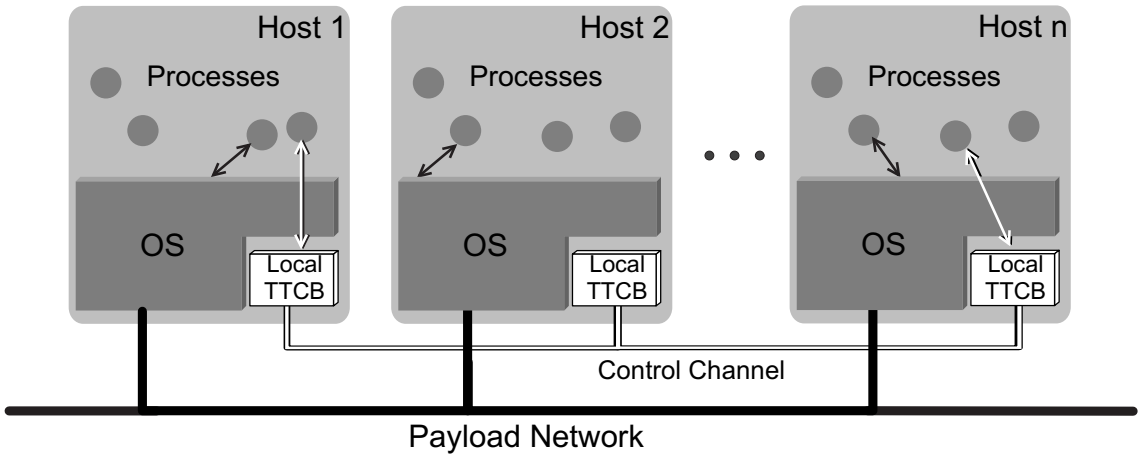

Figure 3.1: The architecture of a system with a TTCB.

First, it presents the TTCB model and architecture (Section 3.1), and describes the services it provides and their implementation, with special emphasis on the security services (Section 3.2). The TTCB services were formally validated to some extent using CSP and FDR (Sections 3.2.1 and 3.2.3). Next, the chapter shows how resilience to attackers can be enforced in a specific implementation of the TTCB: the *COTS-based TTCB*. This implementation of the TTCB follows a design methodology based on a *composite fault model*, that clearly identifies the malicious faults that have to be processed in order to prevent intrusions in the TTCB (Section 3.3). The reader should have in mind this difference between the TTCB model and a specific implementation proposed here, that is not the only one which is possible. To conclude, the chapter motivates the design of intrusion-tolerant systems using the TTCB (Section 3.4).

3.1 The TTCB model and architecture

The TTCB is a secure real-time distributed component that aims to assist the execution of applications. The architecture of a system with a TTCB is suggested in Fig-

AN1 Broadcast –	The AN has an unreliable packet broadcast primitive
AN2 Integrity –	Nodes can detect if packets were corrupted in the network. Corruptions are converted to omission failures
AN3 Omission degree –	No more than Od omissions may occur in a given interval of time
AN4 Bounded delay –	Any correct packet is received within a maximum delay T_{send} from the send request
AN5 Partition free –	The network does not get partitioned
AN6 Broadcast Degree –	If a broadcast is received by any local TTCB other than the sender, then it is received by at least Bd local TTCBs
AN7 Confidentiality –	The content of network traffic cannot be read by unauthorized entities
AN8 Authenticity –	Nodes can detect if a packet was broadcast by a correct node

Table 3.1: Abstract Network (AN) properties.

ure 3.1. An architecture with a TTCB has a local module in some hosts, called the *local TTCB*. These modules are interconnected by a *control channel* or *control network*, depending on the implementation. This set up of local TTCBs interconnected by the control channel/network is collectively called *the* TTCB. The TTCB is used to assist protocols/applications running among processes in the hosts concerned, on any usual distributed system architecture, encompassing a set of hosts interconnected by a network (e.g., the Internet). We call the latter the *payload system and network*, to differentiate from the TTCB part.

Conceptually, a *local TTCB* should be considered to be a *module* inside a host, with a well defined interface, and protected from the OS. In practice, this conceptual separation between the local TTCB and the OS can be achieved in several ways: (1) the local TTCB can be implemented in a separate, tamperproof hardware module —coprocessor,

PC board, etc.— and so the separation is physical; (2) the local TTCB can be implemented on the native hardware, with a virtual separation and shielding implemented in software, between the former and the operating system and processes. The direction followed in the thesis was the second, the one based on COTS components (hardware and software), since it presented more challenges and yielded a ready-to-use prototype in PC platforms. This design of the TTCB is discussed later in the chapter.

The local TTCBs are assumed to be fail-silent, i.e., they can only fail by crashing. The TTCB cannot produce erroneous interactions or results (even on account of attacks). Every local TTCB has a clock and the clocks are synchronized. This means that clock values of correct local TTCBs at any real-time t differ by at most a known constant π, the precision of the clock set.

The TTCB control channel has well-defined characteristics, specified in Table 3.1 as a set of abstract network properties (Veríssimo & Rodrigues, 2001, Chapter 13). The internal protocols of the TTCB are designed on the top of this abstract network. This way, the control channel does not have to rely on a specific network technology: the abstract network can be mapped onto different networks with the assistance of simple adaptation mechanisms.

The TTCB offers two sets of services listed in Table 3.2. These services can be considered to be part of the system runtime environment. They can be called by applications, protocol layers and other software components. An application can use all the services, but usually only a subset is used.

The motivation for the security services was to support the execution of intrusion-tolerant protocols and applications. The thesis shows precisely how these services can be used to implement reliable multicast, consensus, membership and atomic multicast protocols. The time services are the same as the Timely Computing Base services, which can be used to support the execution of partially-synchronous protocols (see Section 2.3.3).

Security services	
Local authentication	For a process to authenticate the TTCB and establish a secure channel with it.
Trusted block agreement	Achieves agreement on a small, fixed size, data block.
Trusted random numbers	Generates trustworthy random numbers.
Time services	
Trusted absolute timestamping	Provides globally meaningful timestamps.
Trusted duration measurement	Measures the duration of an operation execution.
Trusted timing failure detection	Checks if an operation is executed in a time interval.
Trusted timely execution	Executes operations securely and within a certain interval of time.

Table 3.2: TTCB services.

3.2 TTCB services

This section presents the TTCB services and their design. The design is generic since it relies on an abstraction of the control network (Table 3.1).

3.2.1 Local authentication service

The purpose of this service is to allow processes to authenticate and establish a *secure channel* with a local TTCB. The need for this service derives from the fact that, in general, the communication path between the process and the local TTCB is not trustworthy. For instance, that communication is probably made through the operating system that may be corrupted and behave maliciously [4]. We assume that the process–local TTCB communication can be subject to passive and active attacks (see

[4]If the process is an OS process or thread, a malicious OS is able to attack not only the process-TTCB communication but also the process itself. In these situations, protecting the communication does not add to the application security although, in practice, it prevents some attacks. However, it makes sense to protect the communication if the processes are protected from the OS, e.g., if they are inside other boards or a Smartcard, or if they use code protection mechanisms (see Section 2.1.3.1).

Section 2.4.1). A call to the TTCB involves two messages, a request and a reply, that can be read, modified, reordered, deleted, and replayed by an attacker.

Every local TTCB has an asymmetric key pair (K_u, K_r) that is used to authenticate it. The process that calls the Local Authentication service is assumed to have a trusted copy of the local TTCB public key K_u. These public keys can be distributed, for instance, manually or using a Public Key Infrastructure (PKI). The private key K_r is assumed to be known only by the local TTCB. A secure channel is obtained establishing a shared symmetric key K_{et} between the process and the local TTCB, that is later used to secure their communication.

The protocol to establish the shared key has to be an *authenticated key establishment protocol* with local TTCB authentication, defined in terms of the following properties (Menezes *et al.* , 1997, Chapter 12):

- *SK1 Implicit Key Authentication.* The process and the TTCB know that no other process has the key.
- *SK2 Key Confirmation.* Both the process and the TTCB know that the other has the key.
- *SK3 Authentication.* The process has to authenticate the local TTCB.
- *SK4 Trusted Against Known-Key Attacks.* Compromise of past keys does *not* allow either (1) a passive adversary to compromise future keys, or (2) impersonation by an active adversary [5].

A simple protocol with properties SK1 through SK4 can be implemented with two messages, i.e., a single function call. The protocol is presented in Figure 3.2. The protocol proof can be found in Appendix A.

[5] A passive adversary "attempts to defeat a cryptographic technique by simply recording data and thereafter analyzing it (e.g., in key establishment, to determine the key). An active attack involves an adversary who modifies or injects messages." (Menezes *et al.* , 1997, Chapter 1)

		Action	Description
1	P → T	$\langle E_u(K_{et}, X_e)\rangle$	The process sends the TTCB the new key K_{et} and a challenge X_e, both encrypted with the local TTCB public key K_u
2	T → P	$\langle S_r(X_e), E_{et}(ID)\rangle$	TTCB sends the process the signature of the challenge obtained with its private key K_r and the process ID (eid) encrypted using key K_{et}

Figure 3.2: Local Authentication service protocol

The shared key K_{et} has to be generated by the process, not by the TTCB. We would desire it to be the other way around, but the only key they share initially is the local TTCB public key, which can be used by the process to protect information that can be read only by the local TTCB (that has the corresponding private key) but not the contrary. K_{et} has to be generated by the process in such a way that a malicious OS cannot guess or disclose it. The generation of a random key requires sources of randomness (timing between key hits and interrupts, mouse position, etc.), sources that in mainstream computers are controlled by the OS. This means that when a process gets allegedly random data from those sources, it may get either data given or known by a potentially malicious OS. Therefore, there is the possibility of a malicious OS being able to guess the random data that will be used by the process to generate the key, and consequently, the key itself. This problem is hard to solve, however, a set of practical criteria can help to mitigate it:

- the process should use as much as possible sources of random data not controlled by the OS.
- The process should use as many different sources of random data as possible. Even if an attacker manages to corrupt the OS, it will probably not be able to corrupt its code in many different places and in such a synchronized way, so that it may guess the random number.

- The process should use a *strong mixing function*, i.e., a function that produces an output whose bits are uncorrelated to the input bits (Eastlake *et al.* , 1994). An example is a hash function such as MD4 or MD5.

For similar reasons, the protocol challenge, X_e, has to be generated by the process using the same approach.

The Local Authentication service protocol is implemented in the TTCB API as a single call with the following syntax:

```
eid, chlg_sign ← TTCB_localAuthentication(key, protection, challenge)
```

The input parameters are the key, the communication protection to be used, and the challenge. All input parameters are encrypted with the local TTCB public key. The output parameters are the process identification –*eid*– used to identify the process in the subsequent calls, and the signature of the challenge.

The Local Authentication service protocol was formally verified in the context of project MAFTIA (Adelsbach *et al.* , 2003, Chapter 4). The protocol was specified in the process algebra CSP and verified using the FDR model checker. The protocol was shown to satisfy properties SK1–SK4.

3.2.1.1 Process-TTCB secure channel

The communication between a process and the local TTCB is protected by a secure channel (or 'trusted path'), based on the existence of the shared key K_{et}. It is reasonable to consider that processes can have different security requirements for this communication. Therefore, when the Local Authentication service is called, the parameter *protection* is used to select one of the three *protection modes* below. (The same parameter is also used to select encryption and hashing algorithms, if necessary.)

1. *Authenticity only.* In particular situations or system architectures, a process may be able to communicate securely with its local TTCB (e.g., if the process is the

OS itself). Therefore, the secure channel has only to guarantee the authenticity of calls to the local TTCB, i.e., that the service is being called by the process whose *eid* comes in the call.

2. *Authenticity and integrity.* The process requires that the communication authenticity and integrity are guaranteed.

3. *Authenticity, integrity and confidentiality.* The process requires not only that the communication authenticity and integrity are guaranteed but also that its content cannot be disclosed.

The way authenticity is guaranteed depends on the mode. In mode 1, it requires only that requests and replies take both the *eid* and the shared key, since only the process and the TTCB know this pair *(eid, secret)*.

Authenticity and integrity in mode 2 are guaranteed putting a message authentication code (MAC) in the messages (Menezes *et al.* , 1997, Chapter 9). The MAC is obtained using the algorithm indicated by the *protection* parameter of the Local Authentication service, for instance, an MD5 hash of the message concatenated with K_{et}. Messages have also to take the *eid* and a sequence number.

Authenticity, integrity and confidentiality in mode 3 are enforced encrypting the message with the key K_{et}, using the cryptographic algorithm indicated by the *protection* parameter.

3.2.2 Random number generation service

This service supplies uniformly distributed random numbers, which can be used as nonces or keys for cryptographic primitives such as distributed authentication protocols. The TTCB provides this service for efficiency since the method described in Section 3.2.1 can be slow and vulnerable.

The interface of the service is a single function that returns a random number:

```
number ← TTCB_getRandom()
```

In a future version of the TTCB, based on an appliance board, we envisage the use of a hardware random number generator. In the current COTS-based TTCB, the random numbers are given by the Linux random number generator. This generator works with an entropy pool that collects random data from several inputs: device driver noise, timing between key hits, timing between some interrupts, mouse position, timing between disk accesses, etc. When a random number is requested, a hash of the entropy poll is calculated using MD5.

3.2.3 The trusted block agreement service

The trusted block agreement service (TBA service for short) performs agreement protocols among sets of processes. These protocols, which for instance, multicast a number of bytes or reach to a consensus with a majority decision, are executed in a secure and timely fashion since the service runs inside the TTCB. The service is not intended to replace agreement protocols in the payload system: it works with "small" blocks of data (currently 160 bits), and the TTCB has limited resources to execute it.

The TBA service is formally defined in terms of the three functions *TTCB_propose*, *TTCB_decide* and *decision*. A process *proposes a value* when it calls *TTCB_propose*. A process *decides a result* when it calls *TTCB_decide* and receives back a result. The function *decision* calculates the result in terms of the inputs of the service. The *result* is composed of a value and some additional information that will be described below. Formally, the TBA service is defined by the following properties:

- *AS1 Termination.* Every correct process eventually decides a result.
- *AS2 Integrity.* Every correct process decides at most one result.
- *AS3 Agreement.* If a correct process decides *result*, then all correct processes eventually decide *result*.

- *AS4 Validity.* If a correct process decides *result* then *result* is obtained applying the function *decision* to the values proposed.
- *AS5 Timeliness.* Given an instant *tstart* and a known constant T_{TBA}, the result of the service is available on the TTCB by *tstart*+T_{TBA}.

The interface of the TBA service has two functions: a process calls *TTCB_propose* to propose its value and *TTCB_decide* to try to decide a result (*TTCB_decide* is non-blocking and returns an error if the agreement did not terminate). The expression *an agreement* is used to denominate an execution of the TBA service.

```
out ← TTCB_propose(eid, elist, tstart, decision, value)
result ← TTCB_decide(tag)
```

An agreement is uniquely identified by three parameters: *elist* (the list of processes involved in the agreement), *tstart* (a timestamp), and *decision* (a constant identifying the decision function). The service terminates at most T_{TBA} after it "starts", i.e., after either: (1) the last process in *elist* proposed or (2) after *tstart*, whichever of the two happens first (the formula for T_{TBA} is given in the next section). That shows the meaning of *tstart*: it is the instant at which an agreement "starts" despite the number of processes in *elist* that proposed. If the TTCB receives a proposal after *tstart* it returns an error.

The other parameters of *TTCB_propose* are: *eid* is the unique identification of a process before the TTCB, obtained using the Local Authentication service (Section 3.2.1); *value* is the 'block' the process proposes; *out* is a structure with two fields, *error*, an error code and *tag*, an unique identifier of the agreement before a local TTCB. A process calls *TTCB_decide* with the *tag* that identifies the agreement that it wants to decide. *result* is a record with four fields: (1) *error*, an error code; (2) *value*, the value decided; (3) *proposed-ok*, a mask with one bit per process in *elist*, where each bit indicates if the corresponding process proposed the value that was decided; (4) *proposed-any*, a similar mask that indicates which processes proposed any value. Currently there are only a few *decision* functions defined, which return the following values:

- *TBA_RMULTICAST*. Returns the value proposed by the first process in *elist* (therefore the service works basically as a reliable multicast).
- *TBA_MAJORITY*. Returns the value proposed by more processes (or one of the values more proposed if there are several with the same number of proposals).
- *TBA_AND*. Returns the 'bitwise and' of the values proposed.
- *TBA_OR*. Returns the 'bitwise or' of the values proposed.
- *TBA_XOR*. Returns the 'bitwise xor' of the values proposed.

3.2.3.1 Trusted block agreement service protocol

The internal protocol that implements the TBA service is time-triggered (Veríssimo & Rodrigues, 2001, Chapter 13): *TTCB_propose* is called asynchronously, and gives the TTCB data that is stored in tables; periodically that data is broadcast to all local TTCBs, including the sender, and, also periodically, it is read from the network and processed.

The protocol uses two tables (Algorithm 1). The *dataTable* stores all agreements data in a local TTCB. Each record has the state of one agreement with the format: *(tag, elist, tstart, decision, vtable)*. All fields have the usual meaning except *vtable*, which is a table with the values proposed (one per process in *elist*). *sendTable* stores data to be broadcast to all local TTCBs. Every record is a proposal with the format: *(elist, tstart, decision, eid, value)*. The agreement is identified by *(elist, tstart, decision)*, *eid* identifies the process that proposed and *value* is the value proposed.

Algorithm 1 shows an implementation of the protocol. It is based on the TTCB assumptions that we summarize here for clearness:

1. the local TTCBs have clocks synchronized with precision π;
2. the protocol code is executed in real-time (therefore there is a worst case execution time for every section of code);

3. every local TTCB communicates with the others exclusively by broadcasting a message with a constant period;

4. the network is described by the Abstract Network model (Table 3.1).

The protocol has four routines. The *propose routine* is executed when a process calls the TTCB function *TTCB_propose* (lines 1-9). The routine begins by doing some tests: if the process already proposed a value for this agreement; if the process that calls the service is in *elist*; if *tstart* already expired (line 3). Other tests, are also made but are not represented since they are not so related to the algorithm functionality. If the propose is accepted, its data is inserted in the tables sendTable and dataTable, and the *tag* is returned (lines 5-9).

The *broadcast routine* broadcasts data to all local TTCBs every T_s (the period) either if there is data in sendTable or not (lines 10-15). Every message is broadcasted $Od + 1$ times in order to tolerate omissions in the network (*Od* is the omission degree). After the broadcast, sendTable is cleaned.

The *receive routine* reads and processes messages every T_r (lines 16-24). Each message is broadcasted $Od + 1$ times (lines 12-14), therefore copies of the same message have to be discarded by the function *read* (line 18). For each message received, the data in each record of sendTable is inserted in dataTable (lines 19-23).

The *decide routine* is executed when a process calls the function *TTCB_decide*. The routine searches dataTable for the agreement identified by the tag and returns an error if it does not exists. If the instant $tstart + T_{TBA}$ passed or the local TTCB has the values proposed by all processes in *elist*, the result is obtained and returned.

In Appendix A we prove that the protocol implements the TBA service and that T_{TBA} can be given by:

$$T_{TBA} = T_s + WCET_{send} + T_{send} + T_r + WCET_{receive} + \pi \tag{3.1}$$

Algorithm 1 TBA service internal protocol.

```
1  {PROPOSE ROUTINE}
2  WHEN process calls TTCB_propose(eid, elist, tstart, decision, value) DO
3  if (process already proposed) or (eid ∉ elist) or (clock() > tstart) then
4     return error;
5  insert (elist, tstart, decision, eid, value) in sendTable;
6  get R ∈ dataTable : R.elist = elist ∧ R.tstart = tstart ∧ R.decision = decision;
7  if (R = ⊥) then
8     R ←(get_tag(), elist, tstart, decision, ⊥); insert R in dataTable;
9  return R.tag;

10 {BROADCAST ROUTINE}
11 WHEN clock() = round_s × T_s DO
12 repeat
13    broadcast(sendTable);
14 until (Od + 1 times);
15 sendTable ←⊥; round_s ←round_s + 1;

16 {RECEIVE ROUTINE}
17 WHEN clock() = round_r × T_r DO
18 while (read(M) ≠ error) do
19    for all (elist, tstart, decision, eid, value) ∈ M.sendTable do
20       get R∈dataTable : R.elist = elist ∧ R.tstart = tstart ∧ R.decision = decision;
21       if (R = ⊥) then
22          R ←(get_tag(), elist, tstart, decision, ⊥); insert R in dataTable;
23       insert value in R.vtable;
24    round_r ←round_r + 1;

25 {DECIDE ROUTINE}
26 WHEN process calls TTCB_decide(eid, tag) DO
27 get R ∈ dataTable : R.tag = tag;
28 if (R≠⊥) and [(clock()>R.tstart+T_TBA) or (all processes proposed a value)] then
29    return (calculate result using function R.decision and values in R.vtable);
30 else
31    return error;
```

The constants in the formula have the following meaning: T_s and T_r are respectively the send and receive periods; $WCET_{send}$ and $WCET_{receive}$ are respectively the send and receive routines worst execution times; T_{send} is the maximum delivery delay; π is the precision of the clock set.

This TBA service internal protocol was formally verified in project MAFTIA (Adelsbach *et al.* , 2003, Chapter 4). The protocol was specified in CSP and verified using FDR. The protocol was shown to satisfy properties AS1–AS5, not in the general case but only in executions with two processes involved. The generalization for more processes was left for future work.

3.2.3.2 The crash-tolerant protocol

The TBA service protocol in the previous section does tolerate the crash of one or more local TTCBs. If a local TTCB crashes during the broadcast loop (lines 12-14), some local TTCBs may receive the message while others do not. This inconsistency can lead to different local TTCBs giving different results to one or more agreements. The solution is to use a Reliable Broadcast protocol. This protocol guarantees, informally, that: (1) all local TTCBs deliver the same messages; (2) if the sender does not crash then all correct (i.e., not crashed) local TTCBs deliver the message (see Section 2.4). If the sender crashes then the message is delivered to all correct local TTCBs or to none.

This section presents a time-triggered Timely Reliable Broadcast that tolerates crashes, assumes channel omissions (Abstract Network property AN3), and is lightweight, in the sense that local TTCBs do not retransmit the messages they receive. A timely reliable broadcast is formally defined in terms of two primitives *R-broadcast (M)* and *R-deliver (M)*, where M is a message, which satisfy the following properties (inspired by (Hadzilacos & Toueg, 1994)):

- *TRB1 Validity.* If a correct local TTCB R-broadcasts M then it eventually R-delivers M.

- *TRB2 Agreement.* If a correct local TTCB R-delivers message M then all correct local TTCBs eventually R-deliver M.
- *TRB3 Integrity.* For any message M, a correct local TTCB R-delivers M at most once and only if M was R-broadcast by *sender (M)*.
- *TRB4 Timeliness.* There is a known constant $T_{broadcast}$ such that, if a message is R-broadcast at instant t, then no correct local TTCB R-delivers M after $t + T_{broadcast}$.

Algorithm 2 Timely reliable broadcast protocol.

```
1  {BROADCAST ROUTINE}
2  WHEN clock() = round_s × T_s DO
3  sender ←my_id(); seq ←round_s;
4  M ←(sender, seq, higherseqVector, data);
5  repeat
6     broadcast(M);
7  until (Od + 1 times)
8  round_s ←round_s + 1;

9  {RECEIVE ROUTINE}
10 WHEN clock() = round_r × T_r DO
11 while (read(M) ≠ error) do
12    for all M-ndlv in notDelivered do
13       if [(M-ndlv.sender = M.sender) and (M-ndlv.number < M.number)] or (M-
         ndlv.number < M.higherseqVector[M-ndlv.sender]) then
14          R-deliver(M-ndlv.data); remove M-ndlv from notDelivered;
15    if (higherseqVector[M.sender] > M.number) then
16       R-deliver(M.data);
17    else
18       put M in notDelivered; higherseqVector[M.sender] ←M.number;
19 round_r ←round_r + 1;
```

The protocol is shown in Figure 2. The *broadcast routine* is similar to the one in the original TBA service protocol. Every T_s a message M is broadcasted $Od + 1$ times to tolerate omissions in the channel. The message is broadcasted even if there is no data to be sent. This is important for the protocol to work properly and for the detection of

local TTCB crashes (a local TTCB is known to be crashed if a message is not received by its deadline (Casimiro & Veríssimo, 1999)). The message has an header with the sender identifier, a sequence number and the table *higherseqVector*. This table has an entry for every local TTCB that contains, for every other local TTCB, the highest sequence number of a message received from that local TTCB.

The *receive routine* starts by reading a message M (lines 9-19). Copies of messages already received are discarded by the function *read*. For every message received, the routine does two things: (1) tests if previously received but not R-delivered messages (stored in *notDelivered*) can be R-delivered (lines 12-14); (2) tests if M can be R-delivered (lines 15-18).

Considering AN6, the protocol tolerates Bd local TTCB crashes in a reference interval of time. A message can be R-delivered by a local TTCB when it knows that all other non-crashed local TTCBs will also R-deliver it (Agreement property). A local TTCB can R-deliver a message $M(s, n)$ when it receives (a) $M(s, n + 1)$ or (b) $M(s', n')$ with *higherseqVector[s]=n+1* (s is the sender and n the message number). The intuition behind this is: if s crashes during the broadcast of $M(s, n + 1)$ but at least one local TTCB receives the message, then at least Bd local TTCBs receive it (AN6) and at most other $Bd-1$ can crash (the protocol tolerates Bd crashes). Therefore, at least one correct local TTCB receives $M(s, n + 1)$ and broadcasts $M(s', n')$ with *higherseqVector[s]=n+1* to the other non-crashed local TTCBs. Since messages are broadcast $Od + 1$ times, all non-crashed local TTCBs either receive $M(s, n + 1)$ or $M(s', n')$ and R-deliver $M(s, n)$. In the protocol, line 13 tests this condition. However, it considers that any $M(s, n_+)$ with $n_+ > n$ causes $M(s, n)$ to be R-delivered, since the Abstract Network does not guarantee the order of the reception of messages. The same is true for $M(s', n')$ with *higherseqVector[s]* $> n$. Line 15 checks if the message received, $M(s'', n'')$, can be R-delivered immediately. This is the case if a message from the same sender but with a higher number was received previously, i.e., if *higherseqVector[*s''*]* $> n''$.

In Appendix A we prove these results and also that the protocol R-delivers a message M within $T_{broadcast}$ of *R-broadcast(M)* (the meaning of the constants is the same as

before):

$$T_{broadcast} = 2 \times (WCET_{send} + T_{send} + T_r + WCET_{receive} + T_s) + \pi \quad (3.2)$$

The TBA service protocol can be made crash-tolerant by replacing lines 12-14 and 18 in Algorithm 1 respectively by the two timely reliable broadcast protocol routines. The second condition in line 28 has also to be substituted by: "all processes *in non-crashed local TTCBs* proposed a value".

The TBA service termination instant is related to $T_{broadcast}$:

$$T_{TBA} = T_s + T_{broadcast} \quad (3.3)$$

The proof of this result is in Appendix A.

3.2.4 TTCB time services

This section describes briefly the TTCB time services. These services were defined on the context of the Timely Computing Base work (Veríssimo *et al.* , 2000b) and their implementation is discussed in detail elsewhere (Casimiro *et al.* , 2000; Casimiro & Veríssimo, 1999).

3.2.4.1 Trusted absolute timestamping service

Every local TTCB has an internal clock that is synchronized to the other local TTCB clocks. This is achieved with a clock synchronization protocol inside the TTCB. The Trusted Absolute Timestamping service gives timestamps that, since clocks are synchronized, are meaningful to all local TTCBs. The precision of the timestamps is lim-

ited by the precision of the clock synchronization protocol. The interface of the service is:

```
timestamp ← TTCB_getTimestamp();
```

When an application running on the payload part of the system asks for a timestamp, it receives it some time after it was generated by the TTCB. This delay is variable, depending mostly on the time taken by the operating system scheduler to give CPU time to the application, on the time the application takes to read the timestamp, and on potential attacks against time inside the host. However, a timestamp can still be useful since, e.g., the difference between two timestamps is an upper bound on the real duration of the time interval between them.

3.2.4.2 Trusted duration measurement service

This services measures the time taken to execute an operation. The service verifies the following property:

- *TDM Duration measurement.* Given any two events occurring in any two hosts at instants t_s and t_e, the TTCB is able to measure the duration between those two events with a known bounded error.

The service is used calling the functions:

```
tag ← TTCB_startMeasurement(start_ev);
duration ← TTCB_stopMeasurement(tag, end_ev);
```

The parameters `start_ev` and `end_ev` are timestamps that indicate respectively the time of the beginning and end of the operation to measure. `duration` is the value measured for the duration of the operation. `start_ev` has to be obtained prior to the execution of the service calling the timestamping service.

Taking in account the variable delay in the communication between a process and the TTCB, this service does not provide a precise measurement of the delay of an operation. It returns the interval between two receptions of two calls in the TTCB, which is a higher bound on the time taken to execute the operation in the payload system.

This definition of the service contemplates both local and distributed operations, i.e., *TTCB_startMeasurement* and *TTCB_stopMeasurement* could be called either in the same or in different local TTCBs. However, the current TTCB implementation allows only local measurements. The measurement of the delay of distributed operations can be done using the Trusted Timing Failure Detection service.

3.2.4.3 Trusted timely execution service

This service allows an application to execute (sporadically) a function with a strict timeliness and/or a high degree of security. The function is executed inside the TTCB before a deadline (eager execution) and/or after a liveline (deferred execution):

- *TTE Timely execution.* Given any function `f` with an execution time bounded by a known constant $T_{X_{max}}$, and given a delay time lower-bounded by a known constant $T_{X_{min}} \geq 0$, for any execution of the function triggered at real time t_{start}, the TTCB does not start the execution of `f` within $T_{X_{min}}$ from t_{start}, and terminates `f` within $T_{X_{max}}$ from t_{start}.

The function `f` is executed between the two instants `start_ev+delay` and `start_ev+t_exec`:

```
end_ev ← TTCB_exec(start_ev, delay, t_exec, f);
```

An issue left for future work is the definition of the functions that can be executed inside the TTCB. The TTCB can either offer a library of generic useful functions or let a process upload functions. The latter case requires a process to ensure that the

function is correct (i.e., that it will not attack the TTCB or create a vulnerability), and that calculates the worst-case execution time (WCET) for the function. When a process uses the Trusted Timely Execution service to request the execution of a function, the WCET is used to assess if the TTCB has resources to execute it (schedulability analysis). In case the TTCB does not have resources, an error is returned.

3.2.4.4 Trusted timing failure detection service

This service is used to detect if a timed action is executed after its deadline. The action is executed in the payload system and the TTCB only verifies its timeliness. It is defined by the two properties:

- *TTFD1 Timed strong completeness.* Any timing failure is detected by the TTCB within a known interval from its occurrence.
- *TTFD2 Timed strong accuracy.* Any timely action finishing no later than some known interval before its deadline is never wrongly detected as a timing failure by the TTCB.

The service has different APIs depending on the timed action being local or remote.

Local detection API Local timing failure detection is done calling the following two functions:

```
tag ← TTCB_startLocal(start_ev, spec, handler);
faulty ← TTCB_endLocal(tag, end_ev, duration);
```

The first function requests the TTCB to observe the timeliness of the execution of an operation. `start_ev` is the start instant and `spec` the expected duration. The `handler` is used to tell the TTCB the reaction to have if a failure is detected, in case it is needed. The handler has to specify a function in the same way as `f` in the Trusted

Timely Execution service. Examples of reactions are a fail-safe shutdown or crashing the host.

The second function disables the detection, i.e., it indicates the TTCB that the action terminated. The parameters returned indicate the termination instant (`end_ev`), the `duration` measured and if there was a time failure or not (`faulty`).

Distributed detection API The basic idea of this interface is that a distributed action is initiated by the transmission of a message from a sender to a recipient. The way the API works is similar to the *local detection API*, i.e., a process calls the TTCB telling that it is going to send a message (start a distributed action, *TTCB_startDistributed*), sends the message, the recipient receives the message, executes the remote operation, and tells the TTCB that it is delivered (*TTCB_delivDistributed*). If the time to receive the message expires, the TTCB executes a function, in case that was requested. Messages can be multicast to several recipients:

```
tag ← TTCB_startDistributed(start_ev, spec, mid, elist, handler);
deliv_ev ← TTCB_delivDistributed(mid, tag);
list_info ← TTCB_waitInfo(tag);
```

The parameter `mid` is a unique message id. The `handler` is executed by the local TTCB of the sender in case there is a timeliness failure. In *TTCB_delivDistributed* the parameters indicate that a message was received. When that call is made, the TTCB checks if there was a timing failure and returns that information.

A process, either the sender or a recipient, can get information relative to timing failures using the function *TTCB_waitInfo*. The input parameter `tag` is optional since the process may decide to wait for information of all or only one of the distributed actions it is involved in. The parameter `list_info` contains the delay to deliver and the indication about timeliness faults for every recipient.

Distributed timing failure detection is implemented by a distributed protocol inside the TTCB. This protocol is described in (Casimiro & Veríssimo, 1999).

3.3 The COTS-based TTCB design

The process of designing a system addresses both functional and non-functional aspects. The functional aspects are concerned with the algorithms and protocols that make the system carry out its service, mostly presented in the previous section. This section is more concerned with the non-functional design of the COTS-based TTCB.

3.3.1 Design methodology

3.3.1.1 Composite fault model with hybrid failure assumptions

The organization of assumptions in terms of the AVI composite fault model underpins the design philosophy (see Section 2.1.1). This model shows how the impairments that may occur to a system, security-wise, have to do with a wealth of causes, which range from internal faults (i.e., vulnerabilities), to external, interaction faults (i.e., attacks) which activate those vulnerabilities, producing faults (i.e., intrusions) that can directly lead to component failure.

The AVI composite fault model was shown in Figure 2.3. The figure also showed where to apply different techniques to prevent the system from failing. Because we differentiated the several fault classes, we can apply these techniques selectively, and in a structured way. Note for example, that an intrusion cannot occur unless there is a vulnerability to be activated by a corresponding attack (it makes no sense to prevent an attack for which there is no vulnerability, or vice-versa).

In our *composite fault model with hybrid failure assumptions*, the presence and severity of vulnerabilities, attacks and intrusions varies from component to component. Consider a component or sub-system like the TTCB, for which a given controlled failure assumption was made. How can we guarantee that assumption, given the unpredictability of attacks and the elusiveness of vulnerabilities?

The first-line techniques are vulnerability prevention (e.g., using correct coding

practices), and then attack prevention (e.g., physically isolating an access point) and vulnerability removal (e.g., patching the OS and removing absolute privileges from the root account).

All these techniques contribute to intrusion prevention. However, after this step there may still be attack-vulnerability combinations to fear from, illustrated in the figure, by the holes in the intrusion prevention barrier. The design must then be complemented with the necessary intrusion tolerance measures, for example, using intrusion detection and recovery or masking, until we justifiably achieve confidence that the component behaves as assumed, failing in the assumed controlled manner, i.e., the component is trustworthy. The measure of its trustworthiness is the coverage of the controlled failure assumptions (Section 2.1.4).

3.3.1.2 The methodology

The design of the TTCB with regard to the non-functional properties follows the principles underlined above. The design methodology has four steps. It makes sense to perform several iterations until the final result.

1. Define the desired system (TTCB) architecture and failure modes
2. Define the environment assumptions and the adaptation mechanisms that enforce these assumptions
3. Design the mechanisms and protocols that enforce the system failure modes
4. Assess the system design

Step one is the definition of the TTCB architecture and failure modes. The TTCB architecture was presented in Section 3.1 but is more detailed below in Section 3.3.2. The architecture itself can prevent some attacks against specific components. For example, if the control network is physically inaccessible to hackers, attacks that require

physical access are prevented. In relation to the failure modes, recall that we consider the local TTCBs to be fail-silent.

Step two is about the system's *environment*, i.e., about whatever is external to the system but that interacts with it: host hardware and OS, networks, attackers, etc. The environment is characterized in terms of a set of assumptions that, in practice, have to be enforced using adaptation mechanisms. The environment assumptions and the adaptation mechanisms are presented in the Section 3.3.3.

Step three deals with constructing the mechanisms and protocols that enforce the fail-silent behavior of the TTCB, on the assumed environment and architecture. This boils down to making the TTCB resilient to attacks and intrusions. The design methodology may recursively be applied to the internal components of the TTCB as part of this step. This is discussed in Section 3.3.4.

Step four consists in assessing the system design, in this case, the COTS-based TTCB subsystem. On the one hand, this is about determining whether the coverage of the design assumptions is acceptably high. On the other hand, about determining whether given the assumptions, the algorithms and their implementation provide the specified services. The verification and assessment of two of the TTCB services was done in the context of project MAFTIA, as mentioned above.

3.3.2 System architecture

The general architecture of the TTCB was presented in Section 3.1. It was also mentioned that our current implementation is based on common PCs with RTAI. To pursue the COTS strategy, our implementation is based on Fast-Ethernet, for campus-wide systems: we provide each host having a TTCB with an extra LAN adapter. We envisage future designs based on tamperproof hardware and wide-area networks such as an ISDN Virtual Private Network (VPN) [6]. A VPN provides a private channel, if we

[6] ISDN is a public digital network technology for data and telephony that provides connections with guaranteed bandwidth in multiples of 64 Kbps (128 Kbps, 1 Mbps...).

assume that the public telecommunications network is not eavesdropped. Additional security can be obtained using secure channels, e.g., encrypting the communication.

RTAI is an engineering of Linux, which was modified so that a real-time executive takes control of the hardware, to enforce real-time behavior of some real-time (RT) tasks. RT tasks were defined as special Linux loadable kernel modules (LKMs), so they run inside the kernel. The scheduler was changed to handle these tasks in a preemptive way and to be configurable to different scheduling disciplines. Linux runs as the lowest priority task and its interruption scheme was changed to be intercepted by RTAI. Real-time FIFOs are the basic mechanism for communication between and with RT tasks.

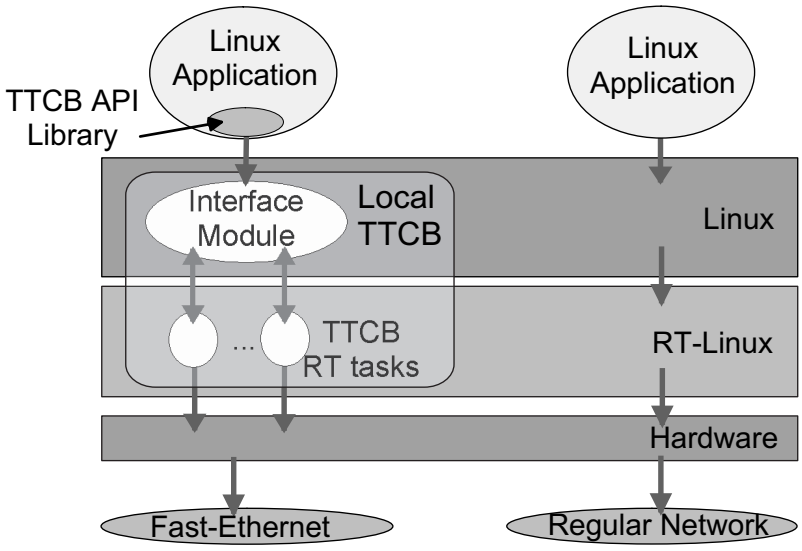

Figure 3.3: Architecture of the COTS-based local TTCB.

The COTS-based *local TTCB* architecture is detailed in Figure 3.3. The API functions are defined in libraries and communicate with the local TTCB using RT FIFOs. Currently there is one library for applications in C and another for Java (TTCB API Library in the figure). The local TTCB is implemented by an LKM (Interface Module) and by a number of RT tasks (TTCB RT Tasks). The TTCB Interface Module handles

calls from the processes. It is not real-time since it is part of the interface of the TTCB. All operations with timeliness constrains are executed by RT tasks. A local TTCB has always at least two RT tasks that handle communication: one to send messages to the other local TTCBs and another to receive and process incoming messages. Additional RT tasks can be used by the Trusted Timing Failure Detection service and the Trusted Timely Execution service (Section 3.2.4).

3.3.3 Environment assumptions and adaptation mechanisms

This section describes the environment assumptions, and the adaptation mechanisms needed to enforce them. The environment assumptions are shown in Table 3.3. The environment includes the PCs with RTAI (the *host*), the payload network and the control network.

A1	The host operating system, the TTCB code and the protection mechanisms are correct when the system starts.
A2	The host kernel memory is not read or written by any attacker.
A3	The control channel access point is not read or written by any attacker.
A4	The data on the control channel is not read or written by any attacker.
A5	Given a known interval of time, the control channel does not corrupt more than k packets.
A6	There are no partitions in the control channel.

Table 3.3: Environment assumptions.

Assumptions A1 and A2 impose limits on what the attacker can do inside a host. Assumptions A3 and A4 impose limits on what the attacker can do to the control chan-

nel. Assumptions A5 and A6 define the behavior of the control channel vis-a-vis accidental faults: a controlled omission degree, and a partition-free network environment, respectively.

3.3.3.1 RTAI and protection

We start by describing RTAI and discussing related protection issues. From the point of view of security, RTAI is very similar to Linux. One of the main vulnerabilities is the existence of a superuser that controls all system resources: it can read, modify and delete any file, any position of memory, etc. Most attacks against Unix/Linux machines at some stage try – and often manage – to obtain superuser privileges, e.g., attacking programs with setuid, using race conditions or buffer overflows.

Recently several Linux extensions and packages appeared that try to limit the power of the superuser: Linux capabilities (Tobotras, 1999), Immunix SubDomain (Cowan *et al.* , 2000), LOMAC (Fraser, 2000), LIDS (Huagang, 2000), etc. The current TTCB implementation uses the first, Linux capabilities, since they are already part of the kernel. Linux capabilities are extensions of the *Posix capabilities*, which are privileges or access control lists (ACLs) associated with processes, allowing to control how they can manipulate objects, i.e., other processes, files, directories, unnamed pipes, memory, and the system clock.

When Linux reboots, the process *init* has the full set of capabilities, and it should do the allocation of capabilities to all other processes. However, this mechanism is not yet fully implemented and the practical way of using the capabilities is with the *capability bounding set*. This set contains all the capabilities that can be held by processes in the system. If a capability is removed from the set, it cannot be used by any process until the next reboot, not even by a process with superuser privileges. The capability bounding set can thus be used to setup limits to the privileges of all processes until the next reboot. This mechanism is very limited since it allows only a resource to be enabled or disabled for all processes/users, however, it fits our needs.

3.3.3.2 Enforcing environment assumptions

Assumptions A1 and A2 impose the only limits on what the attacker can do inside a host. Otherwise, we assume that it can access the host, run software there, and become root or run processes with superuser privileges.

A1 states that the system is correct when it starts to run, i.e., the operating system, the TTCB and the protection mechanisms are not corrupt. This basic assumption can be enforced starting both the OS and the TTCB from a read-only device, such as a ROM or a CD-ROM. The protection mechanisms mentioned in A1 are basically a set of commands in a script that remove a set of Linux capabilities from the capability bounding set. This script is executed whenever the host is rebooted and has also to be stored in a read-only device.

Assumption A2 protects the working space of both the RTAI kernel and the modules that support the TTCB. If the attacker manages to modify the kernel memory, he has a dramatic potential for damage, which ranges from modifying kernel or TTCB code or state, to arbitrarily controlling any of the system components, since code in the kernel memory can execute privileged CPU instructions. Assumption A2 is enforced by removing two vulnerabilities. The removal of these vulnerabilities reduces the power of the superuser and consequently, the power of the attacker:

- Loadable kernel modules: Loadable kernel modules (LKMs) are the standard way of inserting code in the kernel in runtime. Their insertion and removal is restricted to the superuser but, since we consider that the attacker can become superuser, it is a vulnerability. This vulnerability is removed taking the capability `CAP_SYS_MODULE` off the capability bounding set. The local TTCB has to insert one LKM and several real-time tasks (that are also LKMs) in the kernel. This has to be done during reboot, before the capability is removed from the bounding set.
- `/dev/mem` and `/dev/kmem` devices: The devices `/dev/mem` and `/dev/kmem` allow an attacker with superuser privileges to change code and data in the kernel. This can be used to change the kernel and local TTCB code and state. This is

even more serious since the file `System.map` maps kernel symbols to physical addresses. An example of how this can be used to corrupt the kernel is a vulnerability that was found on the implementation of the capability bounding set itself. The physical address of this variable has the symbol `cap_bset`. A simple 'grep' of `System.map` allows one to get the physical address of the variable and a simple write in the memory allows the modification of the bounding set value. These devices can even be used to insert code in the kernel (Cesare, 1998). This vulnerability is removed disabling access to the two devices. This is done by removing `CAP_SYS_RAWIO` from the capability bounding set.

Assumptions A3 through A6 refer to the control channel. Assumption A3 stipulates that an attacker cannot access the control network adapter from inside the host, and in consequence, he can neither send to, nor read or intercept packets from, the control network. This can be enforced removing the access to the LAN controller (the device) so that only code in the kernel uses it.

Assumption A4, on the other hand, is secured by ensuring that an attacker does not have physical access to the control network medium devices (cables, switches, etc.). The assumption makes sense if we consider a short-range, inside-premises closed network, connecting a set of servers inside a single institution, with no other connection. We are assuming that the attacker comes from the Internet, through the payload network, without physical access to the servers or control network hardware. Long-range solutions also use technologies such as ISDN VPNs, which are hard for the common Internet attacker to tamper with in conjunction with an attack through the payload network. Note however that assumption A4 can still be enforced for a more powerful hacker who can eavesdrop on the control channel, by using cryptographic protection in the inter-TTCB communication.

In the just assumed absence of active attacks on the control channel, assumptions A5 and A6 establish limits to the events that may affect the timeliness of communication on the former, so that known bounds can be derived on message delivery delays, and failure detection can be accurately done. Networks can be tested in order to

find out the maximum number of packets they may corrupt in an interval of time, the omission degree (Veríssimo *et al.* , 1989). Likewise, short-range LANs have negligible partitioning, which can be further improved by using redundant channels. This would be essential to enforce A6 in wider-area networks.

3.3.4 Enforcing system failure modes

The AVI composite fault model in Figure 2.3 shows that different techniques can be used to make a system resilient to intrusions. The *attacker* in the figure is part of the environment, so its behavior is modeled by the environment assumptions in Table 3.3. Now, look at assumptions A1 through A4 in the table: they impose restrictions to the behavior of the attacker. Hypothesizing about limits to the behavior of malicious entities, such as hackers or viruses is, of course, not acceptable. Therefore, in the previous section we devised mechanisms that impose these restrictions in practice, i.e., that enforce the assumptions despite the potential arbitrary behavior of the attacker.

Assumptions A1 through A4 effectively do attack prevention (see Figure 2.3): it is an assumption that the attacker is not able to attack either the TTCB software modules or the control channel. Therefore, at this step of the methodology there is no need to enforce the system resilience to attackers. Handling the attacks/intrusions at step two (environment assumptions) is the same as doing it at step three, the one we are now. If we made RTAI part of the system then it would be the system that would be preventing or tolerating the faults, instead of the environment, i.e., protection would be made in step three instead of two. However, in this particular case, the way it is done seems more intuitive.

What remains to be defined at this stage is how the abstract network properties (Table 3.1) are obtained on top of the real network, taking in account the environment assumptions.

Property AN1 is available in the Ethernet and can be simulated with IP multicast or with several message sends in other networks. Property AN2 is imposed by most net-

works, through the *cyclic redundancy check* (CRC), if no attacks on the network are considered (the assumptions A3/A4). If there are attacks, message authentication codes (MACs) have to be used instead. Property AN3 is guaranteed by the environment assumption A5.

For property AN4 to be guaranteed in a dedicated switched Fast-Ethernet, packet collisions have to be avoided, since they would cause unpredictable delays (this issue is discussed at length in (Casimiro *et al.* , 2000)). This requires that: (1) only one host can be connected to each switch port (hubs cannot be used); and (2) the traffic load has to be controlled. The second requirement is handled by an access control mechanism, which accepts or rejects the execution of a service taking in account the availability of resources (buffers and bandwidth). This mechanism controls the network traffic preventing that a set of bounds are exceeded: the switch buffering and switching capacities; the buffering capacity of the network boards used by the local TTCBs; and the network bandwidth.

Property AN5 is guaranteed by the assumption A6. Property AN6 depends on several factors having to do with the transmission technology, and medium topology. In the network we are considering, a switched Fast-Ethernet, the broadcast degree Bd can easily exceed half of the nodes. Properties AN7 and AN8 are guaranteed by the assumptions A3 and A4, and could be enhanced using common cryptographic schemes.

3.4 Intrusion tolerance with the TTCB

After delving into the discussion of the TTCB services and design, a pertinent question at this stage is: *What is the TTCB good for?* This question is best answered after explaining the failure assumptions followed in the MAFTIA architecture.

3.4.1 Strategy for intrusion tolerance

With the TTCB, we can implement intrusion tolerance mechanisms, on a hybrid of arbitrary-failure (the payload system) and fail-silent (the TTCB) components. The TTCB is designed to assist crucial steps of the operation of middleware protocols. We use the word "crucial" to stress the tolerance aspect: unlike classical, prevention-based approaches (e.g., Reference Monitor), the component does not stand in the way of all resources and operations. As a matter of fact, protocols run in an untrusted environment. Local processes only trust interactions with the security kernel and single components can be attacked and intruded. Correct services are provided using distributed fault tolerance mechanisms, for example through agreement and replication amongst collections of processes in several hosts.

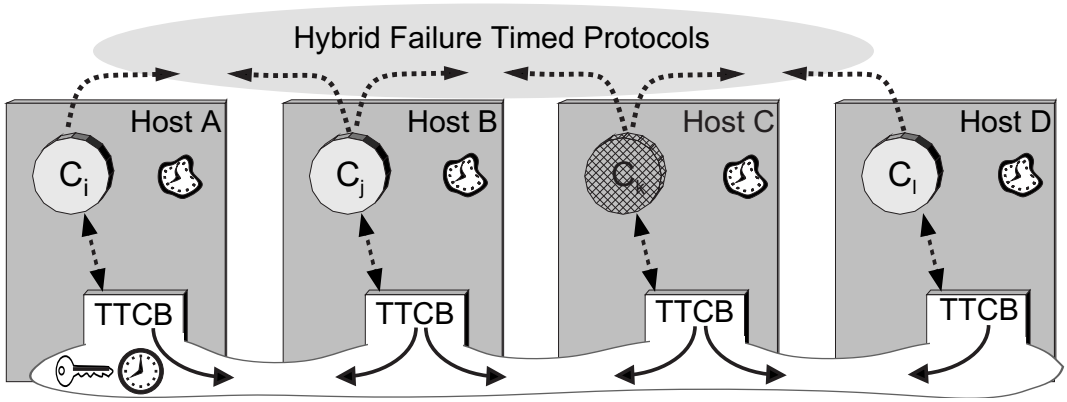

Figure 3.4: Intrusion tolerance with a TTCB.

Observe Figure 3.4: software components C_i interact through protocols that run on the payload system (the top arrows). However, they can locally access the TTCB in some steps of their execution, e.g., to be informed whether a message just received was or not corrupted. The white color is used to indicate a trusted environment (the TTCB). The key means the environment is secured using cryptography. The grey colors for the payload system mean untrusted.

Trusting the TTCB security kernel means that it is assumed that it is not feasible to subvert the TTCB, but it may be possible to interfere in its interaction with processes. In similar terms, whilst we let a local host be compromised, we must make sure that it does not undermine fault-tolerant operation of the protocols amongst distributed components. This implies two things: the operation of protocols can be intruded upon and individual components can be corrupted (e.g., C_k); and special care must be taken in order to preserve the validity of the interactions of a correct process with its local TTCB. The reader is referred to the next chapters, which show the use of the TTCB to implement intrusion-tolerant protocols.

In order to understand the assumptions on timeliness of our system, let us analyze Figure 3.4 again: the clock inside the TTCB area is meant to suggest it is a fully synchronous (or hard real-time) component. On the other hand, the warped clock in the payload area suggests that it has uncertain timeliness, or partial synchronism. It can even be asynchronous.

Constructing secure timed protocols in these environments is a hard task, due to the risk of attacks on the timing assumptions. For that reason, most known secure broadcast or Byzantine agreement protocols assume an asynchronous system. However, certain services, if provided in a trusted way – by the TTCB – can provide invaluable help.

3.4.2 Example applications with a TTCB

This section exemplifies the use of the TTCB in two different settings. Figure 3.5(a) shows a web server replicated inside a facility, in a company or another institution. Clients call the server using an intrusion-tolerant protocol. This protocol uses the TTCB to carry out some crucial steps, but otherwise runs in the payload system. If a subset of replicas is corrupted and behaves maliciously, the server still provides correct results, tolerating these malicious faults. The inside-facility TTCB can be the COST-based TTCB described in this chapter. This solution requires an extra Fast-Ethernet adapter

per host and an extra network switch, a negligible cost.

Several Internet authentication schemes rely on highly secure and distributed servers. For instance, Public Key Infrastructures have Certification Authorities (CAs) with these characteristics. Figure 3.5(b) shows a TTCB distributed over a wide area, that allows the execution of intrusion-tolerant protocols over such an extension. The TTCB control channel has to be a highly secure and wide channel, with guaranteed bandwidth (e.g., the above-mentioned ISDN VPN).

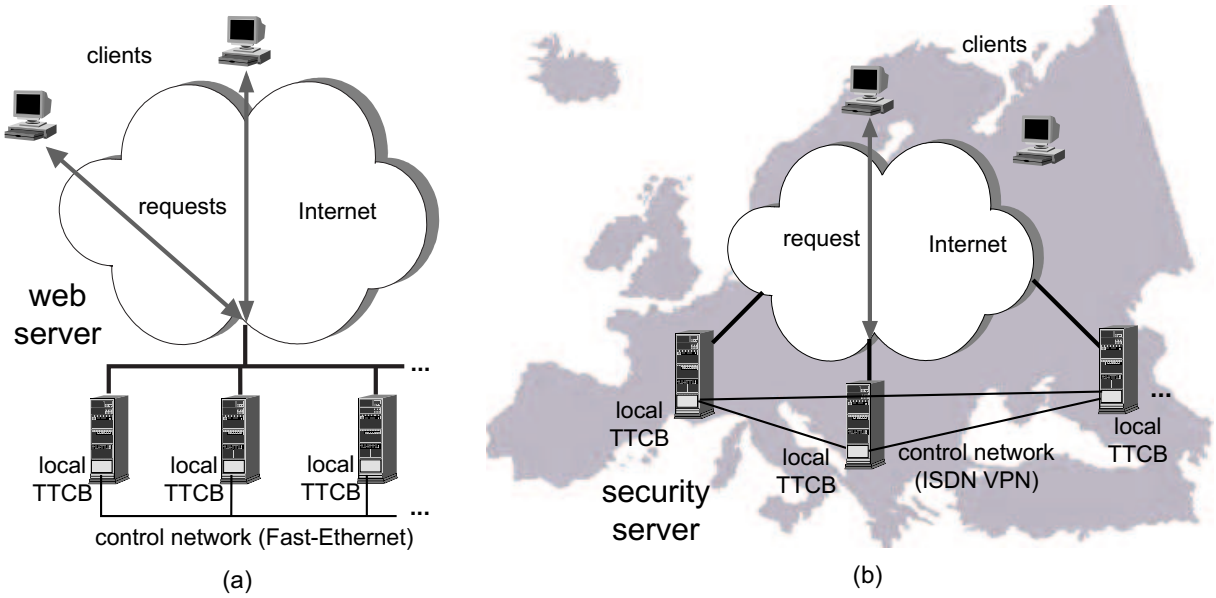

Figure 3.5: Examples of intrusion-tolerant systems with a TTCB: (a) replicated web server; (b) distributed security server.

3.5 Related work

The TTCB is a distributed security kernel which is radically different from the classic Trusted Computing Base (TCB) (National Computer Security Center, 1983) or the Network Trusted Computing Base (NTCB), composed of a set of interconnected

TCBs (National Computer Security Center, 1987). The objective of both the TCB and the NTCB is to provide *intrusion prevention* for all critical software in the host, i.e., to prevent that attacks against whatever is important in a host have success. The TTCB, on the contrary, is supposed to be the only secure component of a host, and to provide a limited set of services that assist processes to tolerate attacks. Even if some processes are attacked with success, the TTCB assists the collection of processes to go on delivering their service correctly. In the next chapters we show how the TTCB can be used to execute intrusion-tolerant protocols. We are not aware of any distributed security kernel with the above-mentioned characteristics of the TTCB. We are also not aware of any real-time security kernel.

The TTCB builds on the Timely Computing Base work (Veríssimo *et al.* , 2000b). The objective of this distributed component it to assist the implementation of timed operations and to detect timing failures. It assumes a benign fault model, i.e., on the contrary to the TTCB it is not resilient to malicious faults. The TTCB addresses a larger spectrum of applications, therefore is provides not only all the functionality of the Timely Computing Base, but also additional security-related services.

The idea of a device that assists the execution of secure applications or services is not new. There are several hardware devices that we can put in that category, e.g., secure coprocessors, cryptographic accelerators, and Smartcards. Some security assistants have limited functionality, e.g., cryptographic accelerators execute cryptographic operations only.

Some devices offer diversified services. IBM 4758 and its predecessor, Citadel, are two general-purpose secure coprocessors (Smith *et al.* , 1998). They can be considered to be complete microcomputers, with their own processor, memory, operating system, etc. Project Dyad explored the use of those devices to help the construction of secure systems and applications (Tygar & Yee, 1993). Five uses were defined: (1) host integrity check (the coprocessor tests if trusted versions of software are executed at bootstrap); (2) secure audit trails (the coprocessor seals audits trails securely in order to detect potential corruption); (3) software copy protection (software can be protected from illegal

usage by encrypting some modules and executing them inside the coprocessor); (4) secure electronic money (coprocessors can ensure the integrity and privacy of electronic money operations); and (5) secure contracts (a set of coprocessors in partners' computers can assist the negotiation and signing of electronic contracts). Similar applications were devised for Smartcards (Itoi & Honeyman, 1999; Stabell-Kuløet *al.* , 1999; Shoup & Rubin, 1996). These card-shaped devices provide a wide range of functionality, from passive memory to a processor with memory and cryptographic primitives.

The Trusted Computing Platform Alliance (TCPA) is defining a secure subsystem called Trusted Platform Module (TPM) (TCPA, 2002) [7]. This subsystem is supposed to exist inside hosts and perform some security related operations. The services already defined include integrity check (to measure and report the state of the operating system and other software in the host), protected store (to store important data inside the subsystem). Other uses mentioned in the documentation are: trusted cryptographic primitives, secure auditing and logging, file integrity and software licensing. Currently there at least two hardware implementations of the TPM available from National Semiconductors and Infineon Technologies.

Although this chapter describes the implementation of the TTCB in COTS PCs and OS, it could also be implemented inside devices like secure coprocessors or Smartcards. Moreover, the TTCB is a distributed component and therefore it can support and assist distributed intrusion-tolerant applications in a more effective way. The TTCB is also real-time, so it can assist the execution of applications with time requirements.

The TBA service internal protocol is basically a consensus protocol with the possibility of selecting the decision function and with the two additional masks in the result. Synchronous reliable multicast protocols are known for a long time (Lamport *et al.* , 1982; Babaoğlu & Drummond, 1985; Babaoğlu *et al.* , 1986; Cristian *et al.* , 1985). On the contrary to the TBA protocol, most of these protocols rely on the diffusion of the messages that are received to all the other recipients in order to guarantee reliability. The idea of using the broadcast degree was introduced in (Babaoğlu *et al.* , 1986).

[7] See http://www.trustedcomputing.org.

3.6 Summary

The chapter describes a security kernel – the TTCB – with innovative features: first, it is distributed, with local parts in hosts connected by a control channel; second, it is real-time, capable of timely behavior; and third, it can be constructed using only COTS components. The chapter also presents the services of the TTCB and gives an intuition on how these services can be used to support the construction of a new generation of intrusion-tolerant protocols. The currently available implementation of the TTCB is based on a Fast-Ethernet network and common hardware running a real-time operating system, RTAI. By applying our design methodology, we expect that the existing implementation exhibits a good coverage of the assumptions, acceptable to most applications. This solution has one extra added advantage – the TTCB can be tested and used in open settings.

In the future, we are considering the implementation of the TTCB inside an appliance board. A version of RT-Linux for embedded systems is already available, which leads us to predict that the port will be straightforward. We also envisage an implementation using a wide area control network.

Note

The content of this chapter was partially published in (Correia *et al.* , 2002a; Correia *et al.* , 2001b).

4

Reliable multicast

Protocols that are able to tolerate Byzantine faults have been extensively studied in the past 20 years (Lamport *et al.* , 1982; Rabin, 1983), and they have been applied to a number of well-known problems, such as consensus and group communication primitives with different order guarantees. These protocols are usually built for a system composed of a set of cooperating processes (or machines) interconnected by a network. The processes may fail arbitrarily, e.g., they can crash, delay or not transmit some messages, generate messages inconsistent with the protocol, or collude with other faulty processes with malicious intent. The synchrony assumptions about the network and process execution have been either the synchronous or the asynchronous models. Recent research in this area, however, has mostly focused on asynchronous systems, since this model is well-suited for describing networks like the Internet and other WANs with unpredictable timeliness (examples can be found in (Castro & Liskov, 1999; Reiter, 1994; Malkhi *et al.* , 1997c; Alvisi *et al.* , 1999; Kihlstrom *et al.* , 2001; Moser *et al.* , 2000; Cachin *et al.* , 2000)). The assumption of this model has also one added advantage – the resulting protocol tolerates timing attacks.

Nevertheless, the asynchronous model has some drawbacks, and among them is the constrain that it imposes on the maximum number of processes that are allowed to fail simultaneously. For instance, Bracha and Toueg showed that, assuming Byzantine faults, it is impossible to send reliable multicasts if there are more than $f = \lfloor \frac{n-1}{3} \rfloor$ faulty processes in a system with n processes (Bracha & Toueg, 1985).

This chapter describes a new reliable multicast protocol for asynchronous systems

with a hybrid fault model: the Byzantine reliable multicast protocol (BRM-M) [1]. In our case, processes and network can behave in a Byzantine way; however, we assume the existence of a distributed security kernel that can only fail by crashing: the TTCB. This kernel only provides limited functionality, but can be called by processes to execute a few small steps of the protocol. By relying on this kernel, our protocol is highly efficient, for instance in terms of message complexity, when compared with traditional protocols. Moreover, it imposes constraints on the number of process failures that are similar to accidental fault-tolerant protocols: for f faults, our protocol requires $n \geq f + 2$ processes, instead of $n \geq 3f + 1$. In reality, our protocol does not impose a minimum number of correct processes. However, in practice, we say that the number of processes has to be $n \geq f + 2$ to denote the notion that the problem is vacuous if there are less than two correct processes. This was already pointed out by Lamport et al. (Lamport *et al.* , 1982).

The chapter makes fundamentally two contributions. It presents a novel way of designing Byzantine-resilient protocols, which rely on the TTCB to execute a few crucial steps. Moreover, it describes a new reliable multicast protocol that is highly efficient and imposes no constraints on the number of faulty processes.

4.1 Process failure modes

A process is *correct* if it follows the protocol until the protocol completion. There are several circumstances, however, that might lead to a process failure. For instance, a process can crash (e.g., due to a power outage) or start to behave maliciously (e.g., produce wrong results). In an arbitrary fault model, which is the model considered in this and the next chapters, no restrictions are imposed on process failures, i.e., they can fail arbitrarily. A process can simply stop working, or it can send messages without regard of the protocol, delay or send contradictory messages, or even collude with

[1] BRM-M is one of two reliable multicast protocols based on the TTCB. The 'M' indicates that the protocol tries to minimize the number of messages sent. The other protocol, BRM-T, tries to minimize the time of execution (Lung *et al.* , 2003).

other malicious processes with the objective of breaking the protocol.

Of the various reasons that can cause a process to produce incorrect results, traditionally the most difficult to tolerate is related to attacks made by humans. Once an attacker takes control of a process, it can make that process behave in any way, and if one wants to be conservative, one has to assume that it can cause that process behave in the worse possible manner to the protocol execution. In the rest of this section, we will look into the attacks that are specific to our architecture, and that might lead to the failure of the corresponding process.

A personification attack can be executed by a local adversary if it is able to get the pair $(eid, secret)$, which lets a process communicate securely with the local TTCB (Section 3.2.1). Before a process starts to use the TTCB, it needs to call the Local Authentication service to establish a secure channel with the local TTCB. The outcome of the execution of this procedure is a pair $(eid, secret)$, where eid is the identifier of the process and $secret$ is a symmetric key shared with the local TTCB. If an attacker penetrates a host and obtains this pair, it can impersonate the process before the TTCB and the TTCB before the process. If this pair is kept secret, the attacker can only try to disrupt or delay the communication between the process and the local TTCB – personification attacks are prevented (see Section 3.2.1.1).

Another personification attack is possible if the attacker obtains the symmetric key that a process shares with another process, a requirement of BRM-M that will be discussed in the next section. In this case, the attacker can forge some of the messages sent between the two processes. Most of the messages transmitted by our protocol do not need to be authenticated and integrity protected because corruptions and forgeries can be detected with the help of the TTCB. The only exception happens with the acknowledgments sent by the protocol, where it is necessary to add a vector of message authentication codes. A successful attack to a host and subsequent disclosure of the shared keys of a process, allows an attacker to falsify some acknowledgements. If the keys can be kept secret, then he or she can only disrupt or delay the communication, in the host or the network.

A denial of service attack happens if an attacker prevents a process from exchanging data with other processes by systematically disrupting or delaying the communication. In asynchronous protocols typically it is assumed that messages are eventually received (reliable channels), and when this happens the protocol is able to make progress. To implement this behavior processes are required to maintain a copy of each message and to keep re-transmitting until an acknowledgement arrives (which might take a long time, depending on the failure). In this chapter, we decided to take a different approach: if an attacker can systematically disrupt the communication of a process, then the process is considered failed as soon as possible, otherwise the attacker will probably disturb the communication long enough for the protocol to become useless. For example, if the payment system of an e-store is attacked and an attempt of paying an item takes 10 hours to proceed, that is equivalent to a failure of the store.

In channels with only accidental faults it is usually considered that no more than Od messages are corrupted/lost in a reference interval of time. Od is the *omission degree* and tests can be made in real networks to determine Od with any desired probability (Veríssimo *et al.* , 1989). If a process does not receive a message after $Od + 1$ retransmissions from the sender, with Od computed considering only accidental faults, then it is reasonable to assume that either the process crashed, or an attack is under way. In any case, we will consider the recipient process as failed. The reader, however, should notice that Od is just a parameter of the protocol. If Od is set to a very high value, then our protocol will start to behave like the protocols that assume reliable channels.

Note that the omission degree technique lies on a synchrony hypothesis: we 'detect' omissions if a message does not arrive after a timeout longer than the 'worst-case delivery delay' (the hypothesis). Furthermore, we 'detect' crash if the omission degree is exceeded. In our environment (since it is asynchronous, bursts of messages may be over-delayed, instead of lost) this artificial hypothesis leads to forcing the crash of live but slow (or slowly connected) processes. There is nothing wrong with this, since it allows progress of the protocol, but this method is subject to inconsistencies if failures are not detected correctly. In our system, we can rely on the timing failure

detector of the TTCB to ensure complete and accurate failure detection amongst all participants (Veríssimo *et al.* , 2000b), and feed a membership service complementing the reliable multicast protocol being described. These mechanisms are out of the scope of the present chapter, but substantiate the correctness of the omission degree technique applied to asynchronous environments, if supported with a timing failure detector.

Another advantage of considering systematically delayed processes as failed is related with the implementation of the TTCB. Since the TTCB is a small component, it can only keep the results of the TBA service for a limited time. If a delayed process asks for a result after it expired, the simplest thing to do is to consider the process as failed. Alternatively, the protocol could be made more complex to recover from this situation. However, there is not much justification in doing so for the reason pointed earlier – if a process is too late it is useless.

4.2 Byzantine reliable multicast

4.2.1 Protocol definition and properties

In each execution of a multicast, there is one sender process and several recipient processes. A message transmitted to a group should be delivered to all member processes (with the limitations mentioned below), including the sender. No assurances, however, are provided about the order of message delivery. Each process can deliver its messages in a distinct order. In the rest of the thesis, we will make the classical separation of *receiving* a message from the network (or from the lower protocol layers) and *delivering* a message – the result of the protocol execution.

Informally, a reliable multicast protocol enforces the following (Bracha & Toueg, 1985): 1) all correct processes deliver the same messages, and 2) if a correct sender transmits a message then all correct processes deliver this message. These rules do not imply any guarantees of delivery in case of a malicious sender. However, one of two things will happen, the correct processes never complete the protocol execution and

no message is ever delivered, or if they terminate, then they will all deliver the same message. No assumptions are made about the behavior of the malicious (recipient) processes. They might decide to deliver the correct message, a distinct message or no message.

Formally, a reliable multicast protocol has the properties below (Hadzilacos & Toueg, 1994). The predicate *sender(M)* gives the message field with the sender, and *group(M)* gives the "group" of processes involved, i.e., the sender and the recipients (note that we consider that the sender also delivers).

- *Validity:* If a correct process multicasts a message M, then some correct process in *group(M)* eventually delivers M.
- *Agreement:* If a correct process delivers a message M, then all correct processes in *group(M)* eventually deliver M.
- *Integrity:* For any message M, every correct process p delivers M at most once and only if p is in *group(M)*, and if *sender(M)* is correct then M was previously multicast by *sender(M)*.

4.2.2 The BRM-M protocol

The Byzantine Reliable Multicast BRM-M protocol is executed in two phases. In the first, the sender multicasts the message one time for the recipients, and then it securely transmits a hash code through the TTCB TBA service. This hash code is used by the recipients to ensure the integrity and authenticity of the message, i.e., that the message received is really the message which was sent (see discussion about hashes in Section 2.4.1). If there are no attacks and no congestion in the network, there is a high probability that the message is received by all recipients, and the protocol can terminate immediately. Otherwise, it is necessary to enter the second phase. Here, processes retransmit the message until either a confirmation arrives or the $Od+1$ limit is reached.

Each multicast is performed at most $Od + 1$ times in order to tolerate accidental omissions (Section 4.1).

Figure 3 shows an implementation of the protocol. A message consists of a tuple with the following fields: $(type, sender, elist, tstart, data)$. $type$ indicates if it is a data message (DAT) or an acknowledgement (ACK). $sender$ is the identifier of the sender process, and $data$ is either the information given by the application or a vector of MACs (see below). $elist$ is a list of eid's in the format accepted by the TTCB TBA service. The first element of the list is the eid of the sender, the others are the eid of the recipients. $tstart$ is the timestamp that will be given to the TBA service.

Each execution of the protocol is identified by $(elist, tstart)$. The protocol uses two low level read primitives, one that only returns when a new message is available, $read\_blocking()$, and another that returns immediately either with a new message or with a non-valid value ($\perp$) to indicate that no message exists, $read\_non\_blocking()$. These two primitives only read messages with the same value of $(elist, tstart)$ which correspond to a given instance of the protocol execution. Other values of the pair are processed by other instances of the protocol. We assume that there is a garbage collector that throws away messages for instances of the protocol that have already finished running (e.g., delayed message retransmissions). This garbage collector can be constructed by keeping in a list the identifiers of the messages already delivered and comparing these with the arriving messages.

4.2.3 First phase of the protocol

With the exception of the beginning, the code presented in the figure is common both to the sender and to the recipients. If the process is a sender, it constructs and multicasts the message to the recipients (lines 3-4). $tstart$ is set to the current time plus a delay T_1. T_1 should be proportional to the average message transmission time, i.e., it should be calculated in such a way that there is a reasonable probability of message arrival before $tstart$. In practice, the value of $tstart$ is a tradeoff: if it is too large, the

Algorithm 3 BRM-M Sender and Recipient protocol.

```
1  {PHASE 1}
2  if I am the sender then                                          {SENDER process}
3     M ←(DAT, my-eid, elist, TTCB_getTimestamp() + T_1, data);
4     multicast M to elist except sender; n-sends ←1;
5  else                                                             {RECIPIENT processes}
6     read_blocking(M); n-sends ←0;
7  outp ←TTCB_propose(M.elist, M.tstart, TBA_RMULTICAST, Hash(M));
8  repeat
9     outd ←TTCB_decide(outp.tag);
10 until (outd.error ≠ TBA_RUNNING);
11 if (outd.proposed-ok contains all recipients) then
12    deliver M; return;

13 {PHASE 2}
14 M-deliver ←⊥;
15 mac-vector ←calculate macs of (ACK, my-eid, M.elist, M.tstart, outd.value);
16 M-ack ←(ACK, my-eid, M.elist, M.tstart, mac-vector);
17 n-acks ←0; ack-set ←eids in outd.proposed-ok;
18 t-resend ←TTCB_getTimestamp();
19 repeat
20    if (M.type = DAT) and (Hash(M) = outd.value) then
21       M-deliver ←M;
22       ack-set ←ack-set ∪ {my-eid};
23       if (my-eid ∉ outd.proposed-ok) and (n-acks < Od+1) then
24          multicast M-ack to elist except my-eid; n-acks ←n-acks + 1;
25    else if (M.type = ACK) and (M.mac-vector[my-eid] is ok) then
26       ack-set ←ack-set ∪ {M.sender};
27    if (M-deliver ≠ ⊥) and (TTCB_getTimestamp() ≥ t-resend) then
28       multicast M-deliver to elist except (sender and eids in ack-set);
29       t-resend ←t-resend + Tresend; n-sends ←n-sends + 1;
30    read_non_blocking(M);          {sets M = ⊥ if there are no messages to be read}
31 until (ack-set contains all recipients) or (n-sends ≥ Od+1);
32 deliver M-deliver;
```

first phase may take longer than what is required; if the value is too small, a correct recipient may not receive the message before $tstart$ and the second phase will have to be executed unnecessarily (i.e., the opportunity to terminate the protocol early is lost).

Recipient processes start by blocking, waiting for a message arrival (line 6). Depending on whether there are message losses, the received message might be of type DAT or ACK, or a corrupted message with the fields $(elist, tstart)$ correct. The variable *n-sends* contains the number of messages that were multicast and is set initially to 1 for the sender and to 0 to the recipients (lines 4 and 6). Next, both sender and recipients propose the hash of the message, $Hash(M)$, to the TBA service (M is the message transmitted by the sender, or the first message received by the recipient), and then they block waiting for the result of the agreement (lines 7-10). The decision function used by the protocol, TBA_RMULTICAST, selects as result the value proposed by the first process in $elist$, which in this case is necessarily the sender (if the sender is correct it puts its eid in the first position of $elist$). Since the system is asynchronous, there is always the possibility, although highly improbable, that the sender experiences some delay and it tries to propose after $tstart$. In this case, *TTCB_propose* will return the error $TSTART\_EXPIRED$ and the sender process should abort the multicast, and the application can retry the multicast later (for simplicity this condition is omitted from the code). If all processes proposed the same hash of the message, all can deliver and terminate (lines 11-12). Recall that the field *proposed-ok* indicates which processes proposed the same value as the one that was decided, i.e., $Hash(M)$.

Figure 4.1 illustrates an execution where processes terminate after this first phase of the protocol. Notice in the figure that the TBA is initiated immediately after all processes have proposed their value and not by $tstart$.

4.2.4 Second phase of the protocol

The second phase is executed if for some reason one or more processes did not propose the hash of the correct message by $tstart$. Variable *M-deliver* is used to store

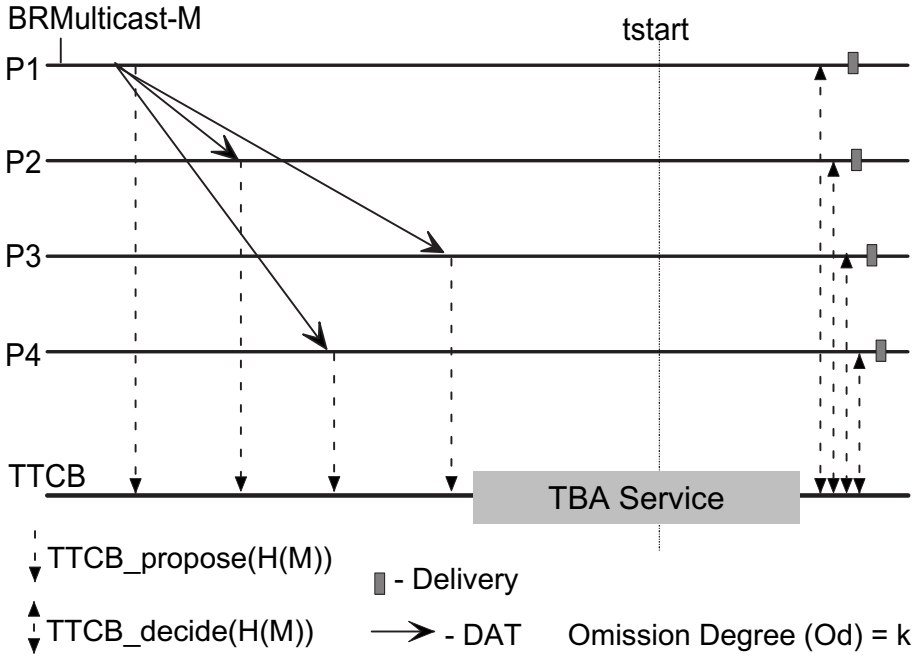

Figure 4.1: BRM-M example execution (best case).

the message that should be delivered, and is initialized to a value outside the range of valid messages (line 14).

The protocol uses message authentication codes (MAC) to protect ACK messages from forgery (Menezes *et al.* , 1997, Chapter 9). This type of signature is based on symmetric cryptography, which requires a different secret key to be shared among every pair of processes. Even though MACs are not as powerful as signatures based on public-key cryptography, they are sufficient for our needs, and more importantly, they are known to be several orders of magnitude faster to calculate. Since ACKs are multicast to all processes, an ACK does not take a single MAC but a vector of MACs, one per each pair (sender of ACK, other process in $elist$) (Castro & Liskov, 1999). A MAC protects the information contained in the tuple *(ACK, my-eid, M.elist, M.tstart, outd.value)*, and is generated using the symmetric key shared between each pair of processes (lines 15-16).

Next, processes initialize variables *n-ack* and *ack-set* (line 17). The first one will count the number of ACKs that have been sent. The second will store the eid of the processes that have already confirmed the reception of the message, either by proposing the correct $Hash(M)$ to the agreement (line 17) or with an acknowledgement message. *t-resend* indicates the instant when the next retransmission should be done (line 18). It is initialized to the current time, which means that there will be a retransmission as soon as possible.

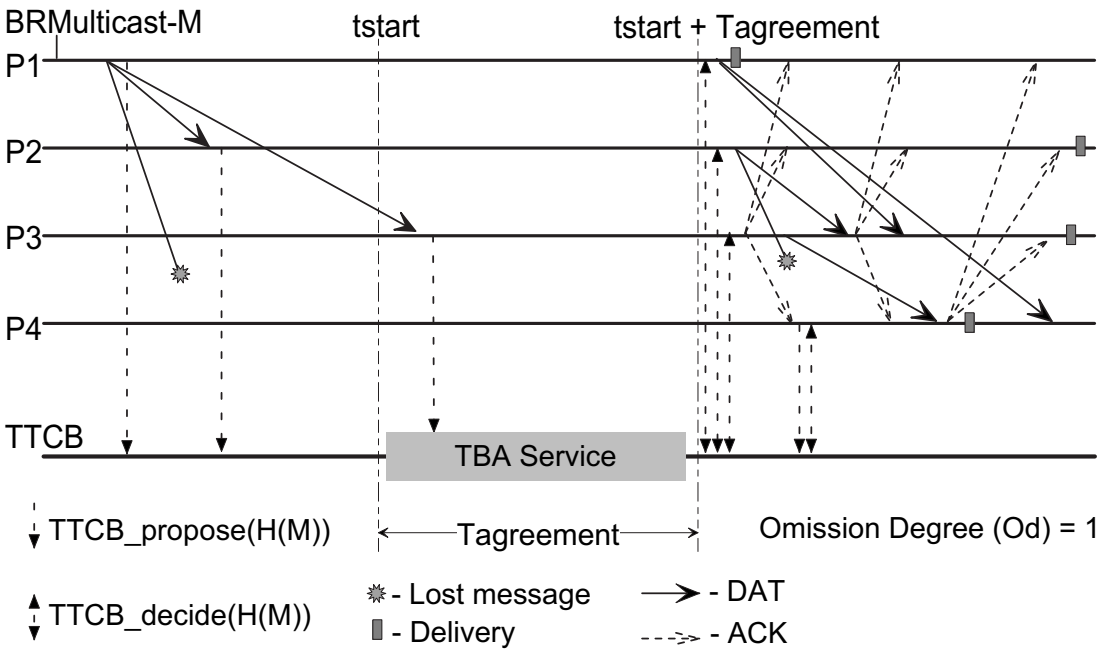

Figure 4.2: BRM-M example execution (normal case).

The loop basically processes the arriving messages (lines 20-26), does the periodic retransmissions (lines 27-29), and reads new messages (line 30). If the message is of type DAT and its hash is the same as the one given by the sender (line 20) then it is saved for later delivery (line 21). Next, the eid of the process is added to *ack-set* to indicate that this process has correctly received the message (line 22). If the process received the message but did not propose the correct hash to the agreement then it

needs to confirm the reception by multicasting an ACK (lines 23-24). The ACKs, like the DAT messages, are only transmitted $Od + 1$ times. If the received message is an ACK with a valid MAC, then the eid of the sender is put in *ack-set* (lines 25-26). Next, if it is time, the message is retransmitted to the processes that did not confirm the reception (lines 27-29). The loop goes on until $Od+1$ messages are sent or all recipients acknowledged the reception of the message (line 31). To complete the protocol, the process delivers the message.

As mentioned above in Section 4.2.1, there are situations in which the protocol does not terminate in a process if the sender is malicious or the process is failed. For instance, a malicious sender could propose a false hash of the message, and in that case no correct recipients would be able to deliver the message. To address this problem, a garbage collection mechanism has to be used in order to prevent correct processes from being clogged with protocol instances that never terminate. This mechanism should interact with a membership service to identify and remove instances waiting for faulty processes.

Figure 4.2 represents an execution of the protocol. The sender multicasts the message once, P2 receives it in time to propose $Hash(M)$, P3 receives the message late and P4 does not receive. When the agreement terminates all processes except P4 have the message and get the result from the TTCB (P4 does not even know that the protocol is being executed). At this point, by observing the result of the agreement, all become aware that only P1 and P2 proposed the hash. Therefore, both P1 and P2 multicast the message to P3 and P4. P3 multicasts an ACK to all processes confirming the reception and sends the message to P4. P1 terminates at this moment because it has already sent the message $Od+1$ times. The first message P4 receives is the ACK sent by P3. P4 saves it in *ack-set* and gets the result of the agreement. Then it receives the right message, and multicasts an ACK. At this moment, all processes terminate.

A proof that the protocol is a reliable multicast and that it tolerates f out of $f + 2$ faults can be found in Appendix A.

4.3 Performance evaluation

The experimental setting used to evaluate the protocol consisted in the COTS-based implementation of the TTCB described in Chapter 3. More specifically the performance results were obtained on a system with six PCs, each containing a Pentium III processor running at 450 Mhz and 192 Mbytes of RAM. The operating system of all PCs was RTAI. The PCs were connected by two 100 Mbps Fast-Ethernet LANs, one for the general-purpose payload network and another for the internal control network of the TTCB (each PC had two network adapters). The protocol was implemented in C, compiled with the standard gcc compiler. The hash function used was MD5 (Menezes *et al.* , 1997, Chapter 9). Whenever possible, the communication among processes was based on IP multicast. Six processes were used in the tests, each one running on a distinct PC, and we assumed a setting where all processes were correct, i.e., no failed processes ($f = 0$). Throughout the experiments the value adopted for the omission degree was two ($Od = 2$). Each measurement was repeated at least 5000 times.

In the first set of experiments, we tried to determine in which phase the protocol terminated. From the observed results, it is possible to conclude that for reasonable values of T_1, in the order of 2 ms, the protocol always terminates in the first (optimistic) phase. We noticed that although IP multicast is unreliable, all messages apparently reached the processes, and for this reason, they were able to propose their hash value before $tstart$ (see Figure 4.1). If messages were lost or if some of the processes were malicious, the second phase would have had to be executed.

In the second set of experiments, we obtained message delivery times for the protocol. Since the protocol always finishes at the end of the first phase, it is possible to use the following methodology to calculate the delivery times. One of the processes is randomly selected as the sender, and then it reliably multicasts a message M of a given size. Then, immediately after delivery, a single recipient is selected to send a reply. This reply is an IP multicast for the same set of processes, with a message of the same size. For each execution of this procedure two times were measured: the round-

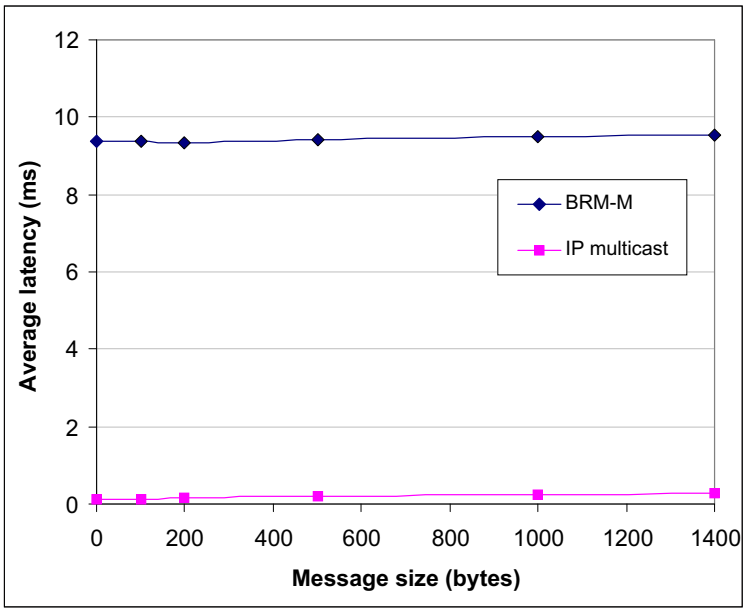

Figure 4.3: BRM-M average delivery time for different message sizes (6 processes).

trip time and the recipient processing time. The round-trip time (Trd) is obtained by the sender, and it corresponds to the time measured between the multicast and the reception of the reply. The recipient processing time ($Tproc$) is the time taken between the reception of the message M in the recipient and its reply. This time includes all tasks executed by the recipient, such as hash calculation, and it corresponds mostly to the time waiting for the TTCB TBA service, i.e., calling TTCB_propose and waiting for TTCB_decide to return the result of the agreement (lines 7-10). If one assumes that an IP multicast always takes the same amount of time, we can use the following formula to calculate the protocol's message delivery time:

$$Td = (Trd - Tproc)/2 + Tproc \tag{4.1}$$

Figure 4.3 plots the average delivery time of the protocol, with 6 processes, as a function of the message data size. These results are compared with the unreliable IP

Multicast (over UDP sockets) performance, also implemented in C and in the same environment of execution.

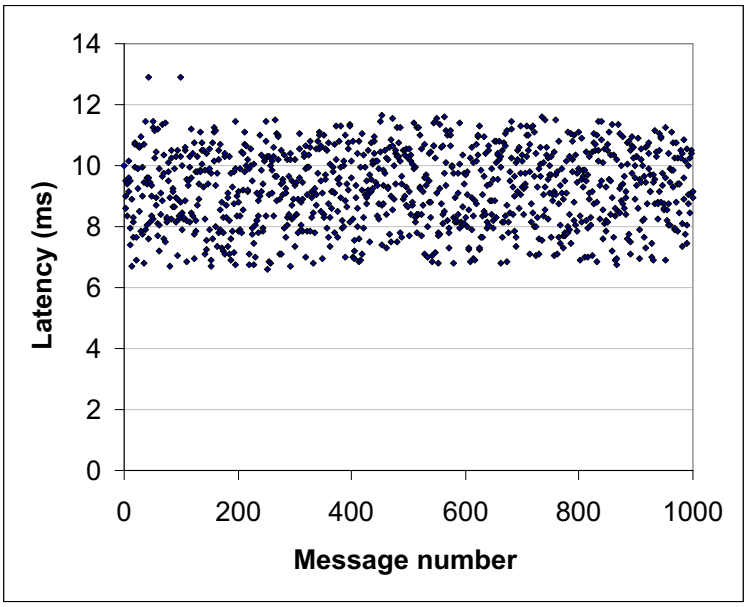

Figure 4.4: BRM-M delivery times for 1000 messages with a size of 0 bytes (6 processes).

The protocol overheads are mainly three: one IP multicast, some processing time (calculate the hash), and the execution of one TBA. Figure 4.3 shows that the additional cost of the protocol in relation to an unreliable IP multicast is approximately 9 ms on average. Since the processing time is in order of a few tens of microseconds, most of this cost corresponds to the waiting period due to the TBA execution. Consequently, we expect our protocol will perform better as the TTCB is optimized, and faster protocols are used to implement the TBA service. Nevertheless, it should be noticed that the current performance results seem already good when compared with other Byzantine resilient protocols that have been published in the literature. For instance, in (Reiter, 1994), for a group of 6 processes and message sizes of 0 and 1 Kbytes, the delivery times were approximately 53 and 57 ms, respectively. However, the test setting was different so the comparison should be taken with prudence.

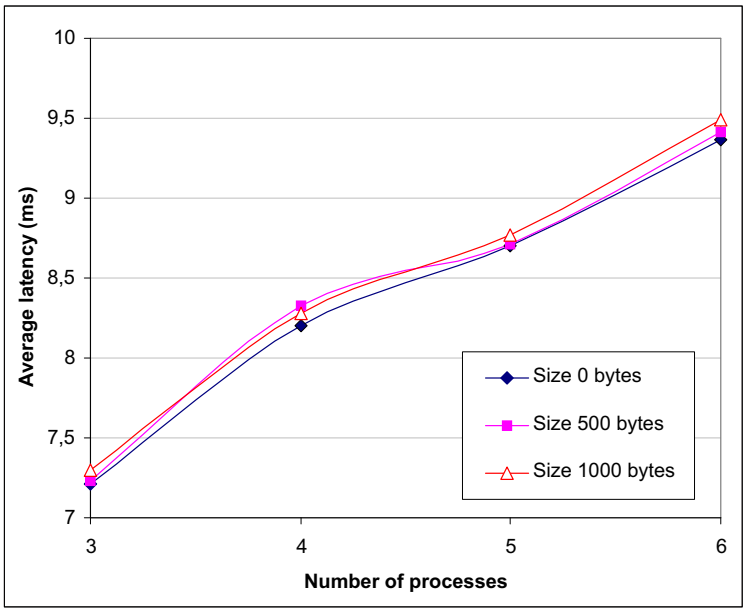

Figure 4.5: BRM-M average delivery times with 3 to 6 processes and different message sizes.

The delivery time values exhibit a reasonably high standard deviation, around 2.5ms. Figure 4.4 displays the delivery times for 1000 executions of the protocol using a message data size of 0 bytes. The main explanation for this behavior is related to the internal implementation of the TBA service of the TTCB. Currently, it uses a time-triggered protocol where interactions with the network only happen every 4 ms (e.g., it only reads messages from the network at the beginning of the 4 ms interval). Therefore, an agreement will take more or less time depending on the instant when processes propose their values within the 4 ms interval.

The third set of experiments analyzed the variation of the latency with the number of processes. The results are shown in Figure 4.5. The main conclusion is that the time does not increase much with the number of processes. The reason is that the main overhead of the protocol is the TBA execution, as discussed above. The main overhead of the protocol in (Reiter, 1994), on the contrary, is the time taken by the public-key

cryptography operations. The number of these operations increases with the number of processes, therefore the protocol time also increases considerably with this number.

4.4 Related work

There is a significant amount of work in the area of reliable broadcasts for distributed systems – most of it, however, has focused on benign failures and/or assumed a synchronous model (Hadzilacos & Toueg, 1994). Reliable multicast protocols tolerating Byzantine faults make no assumptions about the behavior of faulty processes (similarly to "Byzantine agreement" in the synchronous time model (Lamport *et al.* , 1982)). As for the number of processes, in asynchronous systems it was proved that less than a third ($f \leq \frac{n-1}{3}$) process may be corrupted (Bracha & Toueg, 1985). In our protocol, with the support of the TTCB, we can overcome this limit, and require only $f \leq n - 2$.

The Rampart toolkit contains a reliable multicast protocol where processes communicate through authenticated reliable channels and use public-key cryptography to digitally sign some of the messages (Reiter, 1994). The protocol is based on a simple echo protocol where the sender starts by multicasting a hash of the message, then it expects a confirmation from a subset of the processes, and finally it multicasts the message (this protocol improves the echo protocol by Toueg (Toueg, 1984) in terms of message complexity at the cost of more computation). Rampart assumes a dynamic membership provided by a protocol which also utilizes a three-phase commit strategy (Reiter, 1996b). Later, Malki and Reiter optimized the Rampart protocol using a method of chaining acknowledgments to amortize the cost of computing the digital signatures through several messages (Malkhi & Reiter, 1997b). Malkhi, Merrit and Rodeh proposed a secure reliable multicast protocol based on dissemination quorums, as a way to reduce delays especially in the case where $f \ll n$ (Malkhi *et al.* , 1997c). This protocol assumes similar channels and uses public key signatures as the previous protocols, but considers static membership, as also (Malkhi & Reiter, 1997b).

The SecureRing system provides a reliable message delivery protocol that uses public key cryptography and assumes a fully connected network (Kihlstrom *et al.* , 2001). The multicast is imposed on a logical ring, where a token controls who can send the messages. The Secure Trans protocol, which is implemented in the Secure-Group system, uses retransmissions and acknowledgments to achieve reliable delivery of messages (Moser *et al.* , 2000; Moser & Melliar-Smith, 1999). These acknowledgments are piggybacked on messages that are themselves broadcasted. Each message is digitally signed to ensure authenticity and integrity.

There are some secure group communication systems which consider a non-Byzantine fault model: Horus, Ensemble and Secure Spread. These systems assume that communication can be attacked but that hosts do not fail. Secure multicast protocols based on message authentication codes are given explicitly for Horus and Secure Spread (Reiter *et al.* , 1994; Amir *et al.* , 2000).

The BRM-M protocol does not need public key cryptography, one of the main bottlenecks of group communication performance (Castro & Liskov, 1999), since it uses the TTCB to securely exchange a digest of the message. In terms of the network, we assumed unreliable channels, therefore the message complexity is proportional to the omission degree.

The FLP impossibility result states that consensus in a distributed asynchronous system has the possibility of nontermination if a single process is allowed to crash (Fischer *et al.* , 1985). This result does not apply to our system model because we have a synchronous subsystem, the TTCB. However, reliable multicast is also a weaker problem than consensus, does not require agreement between the processes on the delivery of the messages, therefore it is not bound by this result (Malki & Reiter, 1996).

4.5 Summary

The chapter presents a new reliable multicast protocol for asynchronous systems with a hybrid fault model. This type of fault model allows some components to fail in a controlled way while others may fail arbitrarily. In our case, we assume the existence of a simple distributed security kernel, the TTCB, which can only fail by crashing, while the rest of the system can behave in a Byzantine way. By relying on the services of the TTCB, the protocol exhibits good behavior in terms of time and message complexity when compared with more traditional Byzantine protocols. Moreover, it only requires $n \geq f + 2$ correct processes, instead of the usual $n \geq 3f + 1$.

Besides describing a novel Byzantine-resilient protocol, the chapter introduces the design of protocols based on our architectural-hybrid fault model and, more specifically, the design of protocols using our distributed security kernel, the TTCB.

Note

The content of this chapter was partially published in (Correia *et al.* , 2002b). A protocol of the same family was published in (Lung *et al.* , 2003).

5

Consensus

Consensus is a classical distributed systems problem with both theoretical and practical interest. Over the years, several other distributed problems have been shown to be reducible or equivalent to consensus, for instance, total order broadcast (see, e.g., (Hadzilacos & Toueg, 1994)). Consensus has been studied in a large number of systems with different characteristics, such as the synchronous and asynchronous time models, with distinct types of failures ranging from crash to arbitrary (a survey of early work can be found in (Fischer, 1983)). On asynchronous systems, consensus has been shown to be constrained by the FLP impossibility result, which says that it is impossible to solve consensus deterministically in a completely asynchronous system (Fischer *et al.* , 1985). Consequently, various researchers have proposed several ways to circumvent this limitation, by using randomization techniques or by assuming weak synchrony assumptions on the behavior of the system (Rabin, 1983; Ben-Or, 1983; Bracha & Toueg, 1985; Dwork *et al.* , 1988; Chandra & Toueg, 1996).

This chapter presents a consensus protocol based on the TTCB and the hybrid fault model being considered. This protocol is not particularly more complex than the reliable multicast presented in the previous section. However, it shows a different way of using the TBA service, this time to agree on a hash proposed by a majority of processes. The chapter also shows how a protocol based on the TTCB relates to the FLP impossibility result.

5.1 System model

The system model considered in this chapter will remain the same for the next chapters, 6 and 7. In fact, it is basically the same as in the previous chapter. The system is still an asynchronous networked system with a TTCB, and the process failure model is basically the same. The difference is the communication model.

The protocol relies on channels that abstract some of the communication complexity. Each pair of processes is interconnected by a *secure channel*, defined in terms of two properties:

- *Eventual reliability:* if p and q are correct and p sends a message M to q, then q eventually receives M.

- *Integrity:* if p and q are correct and q receives a message M with *sender(M)* = p, then M was sent by p and M was not modified in the channel. [1]

Each pair of correct processes is assumed to share a symmetric key known only by the two. With this assumption, the two properties above can be implemented easily and efficiently. Eventual reliability is obtained by retransmitting the messages periodically until an acknowledgment is received. Message integrity is achieved by detecting the forgery and modification of messages through the use of Message Authentication Codes (MACs) (Menezes *et al.* , 1997). A MAC is basically a secure checksum obtained with a hash function and a symmetric key. They are about three orders of magnitude faster to calculate than digital signatures (Castro & Liskov, 1999). A process adds a MAC to each message that it sends, to allow the receiver to detect forgeries and modifications. Whenever such detection is made, the receiver simply discards the message, which will be eventually retransmitted if the sender is correct.

[1] The predicate *sender(M)* returns the sender field of the message header.

5.2 Consensus

This section describes a consensus protocol tolerant to Byzantine faults. For presentation simplicity, we start by explaining how to reach consensus on a value with a small number of bytes, and then this result is extended by removing this limitation.

The consensus protocol utilizes as building block the TBA service. The reader however, should notice that, as tempting as it might be, it is *not* possible to solve the consensus problem in the payload system simply by using the TBA service of the TTCB. In fact, the problem does not become much simpler because the protocol still needs to address most of the difficulties created by a Byzantine asynchronous environment. For instance, since the protocol runs in the asynchronous part of the system, it cannot assume any bounds on the execution of the processes, on the observed duration of the TTCB function calls, or on the message transmission times. Moreover, since processes can be malicious, this means that they might provide incorrect values to the TTCB or other processes, or they may delay or skip some steps of the protocol. What we aim to demonstrate is that the 'wormholes' model, materialized here by the TTCB, allows simpler solutions to this hard problem.

5.2.1 Consensus problem

The problem of consensus can be stated informally as: how do a set of distributed processes achieve agreement on a value despite a number of process failures? There are several different formal definitions of consensus in the literature. In the context of a Byzantine fault model in asynchronous systems, a common definition is (see, e.g., (Dwork *et al.* , 1988; Malkhi & Reiter, 1997c; Kihlstrom *et al.* , 2003)):

- *Validity.* If all correct processes propose the same value v, then any correct process that decides, decides v.
- *Agreement.* No two correct processes decide differently.

- *Termination.* Every correct process eventually decides.

Validity and Agreement properties must always be true otherwise something bad might happen. Termination is a property that asserts that something good will eventually happen. This distinction is important because our protocol has different requirements in terms of number of failed processes, in order to be respectively safe or live.

In case all correct processes propose the same value, Validity guarantees that it is the value chosen, even in the presence of alternative malicious proposals. In any other case, namely if correct processes propose different values, the consensus protocol is allowed to decide on any value, including on a value submitted by a malicious process.

5.2.2 Block consensus protocol

The *block consensus* protocol reaches consensus on a value with a limited number of bytes. When compared with other Byzantine-resilient consensus protocols, block consensus is quite simple since most of its implementation relies on the TBA service of the TTCB, and no information has to be transmitted through the payload channel. Nevertheless, it serves to illustrate two interesting features of our system model. First, it demonstrates that it is possible to construct a consensus protocol capable of tolerating arbitrary attacks based on an agreement protocol that was developed under the crash fault model. Since crash-resilient protocols are much more efficient than the Byzantine-resilient kind, we expect block consensus to exhibit very good performance. Second, it shows: (i) how a protocol running under the asynchronous model can interact with one running synchronously (in the TTCB); and (ii) how the protocol relates to the FLP impossibility result and how the addition of a weak synchrony assumption is required to guarantee termination (see Section 5.2.4).

The protocol is presented in Algorithm 4. The arguments are the list of the n processes involved in the consensus (*elist*), a timestamp (*tstart*), and the value to be proposed (*value*). *tstart* has to be the same in all processes. For the participants, this

requirement is similar to what is observed in other consensus protocols where all processes have to know in advance a consensus identifier. However, the identifier conveys a meaningful absolute time to the TTCB: processes despite being time-free, can agree on a value obtained from the Trusted Absolute Timestamping service to synchronize their participation to the consensus. The number of bytes of *value* should be the same as the size imposed by the TBA service (currently 20 bytes). In case it is smaller, padding is done with a known quantity (e.g., with zero). The number of processes which can fail are $f = \lfloor \frac{n-1}{3} \rfloor$ out of n.

Algorithm 4 Block consensus protocol (executed by every process).

```
1  function consensus(elist, tstart, value)
2  round ←0;                                                    {round number}
3  repeat
4     out_prop ←TTCB_propose(eid, elist, tstart, TBA_MAJORITY, value);
5     repeat
6        out_dec ←TTCB_decide(out_prop.tag);
7     until (out_dec.error ≠ TBA_RUNNING);
8     tstart ←tstart + T * func(α, round);                      {α ∈ [0, 1[}
9     round ←round+1;
10 until (f + 1 processes proposed the same value) or (2f + 1 processes proposed);
11 decide out_dec.value;
```

The protocol works in rounds until a decision is made. In every round, each process proposes a value to the TBA (line 4) and gets the result (lines 5-7). The value decided by TBA is the value proposed by most processes (decision function TBA_MAJORITY). The protocol terminates when one of the conditions is satisfied (line 10):

1. at least $f + 1$ processes proposed the same value v: this condition implies that at least one correct process proposed v. Therefore, either (1) all correct processes proposed v or (2) not all correct processes proposed the same value. In both cases, the protocol can terminate and decide v.

2. at least $2f+1$ processes proposed a value but no subset of processes with the same

value has a size larger than f: this condition implies that some correct processes proposed distinct values. In this case, the protocol can terminate and decide on any value. For example, our implementation will choose the most proposed value, if it exists.

Both conditions can be tested using the two masks returned by *TTCB_decide*. The first one is constructed with the *proposed-ok* mask and the second one can be evaluated with the *proposed-ok* and the *proposed-any* masks (Section 3.2.3). The TBA execution starts when either all processes have proposed a value or time reaches *tstart*. Block consensus assumes that eventually there is a round when *enough* processes manage to propose to the TBA before *tstart*. 'Enough' here is defined in terms of the two conditions that allow the protocol to terminate. The algorithm keeps retrying until this happens (lines 3-10).

The *tstart* of the next round is calculated by adding a quantity to the previous *tstart*, computed using constants T and α, and function $func$ (line 8): $func$ is a monotonically increasing function of $round$, where α controls the slope, $\alpha \in [0, 1[$. For example, linear ($func \equiv 1 + \alpha * round$), or exponential ($func \equiv (1+\alpha)^{round}$). Thus, by increasing the period of retry upon each repetition, we will eventually manage to get enough processes to propose. There is an interesting tradeoff here: with a larger *tstart* the probability of termination in real systems increases, since more time is given for proposals; on the other hand, if one process is malicious and does not propose, then a larger *tstart* will delay the execution of the TBA service, and consequently the consensus protocol. Incidentally, note that processes, being time-free, are totally unaware of the real-time nature of *tstart*, they just deterministically increase an agreed number, which is only meaningful to the TTCB.

Figure 5.1 illustrates an execution of the protocol in a system with four processes where P1 is malicious. In the example, P1 and P2 are able to propose on time for the first TBA. P4 starts on time, but is delayed for some reason (e.g., a scheduling delay) and proposes after *tstart(i)*. Therefore, it will get an error from the TBA service, and its value will not be considered in the agreement. P3 is also delayed, and only starts

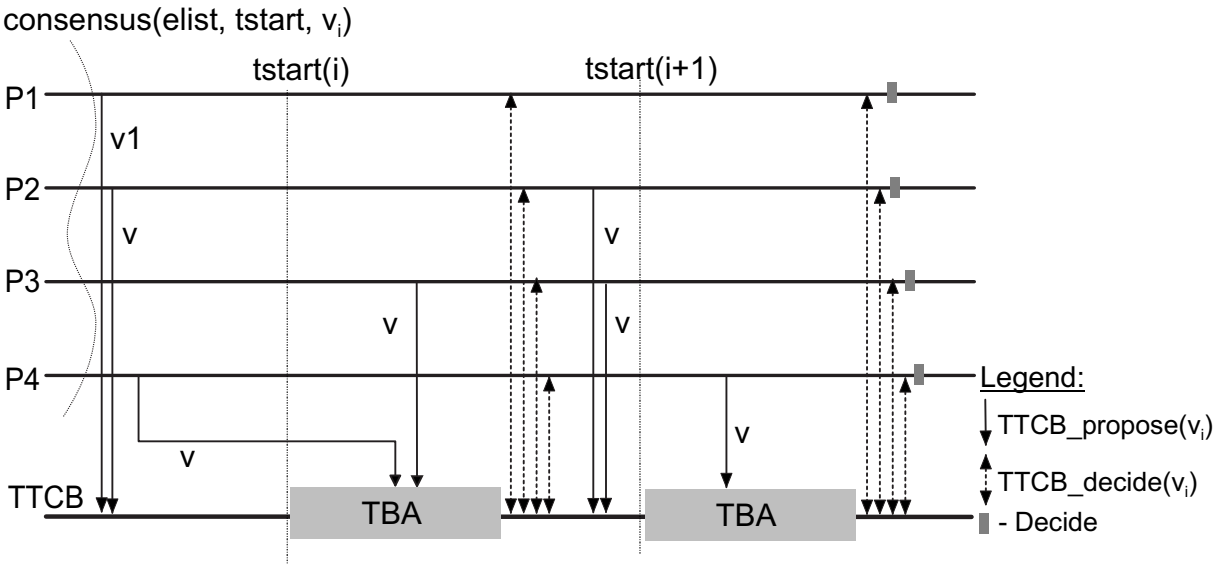

Figure 5.1: Block Consensus protocol example execution (with *n=4* and *f=1*).

to execute after *tstart(i)*, and consequently, its proposal is also disregarded. When the TBA finishes, all processes get the result, which in this case will be based on the proposals from P1 and P2. Since P1 is malicious, it attempts to force an incorrect decision by proposing $V1$ that is different from the value of the correct processes (which is V). Nevertheless, since none of the conditions is satisfied (line 10), another round is executed. Here, process P1 skips the proposal step, but two correct processes manage to propose before *tstart(i+1)*. In the end, they will all be able to decide, since the first condition will be true.

The correctness of the protocol is proved in Appendix A.

5.2.3 General consensus protocol

For presentation simplicity, we first described the block consensus protocol, which achieves agreement on a data value with at most the size of the TBA service block. This section presents a consensus protocol without this limitation. The *general consensus* protocol makes use of the payload channel to multicast the values being proposed, and then utilizes the TBA service to choose which value should be decided. The number of processes which can fail is also $f = \lfloor \frac{n-1}{3} \rfloor$ out of n.

The protocol is presented in Algorithm 5. The arguments have the same meaning as in the block consensus. Each process starts by initializing some variables, and then it multicasts the value through secure channels to the other processes (line 6). Next, the protocol works in two phases, where it runs a minimum of one round in the first phase, but depending on the values and on the timing of the proposals, it may need several rounds in both phases.

In the *first phase* processes propose to the TBA a *hash* of their own values (line 11). This phase and the protocol both terminate if $f + 1$ processes propose the same hash to the TBA (line 19). In this case, the value decided is the one that corresponds to that hash (lines 20, 23, and 26). Since $f + 1$ proposed the hash, then at least one of the processes has to be correct. Consequently, it is safe to use that value as the decision (the argument is equivalent to the first condition of block consensus). Moreover, since a correct process always starts by multicasting its value through reliable channels, then we can be sure that eventually all correct processes will receive the value, and will be able to terminate.

The protocol enters the *second phase* when $2f + 1$ processes proposed a hash but no subset greater than f proposed the same hash (lines 17-18). This situation only happens when the correct processes do not have the same initial value. In this case the definition (Section 5.2.1) allows any value to be chosen. The simpler solution would be to choose a pre-established value, e.g., zero. However, it is more interesting to make the protocol agree on one of the various values proposed. This is the purpose of the second phase.

The second phase uses a rotating coordinator scheme (Reischuck, 1982) where in each round a different process becomes the coordinator ($coord = round$ mod n), and then its value is selected as the (potential) decision.

Processes pick the value of the current coordinator to propose it to the TBA. If this value is not available (for instance, because it was delayed or the coordinator crashed), then it is necessary to choose another value. In our case, we decided to use a simple deterministic algorithm where a process goes through the *elist* until it finds

the first process whose message has already been received (implemented by function *nextSenderMesg()*, line 10). Basically, the process first tries to see if the message from $coord = elist[k \ mod \ n]$ has arrived, then it tries for $elist[(k+1) \ mod \ n]$, next for $elist[(k+2) \ mod \ n]$, and so on, until a message is found. There is the guarantee that at least one message will always exist because the initial multicast (line 6) immediately puts one message in the *bag* [2]. This algorithm has the interesting characteristic that it skips processes that did not manage to send their value, allowing the consensus to finish faster.

Since the value being decided might have been proposed by a malicious process, an extra precaution has to be considered. The malicious process might have sent the value just to a sufficiently large subset of processes to ensure that a decision could be made (e.g., f processes). Then, the rest of the processes would never get the decided value – they would only get the corresponding hash. To solve this problem, processes have to retransmit the value to the other processes (lines 24-25). The masks from *TTCB_decide* are used to determine which processes are these.

The correctness of the protocol is proven in Appendix A.

5.2.4 Termination and the FLP impossibility result

Fischer, Lynch and Paterson showed that consensus in an asynchronous system has the possibility of nontermination if a single process is allowed to crash (Fischer *et al.* , 1985). Throughout the years, several proposals have been made to circumvent this FLP impossibility result, for example, by using randomization (Rabin, 1983; Ben-Or, 1983) or by making partial synchrony assumptions (Dwork *et al.* , 1988). More recently, the concept of *unreliable failure detectors* was defined in order to hide this sort of assumptions (Chandra & Toueg, 1996).

The system we consider is not fully asynchronous but a combination of asyn-

[2]We use the word 'bag' to denote a set of messages. A bag does not have an order and does not store duplicated data.

Algorithm 5 General consensus protocol (executed by every process).

```
1  function consensus(elist, tstart, value)
2  hash-v ←⊥;                                                    {hash of the value decided}
3  bag ←⊥;                                                       {bag of received messages}
4  round ←0;                                                     {round number}
5  phase ←1;                                                     {protocol phase}
6  multicast(elist, tstart, value) to processes in elist;       {send value through payload channel}

7  loop
8    repeat
9      if (phase = 2) then {phase 1: use my value — phase 2: choose a value from a process}
10       value ←{M.value : coord = (round mod n) ∧ M=nextSenderMesg(coord, elist, bag)
         };
11     out_prop ←TTCB_propose(eid, elist, tstart, TBA_MAJORITY, Hash(value));
12     repeat
13       out_dec ←TTCB_decide(out_prop.tag);
14     until (out_dec.error ≠ TBA_RUNNING);
15     tstart ←tstart + T ∗ func(α, round);
16     round ←round+1;
17     if (2f + 1 processes proposed) and (less than f + 1 processes proposed the same value)
       then
18       phase ←2;
19   until (f + 1 processes proposed the same value);                {decision condition}
20   hash-v ←out_dec.value;

21   when receive message M
22   bag ←bag ∪ {M};

23   when (hash-v ≠ ⊥) and (∃M∈bag : Hash(M.value) = hash-v)
24   if (phase = 2) then
25     multicast M to processes in elist except those that proposed Hash(M.value);
26   decide M.value;
```

chronous (payload) and synchronous (TTCB), so FLP does not apply. The precise boundaries in terms of communication synchrony, hosts synchrony and message delivery order in which the impossibility exists were detailed in a paper by Dolev et al. (Dolev *et al.* , 1987). How does the *block consensus protocol* fit in the categories in that paper? The hosts are asynchronous but all communication is done using the TBA, therefore it is synchronous. The protocol does not receive messages but results of the TBA and all correct processes execute the same TBAs in the same order, therefore the communication is ordered. All processes receive the same results of the TBAs so the communication can be classified as 'broadcast'. The receive and send operations (decide/propose in this case) are not atomic. With this scenario the paper concludes that there is no bound on the number of faults that the protocol can tolerate, therefore FLP does not apply. The crucial issue is the communication being ordered; the result would be the same if the communication was asynchronous. In relation to the *general consensus protocol*, the same reasoning applies to the consensus about the hash of the value proposed, therefore FLP does not apply also.

To ensure the termination of the consensus protocol, it is necessary to make a weak synchrony assumption about the execution of the processes. The protocol, however, was built in such a way that even if this assumption is never verified, it never violates the Agreement and Validity properties.

The protocol is executed in rounds and in each round processes attempt to propose a value to the TBA service before *tstart*. If in one of the rounds *enough* processes are capable of providing their values on time, then they are able to exit the main loop, and complete the consensus protocol. Therefore, the assumption that guarantees termination is that: *eventually there will be a round where at least* $2f + 1$ *processes manage to call* TTCB_propose *before one of the* tstart *deadlines* [3]. This is a very weak assumption since it is only about the hosts (not the network) and it is required to *eventually* occur (it does not have to happen at a specific cycle).

[3] In the general consensus protocol, if correct processes propose different values, it is necessary two (non-contiguous) rounds.

5.3 Evaluation of the protocols

This section evaluates the two versions of the consensus protocol in terms of time and message complexity.

5.3.1 Time complexity

The time complexity of distributed algorithms is usually evaluated in terms of number of rounds or phases. Using this method, the two versions of the protocol described take one round in the best case, i.e., in a run where no failures occur. However, since these criteria can be ambiguous, Schiper introduced the notion of *latency degree* (Schiper, 1997). The idea is based on a variation of Lamport's logical clocks which assigns a number to an event (Lamport, 1978), with the following rules:

1. send/multicast and local events at a process do not change its logical clock;
2. the timestamp carried by message M is defined as $ts(M) = ts(send(M)) + 1$, where $ts(send(M))$ is the timestamp of the $send(M)$ event;
3. the timestamp of a $receive(M)$ event on a process p is the maximum between $ts(M)$ and the timestamp of the event at p immediately preceding the $receive(M)$ event.

We extended the notion to systems with a TTCB, by introducing a new set of rules:

4. a *TTCB_propose* event at a process does not change its logical clock value;
5. the timestamp associated to an execution of the TBA service A is defined as $ts(A) = ts(TTCB\_propose(A)) + 1$, where $ts(TTCB\_propose(A))$ is the largest timestamp of the $TTCB\_propose$ events performed for A;
6. the timestamp of a $TTCB\_decide(A)$ event on a process p is the maximum between $ts(A)$ and the timestamp of the event at p immediately preceding the $TTCB\_decide(A)$ event.

These new rules were defined considering the current implementation of the TBA protocol. The protocol consists basically in every local TTCB sending the value proposed by its local process(es) to the other local TTCBs. Applying the original rules for send and receive events (rules 1-3), we derive the rules for *TTCB_propose* and *TTCB_decide* (rules 4-6).

Let us now define latency degree. For an execution of a consensus algorithm $\mathcal{C}$, the *latency* of $\mathcal{C}$ is the largest timestamp of all *decide* events. The *latency degree* of $\mathcal{C}$ is the minimum possible latency of $\mathcal{C}$ over all possible executions (Schiper, 1997).

Now we calculate the latency degree for both consensus protocols applying the rules above. The logical clocks start with 0 at every process.

- *Block consensus protocol:* (1) the TBA has $ts(A) = 1$ (rules 1, 4, 5); (2) *TTCB_decide(A)* has a timestamp of 1 at every host (rule 6); (3) every process decides at line 11 with that logical clock value so the latency degree of the protocol is 1.

- *General consensus protocol:* All correct processes with same value: (1) multicast at line 6 has $ts(M) = 1$ (rules 1, 2); (2) the TBA which is started at line 11 has also $ts(A) = 1$ (rules 1, 4, 5); (3) if a process receives a message, the timestamp is 1 (rule 3); (4) all processes decide with a logical clock value of 1 (rule 6), and therefore the latency degree is 1. Correct processes with distinct values: (1) (2) and (3) are the same; (4) processes enter in phase 2 and execute another TBA with $ts(A1) = 2$ (rules 4, 5); (5) all processes decide with a logical clock value of 2 (rule 6), and therefore the latency degree is 2.

Table 5.1 compares the latency degrees of both versions of the protocol with other asynchronous Byzantine-resilient protocols that solve similar consensus problems. Our protocols have the best latency degree. The translation into execution time is far from trivial (Keidar, 2002), but in our case we can say that the best case execution time of the protocols is the time for executing a single TBA, which is in the order of 4 ms with the current TTCB implementation. Although we are not aware of any measurements

Protocol	Latency degree	Requirements
Dwork et al. (Dwork *et al.* , 1988)	4	Signed messages
Dwork et al. (Dwork *et al.* , 1988)	7	–
Malhki & Reiter (Malkhi & Reiter, 1997c)	9 or 6	Signed messages
Kihlstrom et al. (Kihlstrom *et al.* , 2003)	4	Signed messages
Block consensus	1	TTCB
General consensus	1 or 2	TTCB

Table 5.1: Latency degrees for some Byzantine-resilient consensus protocols.

of consensus execution times, protocols that rely on signatures have to use public-key cryptography, and therefore they are allegedly slower than ours.

In the presence of process failures, both versions of the protocol also have small latency degrees because they are mostly decentralized. Block consensus continues to have a latency degree of 1, and General consensus has a latency degree of 1 in case all correct processes start with the same value. The other protocols presented in Table 5.1 are all based on a (rotating) coordinator scheme, and therefore, their performance might be affected by the failures (e.g., the first coordinators are all malicious). For instance, the latency degree of the protocols by Dwork et al. (Dwork *et al.* , 1988) can be as high as $4(f + 1)$ for the protocol with signed messages, and $6(f + 1) + 1$ for the other protocol.

5.3.2 Message complexity

The message complexity of a protocol is evaluated in terms of the number of transmissions in the payload channel. Both versions of the protocol have the additional cost of performing TBAs which use the control channel. Table 5.2 shows the total number of messages sent by our protocols in the payload channel, considering the cases when a multicast is a single message (label "multicasts"), or when it is $(n - 1)$ "unicasts" (plus a local delivery) of the same message.

	Best case			Worst case		
Protocol	Multicasts	Unicasts	TBAs	Multicasts	Unicasts	TBAs
Block consensus	0	0	1	0	0	no limit
General consensus	n	$n(n-1)$	1	$2n$	$n(n-1)+$ $+n(n-f-1)$	no limit

Table 5.2: Message complexities for the consensus protocols.

5.4 Related work

The past twenty years saw several variations of the consensus problem presented in the literature. Consensus protocols can decide on a 0 or 1 bit (binary consensus), on a value with undefined size (multi-value consensus), or on a vector with values proposed by several processes (vector consensus or interactive consistency). Several Byzantine-resilient consensus protocols were proposed, using different techniques to circumvent FLP.

Recently several works applied the idea of Byzantine failure detectors to solve consensus (Malkhi & Reiter, 1997c; Kihlstrom *et al.* , 2003; Doudou & Schiper, 1997; Doudou *et al.* , 2002; Baldoni *et al.* , 2000). All these protocols use signatures implemented with public-key cryptography. Any process p can generate a signature $S(p, v)$ that cannot be forged, but which other processes can test. Likewise, they are all based on a rotating leader/coordinator per round. Malkhi and Reiter presented a binary consensus protocol in which the leader waits for a number of proposals from the others, chooses a value to be broadcasted and then waits for enough acknowledgments to decide (Malkhi & Reiter, 1997c). If the leader is suspected by the failure detector, a new one is chosen and the same procedure is applied. The same paper also described a hybrid protocol combining randomization and an unreliable failure detector. The protocol by Kihlstrom et al. also solves the same type of consensus but requires weaker communication primitives and uses a failure detector that detects more Byzantine failures, such as invalid and inconsistent messages (Kihlstrom *et al.* , 2003).

Doudou and Schiper present a protocol for vector consensus based on a *muteness failure detector*, which detects if a process stops sending messages to another one (Doudou & Schiper, 1997). This protocol is also based on a rotating coordinator that proposes an estimate that the others broadcast and accept, if the coordinator is not suspected. This muteness failure detector was used to solve multi-value consensus (Doudou *et al.* , 2002). Baldoni et al. described a vector consensus protocol based on two failure detectors (Baldoni *et al.* , 2000). One failure detector detects if a process stops sending while the other detects other Byzantine behavior.

Byzantine-resilient protocols based on partial synchrony assumptions, both with and without signatures, were described by Dwork et al. (Dwork *et al.* , 1988). The protocols are based on a rotating coordinator. Each phase has a coordinator that locks a value and tries to decide on it. The protocols manage to progress and terminate when the system becomes stable, i.e., when it starts to behave synchronously.

Other techniques were also used to circumvent FLP in Byzantine-resilient consensus protocols. Randomized/probabilistic protocols can be found in (Bracha & Toueg, 1985; Cachin *et al.* , 2000). More recently, the condition-based approach was introduced as another means to circumvent FLP (Mostefaoui *et al.* , 2001; Friedman, 2002). Protocols based on this approach satisfy the safety properties but termination is guaranteed only if the inputs verify certain conditions.

5.5 Summary

This chapter defines two versions of a novel consensus protocol. Both versions have very low time and message complexities (latency degree is at least twice as good as the other protocols analyzed), and they do not require public-key cryptography, which is currently considered one of the most important sources of overhead of Byzantine-resilient protocols. The chapter also shows a new way to use the TTCB TBA service to implement intrusion-tolerant protocols. The idea is to use this service to make a voting on the values proposed by the processes, and to decide when enough

processes voted the same. Finally, the chapter has shown how a TTCB-based protocol manages not to be bound by the FLP impossibility result. However, a weak synchrony assumption is required for termination. Future work will be made on trying to avoid the need for this assumption, something that we envisage to be possible using the wormhole model.

Note

The content of this chapter was partially reported in (Correia *et al.* , 2003a; Correia *et al.* , 2003b).

6

Membership service

Group communication is a well known paradigm for data transmission among distributed sets of hosts or processes. The membership service is the component in charge of keeping an updated list of the group members, processing joins and leaves of the group, and assessing the failure of hosts/processes. This chapter is about the design of an intrusion-tolerant membership service. The service has to continue to provide correct results despite intrusions on a number of hosts, possibly with malicious behavior of their membership service code, and attacks in the network (e.g., delay, modification, or replay of messages).

Most work in group communication, and specifically in membership services, has considered only crash failures (see Section 2.4). Recently, interest emerged in the problem of designing membership services for environments that might suffer arbitrary faults, including attacks and intrusions. We are aware of only three intrusion-tolerant membership services, all for asynchronous systems: Rampart (Reiter, 1996b), SecureRing (Kihlstrom *et al.* , 2001) and SecureGroup (Moser *et al.* , 2000; Moser & Melliar-Smith, 1999). Project ITUA also defined and implemented an improved version of the Rampart's service (Ramasamy, 2002).

The membership service is based on our hybrid fault model. The TTCB allows the membership service to have interesting features when compared with similar services in the literature. First it seems to be around one order of magnitude faster. One of the reasons is that it does not use public-key cryptography, a well known bottleneck in these services. Second, its performance seems to degrade slower with the number of hosts involved. Third, it makes decisions in a distributed way, i.e., the service does not

rely on a leader, contrary to most services that have been proposed previously. In these services, the failure of a leader has to be detected using timeouts, and in consequence, a hacker can delay the service by postponing the communication and creating false failure suspicions of successive leaders.

6.1 System model

This chapter is about a membership service for groups of hosts. From now on, the word *site* will be used to denote in an abstract way the software component that executes the service in a host (recall the MAFTIA middleware architecture in Figure 2.5). In practice, the site can be implemented, for instance, as a Unix process. The communication model considered is the same as in the consensus chapter (Section 5.1). The site failure modes are also the same as the process failure modes in the consensus chapter, basically the same as in Section 4.1. The membership protocol assumes that $f \leq \lfloor \frac{|V^n|-1}{3} \rfloor$ sites can fail in a given membership V^n. Notice that f is not a constant but is defined for every membership V^n in terms of the number of sites in it, $|V^n|$.

Wide-area networks are prone to link failures and other communication fluctuations. Such effects can lead to network partitions, i.e., to the virtual separation of the network in several subnetworks that are temporarily unable to communicate. This may cause the temporary division of a group in two or more subgroups with possibly different sizes. To handle this type of failures, the membership service uses a primary partition model (Birman, 1997), in which only one of the subgroups is allowed to make progress. Sites belonging to the rest of the subgroups are eventually removed from the primary partition subgroup, and forced to exit. We have to utilize a primary partition model because in an arbitrary failure environment, progress can only be made in subgroups that have at least $2f + 1$ elements of the original group. After one or more partitions, at most one subgroup in that condition exists.

6.2 Properties of the Membership Service

A membership service handles basically three operations: the addition of members to a group, the removal of failed members, and the removal of members by their own initiative. These operations will be called respectively *join*, *remove* and *leave*. The failure of a site is detected in every host by a *failure detector* module (see Section 6.3.6). This component is regarded as part of the *site* abstraction.

The membership service generates *views*, i.e., numbered events containing the group membership. A new view is installed whenever the membership is changed due to a member join, leave or removal. A group of sites with a single member is created when the first member joins and installs the first view. A site S_j sees a view as an array V_j^n containing one entry per each member site. The index n reflects the n^{th} view of the group. The service guarantees that each correct site has the same view at every instant of logical time, i.e., after the installation of the same (totally ordered) views in every site. The membership service executes the *membership protocol*, which is defined formally in terms of the following properties (similar to (Reiter, 1996b)):

- *Uniqueness.* If views V_i^n and V_j^n are defined, and sites S_i and S_j are correct, then $V_i^n = V_j^n$.
- *Validity.* If site S_i is correct and view V_i^n is defined, then $S_i \in V_i^n$ and, for all correct sites $S_j \in V_i^n$, V_j^n is eventually defined.
- *Integrity.* If site $S_i \in V_i^n$ and V_i^{n+1} is not defined then at least one correct site detected that S_i failed or S_i requested to leave. If site $S_i \in V_i^{n+1}$ and V_i^n was not defined at S_i then at least one correct site authorized S_i to join.
- *Liveness.* If $\lfloor \frac{|V^n|-1}{3} \rfloor + 1$ correct sites detect that S_i failed or receive a request to join, or one correct site requests to leave, then eventually V^{n+1} is installed, or the join is rejected.

Uniqueness guarantees that all correct sites in a group see the same membership. Validity ensures that if a view is defined at a site then the site is in the view (often

called Self-Inclusion property). Validity also guarantees that every correct site in a view eventually installs the view. Integrity prevents malicious sites from removing or adding sites to the group. Liveness ensures that a new view is installed when a number of correct sites detect a failure, or a correct site wants to join or leave.

6.3 Membership protocol

The *Membership Protocol* is a finite state machine that evolves at each site in two states: *Normal* and *Agreement*. When a site joins a group it enters the Normal state. This is the state where the system is supposed to be most of the time, and where sites may communicate normally. Then, when another site wants to join or leave, or when a site is suspected to have failed, certain *events* are generated and the protocol changes to the Agreement state. In this state, the sites of the current view try to agree on the next view, by running the *View Change Agreement protocol* (VCA). When VCA terminates, the new view is installed and the state changes back to Normal.

The protocol handles three events corresponding to the three operations mentioned above: join, remove and leave. This section describes the service in terms of generic events $Ev(S_j)$, i.e., event Ev about site S_j. For instance, Ev can be the event generated by the failure detector in a site indicating that S_j failed and should be removed. Later in the chapter details are given about the correspondence between the generic and the concrete events.

6.3.1 Example execution

As a first insight of the execution of the protocol, we will present an example of a site removal by the membership service (see Figure 6.1). Initially, the group has four sites, S1 to S4. S4 is malicious and performs malicious actions that are detected by the (Byzantine) failure detectors of sites S1 and S2. When this happens, S1 and S2 multicast a *(INFO, Remv(S4))* message saying that S4 should be removed from the group. Even if

S3 does not detect the misbehavior of S4, when it gets $f + 1 = 2$ messages stating that S4 should be removed, it knows that at least one correct site detected the failure, since at most $f = 1$ sites can fail and "lie". Therefore, when S3 receives the second *(INFO, Remv(S4))* message it also multicasts the same information.

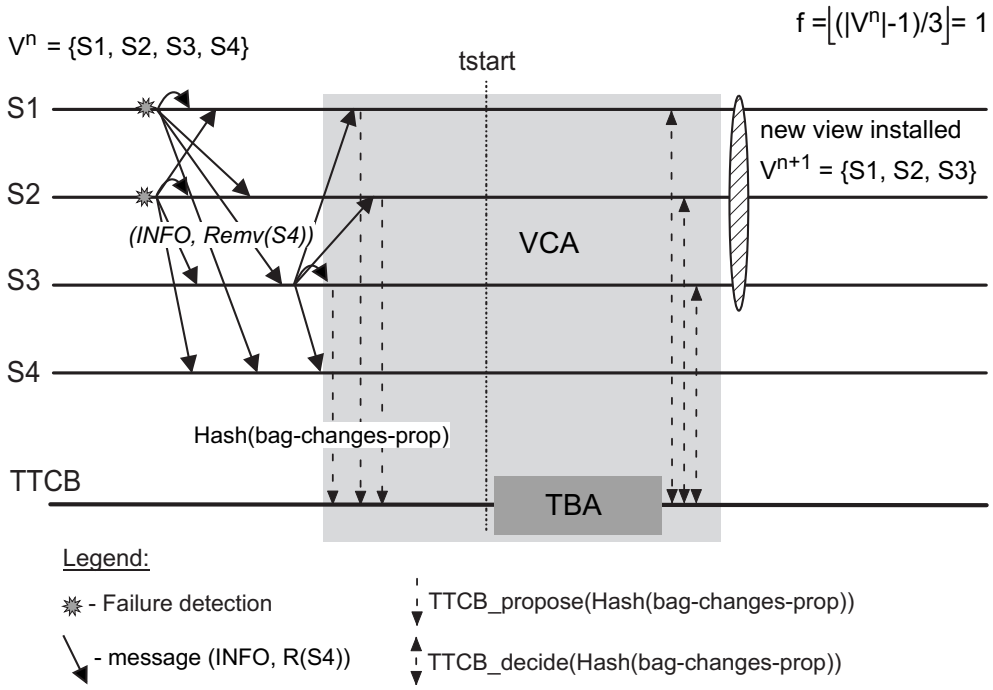

Figure 6.1: Membership service example execution.

When a site receives $2f + 1 = 3$ messages saying S4 failed it knows that all correct sites will also receive 3 or more messages (justification in the next section). Therefore it can move to the Agreement state with the confidence that all correct sites will do the same. It can put *Remv(S4)* in a bag (called *bag-changes*), where it saves all the changes that have to be applied to the current view, also knowing that all correct sites will do the same.

In the Agreement state the sites execute the VCA protocol. The objective is to make all correct sites decide the same changes to the view. The protocol uses the TBA service of the TTCB to agree on a digest of the *bag-changes-prop* (which is approximately equal

to *bag-changes*). In the example, S1 to S3 propose identical digests – they have the same *Remv(S4)* event in the bag – and TBA returns that digest, since it decides the most proposed value. Next, the new view is installed and S4 is removed.

6.3.2 The protocols

The membership service is implemented using two protocols. The basic membership protocol is described first (Algorithm 6) and the VCA protocol is presented next (Algorithm 7). Throughout the following discussion it is assumed that each message carries the current view number. The communication channels only deliver messages that were transmitted in the current view. Messages from the past, i.e., messages that were sent in a previous view, are discarded, and messages from future views are stored for later delivery. The correctness proof of the protocols can be found in Appendix A.

Whenever a site finds out that a new event $Ev(S_j)$ has occurred, it sends an INFO message – *(INFO, myid,* $Ev(S_j)$*, valid-tstart-send)* – to all sites in the current view (including itself). There are two ways a site can learn about new events: (1) it "sees" the event by itself, e.g., it detects the failure of S_j (lines 6-10 in the algorithm); or (2) it receives *(INFO, *,* $Ev(S_j)$*, *)* messages from $f + 1$ sites, which means that at least one correct site "saw" the event [1] (lines 11-16). In the message, *myid* is the site identifier and *valid-tstart-send* is a timestamp (discussed later). Sites put the INFO messages that arrive in the bag *bag-info* (lines 11-12). Function *count(*$Ev(S_j)$*, bag-info)* counts the number of INFO messages with $Ev(S_j)$ received from different sites (line 13).

When a site receives *(INFO, *,* $Ev(S_j)$*, *)* messages from $2f + 1$ different sites (line 17), it knows that all correct sites will receive at least that number of messages because: (1) if the site received $2f + 1$ messages then every correct site will eventually receive at least $f + 1$ messages (since at most f sites can fail); (2) when these correct sites receive these $f + 1$ messages they will also multicast (lines 13 and 16). Therefore, when a site receives $2f + 1$ INFO messages about $Ev(S_j)$ it can put $Ev(S_j)$ in another bag, called

[1] The star '*' is a wildcard that indicates any value.

Algorithm 6 Membership protocol.

```
1  INITIALIZATION:
2  bag-info ←⊥;                                          {bag with INFO messages}
3  bag-changes ←⊥;          {bag with changes the site wants to be done to the view}
4  valid-tstart-send ←⊥;        {valid tstart to send in INFO messages in this view}
5  state ←NORMAL;                                                {protocol state}

6  WHEN Ev(S_j) DO
7  if (I did not multicast (INFO, myid, Ev(S_j), *) in this view) then
8     if (valid-tstart-send = ⊥) then
9        valid-tstart-send ←next-valid-tstart();
10    multicast (INFO, myid, Ev(S_j), valid-tstart-send);

11 WHEN M = (INFO, sender-id, Ev(S_j), valid-tstart) received DO
12 bag-info ←bag-info ∪ M;
13 if (count(Ev(S_j), bag-info) ≥ f+1) and (I did not multicast (INFO, myid, Ev(S_j), *) in this
   view) then
14    if (valid-tstart-send = ⊥) then
15       valid-tstart-send ←next-valid-tstart();
16    multicast (INFO, myid, Ev(S_j), valid-tstart-send);
17 if (count(Ev(S_j), bag-info) = 2f+1) then
18    bag-changes ←bag-changes ∪ Ev(S_j);
19    if (state = NORMAL) then
20       state ←AGREEMENT;
21       execute vca( smallest-tstart(bag-info) );

22 WHEN bag-changes-done = vca(tstart) returned DO
23 add and remove sites in bag-changes-done from view;
24 view-number ←view-number + 1;
25 bag-info ←⊥; bag-changes ←⊥; valid-tstart-send ←⊥;
26 send state to new members;
27 state ←NORMAL;
```

bag-changes, with the confidence that all correct sites will eventually do the same.

At this stage, the site goes to the Agreement state if it is still in the Normal state (lines 19-21). INFO messages carry a *valid-tstart-send* parameter. When a site executes the VCA protocol, it gives as argument the smallest *valid-tstart-send* that was received – function *smallest-tstart(bag-info)* returns this value (line 21). The meaning of *valid-tstart-send* will be made clear in the next section. If the site was already in the Agreement state, then it simply updates the *bag-changes* with the new event (line 18), to ensure that this event is included in the next TBA execution. When the VCA protocol decides a value, i.e., a set of view changes, some housekeeping is performed and the state goes back to Normal (lines 22-27).

An event is only considered for agreement if $2f + 1$ or more sites have shown that they know about it. Until this quorum is reached, the event is simply stored for later processing. Consequently, there might be some events that still may need to be dealt with when a new view is installed. The solution that was chosen for this problem requires that these events are re-issued in the next view(s), until they are eventually processed. This solution is relatively simple to implement because it only requires that sites re-send their requests (in case of joins or leaves), or that the failure detector re-indicates the failure of a site.

6.3.3 View change agreement protocol

The previous section explains how sites decide to engage in the VCA (View Change Agreement) protocol (Algorithm 7). VCA runs as a series of executions of the TTCB TBA service, each one trying to agree in which way the current view needs to be updated. In the best case, which corresponds to the most common scenario, only one TBA is executed, as illustrated in the example of Figure 6.1.

The core of the protocol is presented in lines 8-15. Each site basically goes on proposing to successive TBAs the changes it thinks that have to be applied to the current view – the sites to add/remove. These updates are put in *bag-changes* by the mem-

Algorithm 7 View Change Agreement protocol (VCA).

```
1  FUNCTION vca(tstart)
2  hash-v ←⊥;                                              {hash of the value decided}
3  bag-msgs-change ←⊥;                                     {bag for messages received with view changes}
4  elist ←list with eid's of sites in V^{view−number} in ascending order;
5  if (tstart < last-tstart) then
6     tstart ←last-tstart;

7  loop
8     repeat
9        bag-changes-prop ←bag-changes;   {stores changes since they can be updated during
         VCA execution}
10       outp ←TTCB_propose(elist, tstart, TBA_MAJORITY, Hash(bag-changes-prop));
11       repeat
12          outd ←TTCB_decide(outp.tag);
13       until (outd.error ≠ TBA_RUNNING);
14       tstart ←tstart + T_{tstart};
15    until (at least 2f + 1 sites proposed the value decided);
16    last-tstart ←tstart;
17    if (outd.value = Hash(bag-changes-prop)) then
18       multicast (CHANGES, myid, bag-changes-prop) to sites not in outd.proposed-ok and
         not being removed;
19       return bag-changes-prop;                          {terminates the algorithm}
20    else
21       hash-v ←outd.value;

22    WHEN M=(CHANGES, *, *) received DO                   {changes message for this vca}
23    bag-msgs-change ←bag-msgs-change ∪ {M};

24    WHEN (hash-v ≠ ⊥) and (∃_{M∈bag−msgs−change} : Hash(M.bag-changes) = hash-v) DO
25    return M.bag-changes;                                {terminates the algorithm}
```

bership protocol (see previous section). In fact, VCA gives TBA a hash of *bag-changes* rather than the actual bag (function *Hash* in line 10) because the TTCB imposes a limit on the size of the values that it accepts. A hash function has, among other properties, the characteristic that is produces a fixed size digest of its input with the guarantee that it is computationally infeasible to discover another input that gives the same output (Menezes *et al.* , 1997, Chapter 9). The current implementation of the TTCB bounds the values to 160 bits, which is enough for standard hash functions like MD5 or SHA-1. Notice that we assume two sites obtain the same hash of *bag-changes* if their two bags have the same content. This is true only if the binary representations of the two bags are identical. This goal can be achieved by representing data in some canonical form.

TBA collects the values given by the sites, and chooses and returns the most frequently proposed value (TBA_MAJORITY decision function). It also returns a mask, *proposed-ok*, indicating which sites gave the value that was decided (line 12). Sites go on engaging in TBAs until a set with at least $2f + 1$ elements proposed similar values (line 15). This loop is assured to eventually terminate because all correct sites (at least $2f + 1$) will eventually get the same values in *bag-changes* (see previous section); therefore they eventually propose identical hashes and this is the decision value (since they are the majority). TBA is executed inside the TTCB so its results are reliable and all correct sites receive the same output.

If a site has the *bag-changes-prop* corresponding to the chosen hash (line 17), it sends that bag to the sites that did not propose the correct hash, to ensure that all correct sites get the changes to the view (line 18). These sites might, for instance, have more events in their *bag-changes-prop* than the others. Therefore, unless they are informed about the necessary updates, they do not know how to move to the next view. The processing of CHANGES messages is done in lines 20-25.

This basically concludes the presentation of VCA. However we still have to discuss the *tstart* parameter passed to the TBAs.

The parameter *tstart*

A correct site can only determine which *bag-changes-prop* should be applied to the current view if it is able to participate in the VCA protocol. The site, however, is not required to engage in all TBAs that potentially might be executed. It only needs to be involved in the TBA that provides the decision accepted by the $2f + 1$ or more sites (line 15), which is called the *last TBA*. With this decision, the site obtains an hash of the *bag-changes-prop* that has to be applied. Therefore, it becomes capable of selecting the correct *bag-changes-prop* either from the various that might arrive (lines 22-23, note that a CHANGES message can have invalid content since it might come from a malicious site) or from its own *bag-changes-prop* (line 17).

The participation of a site in a last TBA does not have to be necessarily active, in the sense that it might call *TTCB_propose* after the time indicated by *tstart*. In this case, the proposal is not accepted by the TTCB, but the site obtains the *tag* of that TBA execution, and eventually gets the decision (lines 11-13). This decision was calculated using the proposals of the other sites that arrived to the TTCB on time.

Therefore, there are two conditions that must be verified in order for a site to obtain the result of the last TBA:

1. Sites must agree on the values that may be used for *tstart*.
2. The value of *tstart* when a site enters *loop* should be less or equal than the *tstart* of the last TBA.

The first condition comes from the requirement that sites that want to participate in the same TBA have to provide to the TTCB identical *tstart* values (besides the same *elist* and TBA decision function). Reaching this agreement might seem hard to accomplish since it involves a consensus among a set of sites. In our case, however, we solve this problem by restricting the values of *tstart* that might be used. Sites can only utilize *valid tstart* values, which are defined as follows:

Valid tstart: any timestamp in the set $\{\forall_{k \in N}, k.T_{tstart}\}$, where T_{tstart} is the interval between valid *tstarts*.

The value of the T_{tstart} constant involves the tradeoff: if T_{tstart} is too low, on average more TBAs will be used to reach agreement but VCA will usually terminate faster; if T_{tstart} is too high, on average the contrary will happen: less TBAs but VCA might take longer to terminate.

The second condition is necessary to guarantee that a site will eventually be able to participate and obtain the result of the last TBA, even if later than its actual execution (the TTCB keeps a record of this decision). This condition is ensured if sites initiate the VCA protocol with the smallest *tstart* of the INFO messages (lines 8-9, 14-15, and 21 of Algorithm 6).

The following reasoning can be used to understand why this requirement implies the second condition. In a VCA execution, the first TBA in which a site engages has a *tstart* greater or equal than the *tstart* that was passed as argument. The last TBA has the participation of $2f + 1$ or more sites. Therefore, the *tstart* of the last TBA is greater or equal than the initial *tstarts* of all the sites that actively participated in the agreement. On the other hand, by our requirement, a site selects the smallest *tstart* from the $2f + 1$ INFO messages that were received, before entering the VCA protocol (Algorithm 6, line 21). Consequently, since the intersection of the set of sites that actively participated in the last TBA and the set of sites that sent the INFO messages has at least one element (f sites can be malicious and "lie"), it is possible to conclude that the initial *tstart* of all correct sites is smaller or equal than the *tstart* of the last TBA.

A malicious site could attempt to delay the protocol by providing an INFO message with a "very small" valid *tstart*, i.e., a timestamp that had passed a long time ago. To prevent this type of attack, the *loop* is always initiated with a *tstart* larger than the *tstart* of the previous VCA execution (lines 5-6 and 16). The reader should notice that this attack does not cause any incorrect behavior from the protocol – it simply delays the execution.

6.3.4 Site leave

The membership protocol was described in terms of generic events $Ev(S_j)$. Now let us see the corresponding real events, starting with the event related to the removal of a member by its own initiative.

A site S_j can decide to leave a group for several reasons, for example, because the user wants to shutdown its machine. In general this decision is taken by a higher level software layer. When that happens the site multicasts a message *(LEAVE, myid)* to all sites in the group (including itself). The reception of this message is the leave event *Leave(S_j)*. This event is then handled normally by the membership protocol, as described in the previous sections. Notice that a malicious site cannot remove a correct site S_j by sending a *(LEAVE, S_j)* message because the communication channels have the integrity property (see Section 5.1).

6.3.5 Site join

In the crash fault model, a site that wants to join a group has simply to find a "contact" with the information about the group membership. The contact can be any member of the group or a third party of some kind. In the Byzantine fault model, the problem is more complex since individual sites or other entities may provide erroneous information. For instance, if a site that wants to join asks the current view from a malicious site S_i, then S_i could return a group composed exclusively of malicious members. Therefore, the implementation of the join operation in an arbitrary failure environment requires the resolution of two sub-problems: first, it is necessary to determine who should be contacted; second, if several answers are received with the information about the group, it is essential that the correct one is selected.

There are two generic solutions for both problems. Either one assumes the existence of a reliable (i.e., trusted) well-known source, or one has to contact a set of sites and assume that a majority of $2f + 1$ of them are correct. Specific examples of these methods are:

- The system administrator manually provides a list of the current group members.
- There is a trusted third party server that always returns correct results.
- There is a known number of n potential member sites from which no more than f can fail ($n \geq 3f+1$). Each site, even if not a current member of the group, stores membership information and provides it when requested. The correct answer is chosen by doing a majority voting.

Independently of the selected approach, it is assumed that a joining S_j site manages to obtain the current view of the group, V^n. Then, S_j multicasts a message *(REQ_TO_JOIN, myid, auth-data)* to all sites in V^n. *auth-data* is application dependent authorization information that is independent of the membership protocol. Therefore, when the *REQ_TO_JOIN* arrives, the protocol upcalls the application asking for the approval of the new site (and passes as parameter the *auth-data*). If S_j is accepted, a join event *Join(S_j)* is generated for further processing. Later, when the new view is installed, S_j gets the group state (line 26 of Algorithm 6). The site has to wait for $(f+1)$ identical copies to know it has received the correct state.

6.3.6 Site removal and failure detection

The *failure detector* module determines if other sites have failed, and produces events *Remv(S_j)* which are then handled by the membership protocol. Although the design of a Byzantine failure detector is not the subject of this chapter, we will provide some insights about its implementation through the rest of this section.

Byzantine failure detectors have to detect different faulty behaviors in the system, ranging from crash to malicious faults. Even for crash faults, this task is impossible to accomplish in asynchronous systems since in general it is not possible to differentiate the situation where a site is very slow or crashed. In our case, we have the same problem since the membership protocol is executed in the asynchronous payload part of the system. Detectors for malicious faults are harder to develop because they have

to be designed, at least in part, in a way that depends on the protocols being used by the sites being monitored (Doudou *et al.* , 2002). They have to know and understand the expected behaviors of these protocols, otherwise, some types of attacks are not detected. For this reason, they will have to look during the execution of the protocols (some of these ideas are borrowed from (Malkhi & Reiter, 1997c; Doudou *et al.* , 2002; Kihlstrom *et al.* , 2003)) for the following activities:

- Determine if a site completely stops interacting, either because it crashed or because it is malicious.
- Find out if a non-crashed site is silent for some part of a protocol or application execution, i.e., if it does not send some expected messages but it continues to send others. For example, in our particular case, a site that does not send a message *(INFO, myid, Ev(S_j), *)* after receiving $f + 1$ INFO messages with the same event.
- Determine if a site sends incorrectly formed or out-of-order messages. For example, a site replays some previously sent message.
- Establish if a site sends unexpected messages or messages with incorrect content. For instance, a site that sends a *(INFO, id, Ev(S_j), *)* with *id* different than *myid*.
- Find out if a site is being externally attacked and intruded. The output of an Intrusion Detection System could be used as an indication of the intrusion.

In practice, one should expect that it will be impossible to build a perfect failure detector, i.e., a detector that is able to eventually catch all failures and never makes mistakes. This is especially true for Byzantine failures. Therefore, our membership protocol, like all other protocols of this kind, will potentially make bad decisions due to incorrect information given by the failure detector. For example, these bad decisions might be: a) non-removal of a malicious site because its evil actions are not detected during a period of time, or b) the exclusion of a delayed but good site. In the first case, the system will continue to function as expected as long as the resilience threshold is not reached, i.e., while $f \leq \lfloor \frac{|V^n|-1}{3} \rfloor$. In the second case, an exclusion only occurs if

$f + 1$ sites detect a failure, which means that a number of mistakes are automatically tolerated by the protocol. Nevertheless, if a correct site is removed, it can always re-enter the group later.

Notice that the membership protocol does not need the failure detector to make progress, on the contrary, e.g., of (Reiter, 1996b). If the failure detector does not detect a failure, for instance because it is not able to detect a given class of failures, the membership will not remove the corresponding site from the view. However, it will still be able to behave according to its specification, i.e., to go on installing new views.

6.3.7 Membership protocol and FLP

The membership protocol is based on an agreement protocol –VCA– and runs mostly in the payload system, which is asynchronous. Therefore it makes sense to wonder about the relation between this protocol and the FLP impossibility result.

The reader should refer to a similar discussion about the consensus protocol (see Section 5.2.4). Here the reasoning is basically the same, since the core of VCA is similar to the core of the consensus protocol (respectively Algorithms 7 and 4). Both VCA and the consensus protocol go on proposing and getting the results of successive TBAs until a given condition is satisfied. Therefore, the communication among the sites is ordered and FLP does not apply for the same reasons as in Section 5.2.4.

The VCA protocol also needs a week, local synchrony assumption for termination to be guaranteed: *eventually there will be a round when at least* $2f + 1$ *sites manage to call* TTCB_propose *before one of the* tstart *deadlines*.

6.4 Performance evaluation

The performance of the membership service was evaluated using the COTS-based TTCB (Chapter 3).The evaluation of the membership performance was done on a sys-

tem with six PCs, each with a 450 Mhz Pentium III processor and 192 Mbytes RAM. The networks were two 100 Mpbs Fast-Ethernet LANs. The code was implemented in C and compiled with *gcc*. The MD5 hash function was used both to calculate the digests and the MACs. The communication was done with IP multicast. Since the maximum number of PCs was limited to six, it was necessary to set $f = 1$. The value used for T_{tstart} was 14 milliseconds, a value slightly over the maximum time TBA takes to run in the current implementation, 13 milliseconds. Each measurement was repeated at least 1000 times.

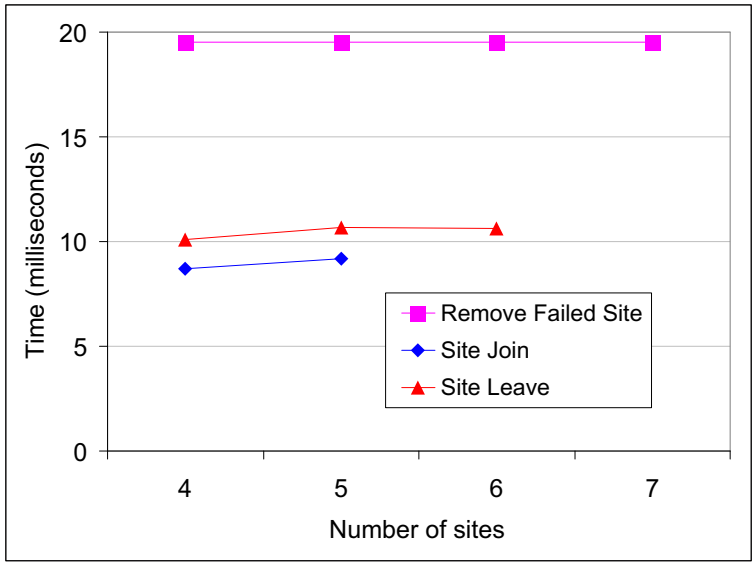

Figure 6.2: Average times to install a new view with the operations remove, join and leave.

The results of the experiments are presented in Figure 6.2 and in Table 6.1. There were *three experiments*. The first experiment quantified the time to *remove a failed site* that stopped interacting, either because it crashed or was corrupted. The failure detection was simulated with a multicast of a short message. The second experiment assessed the time for a site to *join*. The site multicasted a REQ_TO_JOIN message to all others, waited for them to install a new view and to get the state transfer from $f + 1 = 2$ sites.

No authorization scheme was used. The third experiment evaluated the time for a site to *leave* the group. The site multicasted a LEAVE message to the group and measured the time until the new view was installed without itself. For the first experiment, the time presented in the figure is the average of the times measured by all correct processes. For the second and third experiments, the times were assessed by the sites that joined or left.

	Remove		Join		Leave	
N.sites	Average	Stddev	Average	Stddev	Average	Stddev
4	19522	3847	8721	2153	10074	1817
5	19522	3875	9164	1418	10655	1882
6	19526	3846			10602	2039
7	19532	3848				

Table 6.1: Average and standard deviation times to remove, join, and leave (μs).

The protocol spends most of the time in the following operations: calculating the MACs for the INFO messages; exchanging these messages; and executing the TTCB TBA. The time to execute TBA is the most important since no public-key cryptography is utilized. As a consequence of this, the performance does not change much with the increase of the number of sites (see figure). Most experiments required a single execution of the TBA service. In approximately 3.4% of the experiments some sites tried to propose a value to the TBA after *tstart*, and in a few of these cases two TBAs had to be run. The join and leave experiments have execution times that are similar. The remove time is almost the double of the others. The reason for the different behavior is that the failed site does not engage in the VCA protocol, and consequently in the TBA(s). A TBA starts to execute when either all sites submit their values or by *tstart* (if a subset of them do not propose). In the join and leave experiments all sites proposed, so TBA was initiated (and terminated) earlier. In the remove case, the TBA had always to wait for *tstart* to begin. If the number of failed sites was increased, the expected

execution time would remain approximately the same since the TBA would also start by *tstart*.

Currently, we are aware of only three other implementations of membership services for systems that might experience Byzantine faults, which are Rampart (Reiter, 1996b), ITUA (Ramasamy *et al.* , 2002) and SecureRing (Kihlstrom *et al.* , 2001). The experimental settings used in the evaluations of these services are completely different from ours. Consequently, it is quite difficult to make a concrete comparison among the various performance results. Nevertheless, we will provide their values so that a qualitative assessment can be made. The Rampart protocol execution times range from 210 to 250 milliseconds for 4 to 6 processes running on a system of Sun SPARCstation 10 spanning several networks, and using RSA public-key cryptography with 512-bit moduli. ITUA obtained values from 400 to 500 milliseconds for detecting a crash and installing a new view on a system with 1GHz Pentium III computers with 256MB RAM, connected by a 100 Mbps Ethernet network. SecureRing obtained values from approximately 400 to 950 milliseconds to change the view with 4 to 8 processors, using RSA with 768 bit moduli (lower values were obtained with 300 and 512 bit moduli). The tests were made in 168 MHz Sun UltraSparc 2 workstations with Solaris, connected by a 100 Mbps Ethernet. Even taking into consideration the different characteristics of the various systems (we used faster machines than Rampart and SecureRing, but slower than ITUA), our results are in the order of 10 to 20 milliseconds, which seems to indicate that the proposed protocol performs better.

6.5 Related work

We are aware of only three intrusion-tolerant membership services: Rampart (Reiter, 1996b), SecureRing (Kihlstrom *et al.* , 2001) and SecureGroup (Moser *et al.* , 2000; Moser & Melliar-Smith, 1999). Project ITUA implemented an slightly improved version of the Rampart's service (Ramasamy *et al.* , 2002). These systems are described in Section 2.4.

The Rampart membership uses a three-phase commit style protocol. Processes in the group send failure suspicions to a leader that tries to change the membership when a majority is received. The sender uses digital signatures (with public-key cryptography) to prove that it received the suspicions. The protocol relies on the failure detector to remove a failed leader and make progress, e.g., to eventually install a new view.

SecureRing is designed for LANs and relies on a logical ring imposed on the communication medium that controls the multicasting of messages. The membership protocol reconfigures the system when one or more hosts exhibit detectable Byzantine failures, which are detected by a Byzantine failure detector (Kihlstrom *et al.* , 2003). SecureGroup is also designed for LANs (Moser *et al.* , 2000; Moser & Melliar-Smith, 1999). The membership protocol is simpler than the others are because it is implemented on the top of a Byzantine-resilient atomic multicast protocol. Both SecureRing and SecureGroup use digital signatures to protect some messages.

6.6 Summary

This chapter presents a novel intrusion-tolerant membership service based on our hybrid fault model and the TTCB. The protocol seems to be around 10 to 20 times faster than similar protocols in the literature. The test conditions were not the same, as also the system model. However, ITUA used faster PCs and weak RSA cryptography (512-bit moduli) and even though the membership presented here seems to be much faster. The performance of these other systems seems to degrade exponentially with the number of processes involved, which is consistent with the fact that the number of public-key operations increases greatly with the number of processes involved. The performance of the membership presented in the chapter seems to degrade very little with the number of hosts, but it was tested with only 6 hosts.

The membership service makes decisions in a distributed way, therefore it does not rely on a leader. This is a considerable advantage for two reasons. The first is that detecting a malicious leader is an operation that can cost some time, therefore a failed

leader delays the protocol. The second is that an attacker can try to delay the service by postponing the communication and creating false failure suspicions of successive leaders.

Note

The content of this chapter was partially reported in (Correia *et al.* , 2003c).

7

View-synchronous atomic multicast

This chapter presents the final protocol for an elementary intrusion-tolerant group communication system (GCS) based on the MAFTIA middleware architecture and the hybrid fault model substantiated by the TTCB.

The GCS supports the abstraction of groups of hosts. Therefore, the system is an instance of the MAFTIA middleware site-level (see Figure 2.5) [1]. This chapter proposes an *Atomic Multicast* protocol (BAM-VS), which is part of the Communication Support Services module (CS). The Site Membership module (SM) is implemented by the membership service described in the previous section and the Site Failure Detector (SF) is discussed in Section 6.3.6. Both the membership and the multicast protocol use the secure channels introduced in Section 5.1. These channels can be implemented either in the Multipoint Network module (MN) or in the CS module.

The GCS proposed here is based on the primary partition model, i.e., in case there is a network partition that divides the group in two or more subgroups only one is allowed to make progress. The membership service and BAM-VS, respectively, change the view or deliver messages when they have contributions from $2f + 1$ sites, with $f \leq \lfloor \frac{|V^n|-1}{3} \rfloor$. No two partitions of a group can have $2f + 1$ hosts so at most one subgroup can make progress when a partition occurs. This issue was discussed in Section 6.1.

The BAM-VS protocol provides a view-synchronous semantics. This property says, informally, that all correct group members deliver the same messages in the same

[1] Throughout the chapter the words host and site are used interchangeably.

view (Birman & Joseph, 1987b; Birman & Joseph, 1987a). Group communication is usually performed using a set of reliable multicast primitives with different order properties. BAM-VS orders messages in total order, i.e., all correct hosts deliver the messages in the same order.

The chapter brings together most of the concepts and protocols presented in the rest of the thesis. The MAFTIA middleware architecture was presented in Section 2.5, the TTCB in Chapter 3 and the membership service in Chapter 6. The view-synchronous Atomic Multicast derives from the reliable multicast protocol in Chapter 4, which was independent of the membership service.

7.1 View-synchronous atomic multicast

There are several similar but different definitions of view synchrony in the literature. The definition used here is given by the following property (inspired in (Chockler *et al.* , 2001)) [2]:

- *View Synchrony:* If two correct sites install views V^n and V^{n+1} then both sites deliver the same messages in view V^n.

A reliable multicast protocol can be formally defined in terms of three properties:

- *Validity:* If a correct site multicasts a message M, then some correct site in *group(M)* eventually delivers M.
- *Agreement:* If a correct site delivers a message M, then all correct sites in *group(M)* eventually deliver M.

[2]The reader should keep in mind the way in which the words *receive* and *deliver* are being used. A protocol layer *receives* a message from lower layers (e.g., from the network) and *delivers* something to higher layers (e.g., to the application).

- *Integrity:* For any message M, every correct site p delivers M at most once and only if p is in *group(M)*, and if *sender(M)* is correct then M was previously multicast by *sender(M)*.

This definition is the same as the one in Chapter 4 except for the meaning of the predicate *group(M)*. This predicate indicates the members of the group in the view in which the message is eventually delivered (since the message does not have to be delivered in the view to which it was initially multicasted). *sender(M)* is the sender of the message.

The *Byzantine Atomic Multicast* protocol is defined in terms of the view synchrony property, the three reliable multicast properties and an order property:

- *Total order:* If two correct sites deliver two messages M_1 and M_2 then both sites deliver the two messages in the same order.

Some view synchrony definitions in the literature impose that messages have to be delivered in the view in which they are sent, e.g., the Horus strong virtual synchrony (Friedman & van Renesse, 1996). Others allow messages to be delivered in subsequent views, e.g., Rampart (Reiter, 1994). Our definition falls in the second category since the first seems to be incompatible with the membership service considered. Suppose that a correct sender is still not aware of an ongoing view change and multicasts a message to the group. It is possible that $2f + 1$ other sites decide to change the view before they receive that message, therefore the message is not delivered in the view in which it was initially sent.

7.2 The protocol

This section describes the view-synchronous atomic multicast protocol – BAM-VS. The protocol is based on the membership service in the previous chapter. It also assumes that only $f \leq \lfloor \frac{|V^n|-1}{3} \rfloor$ sites can fail in a given view V^n. The protocol considers

the same system model as in Chapters 5 and 6, including the secure channels described in Section 5.1.

Pseudo-code for the basic protocol can be found in Algorithm 8. A message of the protocol is uniquely identified by *mid = (sender-eid,tstart)*. This requires that the lists with site identifiers *elist* have to be in a canonical form: the first *eid* is the sender's, and the others are in ascending order. A brief justification for the uniqueness of *mid*: the sender uses the TTCB TBA service to give all sites a hash of the message; an execution of the TBA is uniquely identified by *(elist, tstart, decision)* (Section 3.2.3) and *decision* is hard-coded in the protocol (TBA_RMULTICAST); the *sender-eid* identifies a single *elist* in canonical form in a view; therefore it is not possible to send another message in the same view with the same id.

The protocol in Algorithm 8 is similar to BRM-M (Chapter 4). The sender gives the TTCB TBA a hash of the message (line 7) and then multicasts the message (line 9). The recipients check if the message received *M* corresponds to the current view (line 14), if *elist* is in the canonical form (line 14) , and if the sender gave the TBA the hash of *M* (lines 21-22). Then, if *M* corresponds to the hash given by the TBA (line 24), the recipient resents it to all sites which did not provide the correct hash to the TBA (line 25). This part of the protocol in Algorithm 7.1 does not deliver the messages, but puts them in a bag *bag-data-msgs* instead (lines 11 and 26).

A message is said to be *stable* if it can be delivered (Moser *et al.* , 1994). The protocol is similar to BRM-M so we can assume for now that it satisfies the properties of Validity, Agreement and Integrity above: if a correct site receives a message all correct sites will eventually receive it. However, to satisfy the view synchrony property the message can be delivered only if all sites agree to deliver it in the current view. Additionally, BAM-VS requires that all sites deliver the messages in the same order. How are these properties satisfied?

The solution is based on modified versions of the membership (Algorithm 9) and VCA protocols (VCMDA, Algorithm 10). The original membership protocol handles three types of events: *Join*, *Remv* and *Leave*. BAM-VS generates a new type of event

Algorithm 8 View-synchronous atomic multicast.

```
1  INITIALIZATION:
2  hash⟨mid⟩ ←⊥;                                          {hash of the message sent}

3  WHEN BAM-VS-multicast(data) is called DO                                  {SENDER}
4  elist ←all sites in current view in canonical form;
5  repeat
6     M ←(elist, TTCB_getTimestamp() + T_1, data);
7     outp ←TTCB_propose(M.elist, M.tstart, TBA_RMULTICAST, Hash(M));
8  until (outp.error ≠ TSTART_EXPIRED);
9  multicast M to all sites in current view;
10 hash⟨mid⟩ ←Hash(M);
11 bag-data-msgs ←bag-data-msgs ∪ {M};          {bag with the messages to be delivered}
12 generate event Datamsg(mid); terminate;      {terminates protocol for message ⟨mid⟩}

13 WHEN M⟨mid⟩ received DO                                               {RECIPIENTS}
14 if (M.elist does not correspond to current view) or (M.elist not in canonical form)
   or (my-eid ∉ M.elist) then
15    return;
16 if (hash⟨mid⟩ = ⊥) then
17    outp ←TTCB_propose(M.elist, M.tstart, TBA_RMULTICAST, Hash(M));
18    repeat
19       outd ←TTCB_decide(outp.tag);
20    until (outd.error ≠ TBA_RUNNING);
21    if (outd.error = DID_NOT_PROPOSE) then
22       return;
23    hash⟨mid⟩ ←outd.value;
24 if (Hash(M) = hash⟨mid⟩) then
25    multicast M to sites in current view except sites in the mask outd.proposed-ok;
26    bag-data-msgs ←bag-data-msgs ∪ {M};
27    generate event Datamsg(mid); terminate;
```

Algorithm 9 Membership and message delivery protocol.

```
1  INITIALIZATION:
2  bag-info ←⊥;                                              {bag with INFO messages}
3  bag-decisions ←⊥; {bag with changes the site wants to be done to the view, and messages
   to deliver}
4  valid-tstart-send ←⊥;                {valid tstart to send in INFO messages in this view}
5  state ←NORMAL;                                                    {protocol state}

6  WHEN Ev(S_j) DO
7  if (I did not multicast (INFO, myid, Ev(S_j), *) in this view) then
8     if (valid-tstart-send = ⊥) then
9        valid-tstart-send ←next-valid-tstart();
10    multicast (INFO, myid, Ev(S_j), valid-tstart-send);

11 WHEN M = (INFO, sender-id, Ev(S_j), valid-tstart) received DO
12 bag-info ←bag-info ∪ M;
13 if (count(Ev(S_j), bag-info) ≥ f+1) and (I did not multicast (INFO, myid, Ev(S_j), *) in this
   view) then
14    if (valid-tstart-send = ⊥) then
15       valid-tstart-send ←next-valid-tstart();
16    multicast (INFO, myid, Ev(S_j), valid-tstart-send);
17 if (count(Ev(S_j), bag-info) = 2f+1) then
18    bag-decisions ←bag-decisions ∪ Ev(S_j);
19    if ((Ev ≠ Datamsgs) or (count(Datamsgs, bag-decisions) = DLVR_WATERMARK))
      and (state = NORMAL) then
20       state ←AGREEMENT;
21       execute vcmda( smallest-tstart(bag-info) );

22 WHEN bag-decisions = vcmda(tstart) returned DO
23 deliver messages corresponding to Datamsg events in bag-decisions ordered by M.tstart;
   remove them from bag-data-msgs and the corresponding events from bag-decisions;
24 if (there are view change events in bag-decisions) then                {install new view}
25    add and remove sites in view change events in bag-decisions from view;
26    view-number ←view-number + 1;
27    bag-info ←⊥; bag-decisions ←⊥; valid-tstart-send ←⊥; bag-data-msgs ←⊥;
28    send state to new members;
29 state ←NORMAL;
```

Algorithm 10 View Change and Message Delivery Agreement protocol (VCMDA).

```
1  FUNCTION vcmda(tstart)
2  hash-v ←⊥;                                              {hash of the value decided}
3  bag-msgs-decision ←⊥;                                   {bag for messages received with decisions}
4  elist ←list with eid's of sites in V^{view−number} in ascending order;
5  if (tstart < last-tstart) then
6     tstart ←last-tstart;
7  data-msgs-deadline ←⊥;

8  loop
9     repeat
10       bag-decisions-prop ←bag-decisions;
11       if (data-msgs-deadline ≠ ⊥) then
12          remove from bag-decisions-prop all events Datamsg with tstart > data-msgs-
            deadline;
13       outp ←TTCB_propose(elist, tstart, TBA_MAJORITY, Hash(bag-decisions-prop));
14       repeat
15          outd ←TTCB_decide(outp.tag);
16       until (outd.error ≠ TBA_RUNNING);
17       if (data-msgs-deadline = ⊥) and (at least 2f + 1 sites proposed any value) then
18          data-msgs-deadline ←tstart;
19       tstart ←tstart + T_{tstart};
20    until (at least 2f + 1 sites proposed the value decided);
21    last-tstart ←tstart;
22    if (outd.value = Hash(bag-decisions-prop)) then
23       multicast (DECISIONS, myid, bag-decisions-prop) to sites not in outd.proposed-ok
         and not being removed;
24       return bag-decisions-prop;                        {terminates the algorithm}
25    else
26       hash-v ←outd.value;

27    WHEN M=(DECISIONS, *, *) received DO                 {decisions message for this VCMDA}
28    bag-msgs-decision ←bag-msgs-decision ∪ {M};

29    WHEN (hash-v ≠ ⊥) and (∃_{M∈bag−msgs−decision} : Hash(M.bag-decisions) = hash-v) DO
30    return M.bag-decisions;                             {terminates the algorithm}
```

when it is ready to deliver a message identified by *mid*: *Datamsg(mid)* (Algorithm 8, lines 12 and 27). This event is handled by Algorithm 9 basically in the same way as the view change events: an INFO message is sent (lines 6-10) and when there are $2f + 1$ INFO messages it is inserted in a bag and VCMDA can be started (lines 11-21). A difference in relation to the view change events is that a single *Datamsg* does not start VCMDA. This protocol is started when a certain condition is satisfied. The code shows one possibility: VCMDA starts when DLVR_WATERMARK data messages are ready to be delivered (line 19). Another condition would be to start VCMDA if a certain time passed from the last delivery. The code can be easily modified to test this condition, or a combination of the two.

VCMDA is used with two purposes. The first is basically the same as VCA, i.e., for all sites to agree on the view changes to do on the current view. However, the view synchrony property states essentially that all correct sites deliver the same messages in a view, therefore VCMDA, together with the view changes, has also to agree on the non-delivered messages still to be delivered in the view. Therefore, VCMDA makes agreement on 'decisions': view changes and message deliveries (this is the reason why several variables were renamed, e.g., VCA's *bag-changes* is called *bag-decisions* in VCMDA).

The second purpose of VCMDA is to agree on messages to be delivered when there are *no view changes*. The objective is to satisfy the total order property. In each execution of VCMDA all sites agree to deliver the same set of messages. Then these messages are ordered according to their *tstart* and delivered (Algorithm 9, lines 22-23). The code executed is precisely the same as before except for lines 25-28 in Algorithm 9.

VCMDA has still another difference in relation to VCA. The number of view change events that can be generated is limited[3], but the same is not true for data messages. The constant appearance of new events could prevent VCMDA from terminating since *bag-decisions* would go on changing indefinitely. The solution for this

[3]The sites that can be allowed to join have to be know in order to be authenticated, therefore their number is limited. The number of sites that can leave or be removed is the same as the number of members. The maximum number of view change events is the addition of these two numbers.

problem is the definition of a deadline *data-msgs-deadline* for each VCMDA execution (Algorithm 10, lines 7, 12, 17-18). Any message with *tstart* greater than *data-msgs-deadline* is not considered for that execution of VCMDA (line 12). *data-msgs-deadline* is the first *tstart* in which at least $2f + 1$ sites propose any value for a TBA (lines 17-18).

The correctness proof of the protocol can be found in Appendix A.

7.3 Performance evaluation

The performance of the BAM-VS protocol was evaluated using the COTS-based TTCB, in the same setting as the membership service in the previous chapter (Section 6.4). The evaluation consisted in three sets of experiments. Every experiment used at least 1000 messages. Every set of experiments measured the average delivery time (or latency) and the sustainable throughput of the protocol. The experiments were based on a prototype of the protocol that did not send messages during the execution of VCMDA. Therefore, the experiments were performed in rounds. Each round started with sites multicasting and receiving messages using the Algorithm 8. Then the sites executed VCMDA, delivered the messages and started another round.

The first set of experiments evaluated the performance of BAM-VS with 4 sites, a single sender, no failed sites and delivery watermarks (DLVR_WATERMARK) from 1 to 25 messages (Figure 7.1). The throughput of the protocol increased considerably until about 10 messages, then started to improve more slowly. If the delivery watermark is low then the time taken by the VCMDA protocol to agree on the messages to deliver is similar to the time taken to send the messages, i.e., the time spent in the first part of the round. If the delivery watermark is high the time used by the VCMDA protocol can be negligible in face of the time taken sending messages. The average latency, on the contrary, increased steadily with the delivery watermark. This was expected since the messages are delivered when VCMDA terminates, and the higher the watermark, the higher the time the first messages sent wait for VCMDA to terminate.

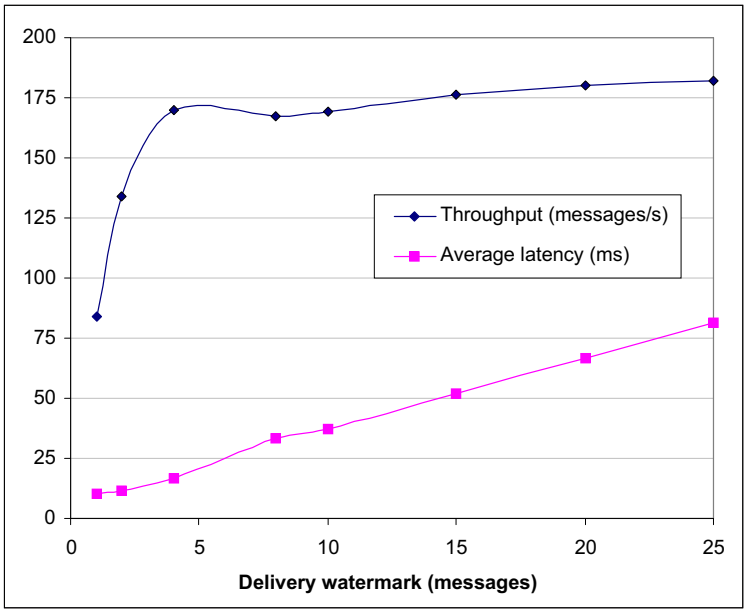

Figure 7.1: BAM-VS performance with different delivery watermarks (4 sites, one sender, 100 bytes messages).

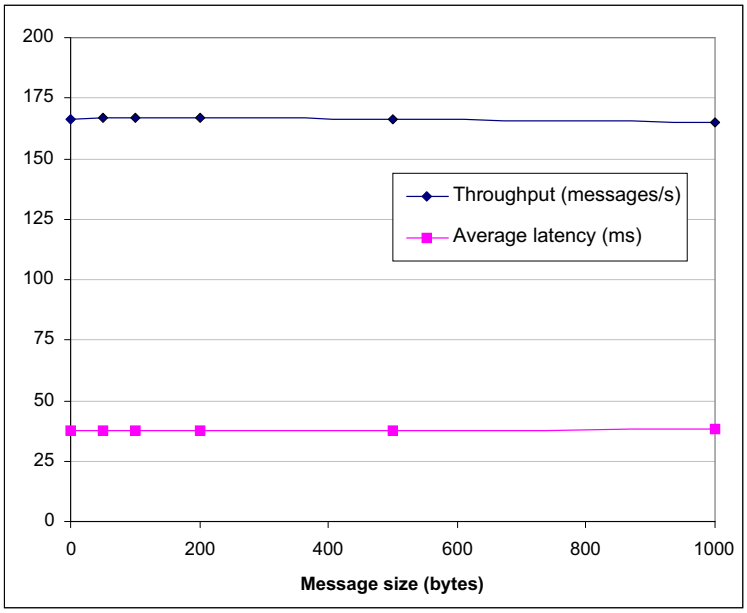

Figure 7.2: BAM-VS performance with different message sizes (4 sites, one sender, watermark of 10 messages).

The second set of experiments measured the performance of the protocol with the variation of the message size (Figure 7.2; sizes do not include the header). The value selected for the watermark was 10, which the first set of experiments indicated to be a good compromise in terms of throughput and latency. The number of sites was 4, there was a single sender and no failed sites. The conclusion from the figure is that this variation of message size does not affect significatively either the sustainable throughput or the latency.

The throughput of the protocol is limited by the capacity of the TTCB to execute TBAs. The TTCB is a real-time subsystem so this capacity is limited. In the TTCB configuration used in the experiments, every local TTCB broadcasts a packet to all other local TTCBs every 8 ms (see the TBA protocol in Section 3.2.3). This packet carries data of at most 3 TBAs so a process in a machine theoretically can successfully propose for at most 375 TBAs per second. The TTCB has also a limited capacity for storing information about TBAs but this number is high enough not to interfere with the experiments. The throughputs obtained show that the protocol can still be tuned to use the TTCB capacity more efficiently. The prototype purposely loses some time in order to avoid the loss of messages in the network, since a retransmission layer was not implemented.

The third set of experiments compared the performance of the protocol with 4 to 6 hosts in three different conditions (Figures 7.3 and 7.4). The first condition is the same as before, i.e., no failed sites, a single sender and a watermark of 10 messages. In the second condition all sites send messages and the watermark is twice the number of sites (i.e., 8, 10 and 12). In the third condition there is a single sender but one site is silent, i.e., it does not participate in the protocol because it is crashed (the watermark is 10). The message size was 100 bytes for all experiments.

The experiment allows several conclusions. One is that the number of sites does not affect considerably the performance of the protocol. This was already concluded in the previous chapter for the membership service. The case in which all sites send might seem to be an exception but the reason for the increase in the throughput and latency is the different watermark used. The second conclusion is that the protocol has

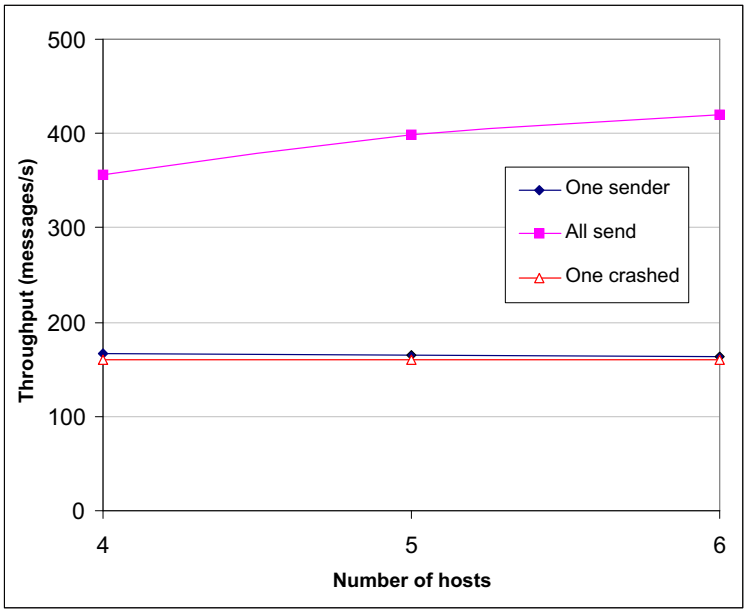

Figure 7.3: BAM-VS throughput with one sender, all sites sending and one silent site (4 sites, watermark of 10 messages, 100 bytes messages).

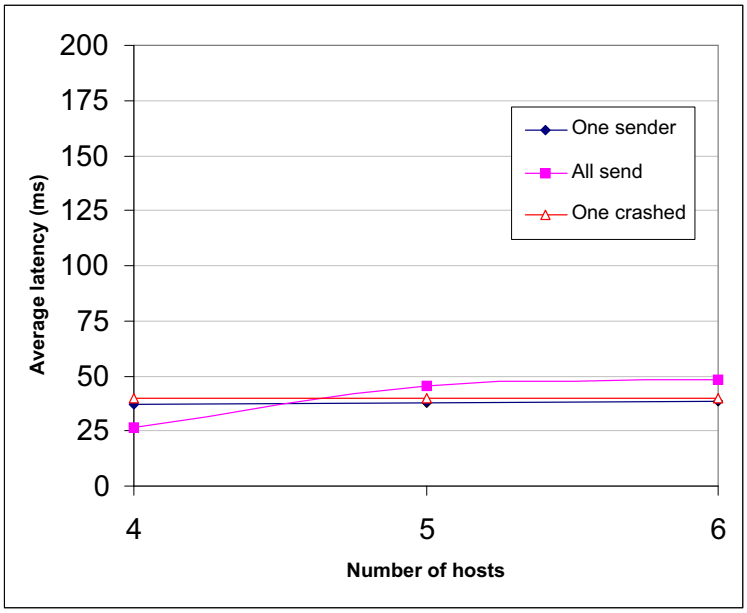

Figure 7.4: BAM-VS average latency with one sender, all sites sending and one silent site (4 sites, watermark of 10 messages, 100 bytes messages).

a higher throughput if all sites send. The third conclusion is that a crashed site does not affect the performance of the protocol. The throughput is slightly better since the silent site does not multicast either data of INFO messages. The latency is 2 to 3 ms worse because the TBA takes longer to run if not all sites involved propose before *tstart*.

There are some numbers available for the performance of intrusion-tolerant view-synchronous atomic multicast protocols in the literature: Rampart (Reiter, 1994) and its more recent implementation by project ITUA (Ramasamy *et al.* , 2002). These numbers have to be compared with the performance of BAM-VS with caution, since the test conditions were quite different. The Rampart test conditions were probably worse than ours but the ITUA were better, as already noticed in Section 6.4. With 4 to 6 processes, the throughput of Rampart was approximately 18 to 14 messages/s with one sender, and 23 to 17 messages/s with all processes sending (300 bits RSA moduli). These numbers are about 10 times worse than BAM-VS. The times for ITUA are presented in terms of time to deliver 10 messages. This is approximately equivalent to the average latency of our protocol with a watermark of 20 messages. ITUA times ranged from 700 to 1100 ms with 4 to 6 processes (1024 bits RSA moduli). BAM-VS had a time of approximately 66,5 ms (the latency does not degrade visibly with the number of sites). The first conclusion from these numbers is that BAM-VS performs better than similar protocols, both in terms of latency and throughput. The second it that BAM-VS also does not degrade its performance with the number of sites involved, although only a limited number of machines was available.

7.4 Related work

The concept of virtual synchrony was introduced by Birman and Joseph (Birman & Joseph, 1987b; Birman & Joseph, 1987a). The concept is based on the idea of a synchronous environment where all messages and events (like view changes) are delivered in order. However, ordering all messages and events is costly. The concept of virtual synchrony tries to preserve the illusion of synchrony, but communication prim-

itives with weaker ordering (FIFO, causal, no order) are provided for applications that are insensitive to that aspect. A discussion of related work in this area can be found in Section 2.4. An interesting survey on total order multicast can be found in (Défago *et al.* , 2000). Most work on this area assumes only crash faults.

There are only three Byzantine-resilient group communication systems that we are aware of which provide the view-synchronous semantics: Rampart, SecureRing and SecureGroup.

The Rampart toolkit provides primitives for reliable and atomic multicast (Reiter, 1994). Processes communicate through reliable channels and use public-key cryptography to sign some of the messages. The atomic multicast is based on the reliable multicast. When a process wants to atomically multicast a message it reliably multicasts it. Then, a designated member of the group, called sequencer, defines a message order and reliably multicasts a message with the order to all members. When a member receives this message it delivers the messages in the given order. A malicious sequencer can refuse to give the order of delivery of one or more messages. That situation has to be detected using a timeout and the malicious sequencer removed from the group. This approach has problems that were already discussed: slow communication, either due to an attack or to accidental faults, can cause correct sequencers to be removed, and if sequencers are repeatedly removed the system does not make progress. An attacker can also force the removal of correct processes to eventually cause the violation of the assumption than no more than f out of $3f + 1$ processes fail.

SecureRing uses a logical ring to impose total order (Kihlstrom *et al.* , 2001). Processes pass a token to the next one in the ring. A total order is intrinsic to the protocol, since only the process with the token can multicast. An unreliable failure detector is used to detect if a malicious process does not pass the token or corrupts its information in some way.

SecureGroup uses a combination of positive and negative acknowledgments to order messages in causal order (Moser *et al.* , 2000; Moser & Melliar-Smith, 1999). A total order is built on the top of this causal order. On the contrary to Rampart and

SecureRing, SecureGroup does not provide 'multicast' but 'broadcast' primitives. All communication in a given domain is ordered and the membership protocol runs on the top of this totally ordered communication.

7.5 Summary

This chapter presents a novel intrusion-tolerant atomic multicast protocol, which was the missing component for the definition of an intrusion-tolerant group communication system. This protocol performs arguably better than similar protocols in the literature, and its throughput does not degrade significatively with the number of sites involved. Additionally, the protocol does not rely on a leader to order messages so it is not necessary to detect the failure of processes for the protocol to make progress. The decisions about the delivery of messages are taken cooperatively by $2f + 1$ sites so progress is ensured as long as the assumption than no more than f sites fail is satisfied.

The GCS presented in Chapters 6 and 7 resumes to the site level in the MAFTIA middleware architecture (see Section 2.5). Groups of participants are mapped into groups of sites, therefore the participant level membership and communication modules are simple to implement. For instance, for participant communication, if a participant atomically multicasts to a group of participants, the message is passed to BAM-VS that delivers the message in total order to all sites; then the site level at each machine gives the message to the participant level, which delivers the messages to all participants of the group in the machine. The participant level membership module is equally simple and was presented in a MAFTIA report (Armstrong *et al.* , 2002, Section 5.4.2.9).

Note

The content of this chapter was partially reported in (Correia *et al.* , 2003c). Preliminary ideas on the design of an intrusion-tolerant group communication system were published in (Correia *et al.* , 2001a).

8

Conclusion

This thesis presents a novel approach for designing and implementing intrusion-tolerant distributed systems. This approach is based on an architectural-hybrid fault and synchrony model. In the context of our system, this model considers that most components in a system can fail arbitrarily, even maliciously, but that there is set of components that are fail-silent and thus secure. Likewise, most of the system can have uncertain synchronism, even be asynchronous, but there is a set of components that are synchronous. In the thesis, this set of components is materialized by a distributed subsystem the TTCB – which is built to remain secure. The TTCB is a synchronous, or real-time, distributed component, with local parts in hosts and its own private secure channel or network.

The thesis presents the TTCB model and the design of a particular implementation of this model, the 'COTS-based' TTCB. The thesis also presents the design of several intrusion-tolerant protocols supported by the TTCB. The novelty is not in the protocols themselves, similar to classical distributed systems protocols, but on their implementation using the TTCB to perform critical but 'small' steps of the protocols.

A reliable multicast protocol provides a first insight into how these protocols can be designed. The protocol is efficient and does not impose limits on the number of faulty processes, although it has been proved that in asynchronous systems (with no TTCB) less than one third of the processes can fail for reliable multicast to be possible.

A consensus protocol shows a different way to use the TTCB TBA service to implement intrusion-tolerant protocols. This protocol also shows how a protocol based on

the TTCB relates to the FLP impossibility result. The TTCB is synchronous therefore the protocol is not constrained by that result. However, to ensure termination we still need a synchrony assumption about the payload system. This assumption can be said to be 'weak' since it is about the behavior of correct hosts, not about the network or about corrupt hosts. The protocol has low time and message complexities.

Finally, the approach is used to design a system with practical interest: an intrusion-tolerant group communication system with a membership service and an atomic multicast primitive. This system provides interesting benefits in relation to similar systems in the literature. Firstly, it seems to perform considerably better than most systems in the literature, both in terms of throughput and latency, since it does not use public-key cryptography in runtime. Secondly, for the same reason its performance degrades very slowly with the number of sites involved. Thirdly, sites make decisions to install a new view or deliver messages cooperatively, in a distributed way. Therefore there is no leader whose failure has to be detected using timeouts. This makes the protocols resilient to a class of time attacks that aims to delay or break protocols making successive leaders be suspected or detected failed. These benefits can give an idea of the relevance of the TTCB to build practical intrusion-tolerant systems.

Present and future work is being and will be pursuit in several areas.

The motivation for a COTS-based TTCB implementation was the facility of installing and testing it by other research groups. However, for practical applications the TTCB probably needs a higher assumption coverage, therefore a hardware implementation will be done in the near future. The local TTCB will run in an appliance board (like a PC104 board) with a very controlled interface with the rest of the machine. A third implementation of the TTCB is also envisaged, this time for WANs. A TTCB distributed over a 'large' area poses some interesting challenges, like the reduced bandwidth, but may have interesting applications.

There are several other TTCB issues that will probably be explored in the future. The TBA service, designed in the context of this thesis, proved to be adequate to support a number of protocols. Is there a simpler and more efficient service that permits

the same? The current implementation of access control in the TTCB is very basic and a better solution would allow a better use of the its resources. How? An attacker can consume the resources of a local TTCB simply by calling the TBA fast enough. A partial solution for this problem could be to allow processes to reserve TTCB resources. The processes that obtained the resources before the intervention of the attacker would be able to use the TTCB under such an attack.

An area of current work is the design of an intrusion-tolerant service based on state machine replication. The protocol executed between the servers is BAM-VS. The performance of the service is especially important for practical purposes so the use of the TTCB-based protocols has much interest.

The protocol that was initially designed for the membership service has been proving to be useful for a considerable different range of applications. The thesis shows how it was successfully used to design an atomic multicast primitive. Another area of current work is the design of a distributed configuration system, which is also based on the same protocol. These applications give the intuition that the protocol can be used as the core of a Byzantine-resilient 'generic consensus service', which can be used to implement several distributed agreement protocols. The idea was introduced originally for crash-tolerant systems (Guerraoui & Schiper, 2001).

A final area of future work is motivated by the reliable multicast protocol, BRM-M. This protocol does not impose a limit on the number of failed processes, on the contrary to similar protocols on asynchronous systems that tolerate less than one third faulty processes. However, the rest of the protocols in the thesis did not manage to improve on the limit of failed processes for similar protocols on asynchronous systems. The area of future work is to study if the TTCB or another wormhole permits to improve on these limits.

A

Correctness proofs

This appendix provides proof sketches for all protocols presented in the thesis.

A.1 TTCB local authentication service protocol

This section is about the TTCB Local Authentication service protocol described in Section 3.2.1 and presented in Figure 3.2.

Theorem 1 *The Local Authentication service protocol is an authenticated key establishment protocol.*

Proof: The protocol is an authenticated key establishment protocol if it verifies properties SK1 through SK4.

SK1. The protocol verifies the Implicit Key Authentication property since the key is passed only in the first message and only the local TTCB has the private key that can decrypt this message, $\langle E_u(K_{et}, X_e)\rangle$.

SK2. The proof of the Key Confirmation property can be divided in two parts. The process knows that the local TTCB has the key because it gave it (first message) and receives back a confirmation that cannot be falsified since it is obtained with the local TTCB private key (second message). The local TTCB knows that the process has the key since it was the process that gave the key to the local TTCB (first message).

SK3. The Authentication property is also verified since the process gives the TTCB a message that only the TTCB can decrypt, $\langle E_u(K_{et}, X_e)\rangle$, and the TTCB gives back the decrypted challenge, X_e, showing that it was able to do the decryption.

SK4. The Trusted Against Known-Key Attacks property primarily depends on the key generation method and encryption algorithm. We assume that the process or the TTCB generate keys that verify that property. SK4 is guaranteed by the protocol since a key does not depend on a previous one. □

A.2 TTCB TBA service protocol

This section is about the TBA service protocol described in Section 3.2.3 and presented in Algorithm 1.

Theorem 2 *If the TBA service is implemented with the TBA service protocol and there are no local TTCB crashes then it verifies Termination, Integrity, Agreement and Validity.*

Proof: All local TTCBs broadcast the values proposed locally to all others. All receive and process the same values since there are no crashes and the protocol tolerates omissions in the network. The result is obtained applying a deterministic function to the values received. Therefore, all local TTCBs obtain the same result.

Termination. A correct process eventually calls *decide* to obtain the result of the TBA. Since all local TTCBs obtain the result, every correct process eventually decides a result.

Integrity. A process that calls decide more than once and obtains the result more than once, obtains always the same result. Therefore, every correct process decides at most one value.

Agreement. All local TTCBs obtain the same result so all correct processes decide the same.

Validity. The result is obtained in the local TTCBs applying the function *decision* to the values proposed. If a correct process decides, then it decides the value obtained by the local TTCBs. □

Theorem 3 *The TBA service protocol verifies Timeliness and T_{TBA} is given by:*

$$T_{TBA} = T_s + WCET_{send} + T_{send} + T_r + WCET_{receive} + \pi \quad \text{(A.1)}$$

Proof: $WCET_{send}$ and $WCET_{receive}$ are respectively the worst case execution times of the send and receive routines. Any value proposed after *tstart* is not accepted for the TBA (line 3). After the value is introduced in sendTable, the broadcast routine takes less than T_s to start to run, so the message with the value will not be broadcasted after $(tstart + T_s + WCET_{send})$ (lines 12-15). The message will take at most T_{send} to arrive to the local TTCBs and the receive routine may take at the most T_r to start to run. Then the message will take less than $WCET_{receive}$ to be received and processed (lines 18-23). The factor π takes in account the local TTCB clocks de-synchronization that leads to a different assessment of *tstart* in different local TTCBs. Adding all these maximum delays, we have the Formula A.1. Since there is a T_{TBA} the protocol has the property of Timeliness. □

A.3 TTCB TBA service crash-tolerant protocol

This section is about the crash-tolerant TBA service protocol and the Timely Reliable Broadcast protocol in Algorithm 2.

Lemma 1 *The Timely Reliable Broadcast protocol verifies the properties of Validity, Agreement and Integrity if there are no local TTCB crashes.*

Proof: Validity. A correct (non-crashed) local TTCB receives the messages that it R-broadcasts. If it R-broadcast $M(s, n)$ then it receives $M(s, n)$. After $M(s, n)$ it R-

broadcasts $M(s, n+1)$, $M(s, n+2)$, etc. After receiving $M(s, n)$, when it eventually receives $M(s, n_+)$ it R-delivers $M(s, n)$, $\forall_{n_+} : n_+ > n$.

Agreement. If a correct local TTCB R-delivers $M(s, n)$, then it received $M(s, n)$ and a message in one of the two conditions tested in line 13. Since we are not considering crashes and we assume that omissions in the network are masked, all local TTCBs receive those two messages and R-deliver $M(s, n)$.

Integrity. The property of Integrity means that no spurious messages are R-delivered. This is a consequence of AN2 and AN8. □

Lemma 2 *The Timely Reliable Broadcast protocol tolerates a single local TTCB crash in an interval of time (not assuming AN6).*

Proof: If a local TTCB crashes during the R-broadcast of $M(s, n+1)$, and no local TTCB receives it, no local TTCB R-delivers $M(s, n)$. If at least a single local TTCB receives $M(s, n+1)$ it R-delivers $M(s, n)$ (first condition on line 13) and, in the next message that it R-broadcasts, it will say to all the other local TTCBs that it received $M(s, n+1)$ (*higherseqVector[s]=n+1*) and they will also R-deliver $M(s, n)$ (second condition in line 13).

Now, let us show that the protocol does not tolerate two or more crashes. Suppose that the local TTCB s crashed and a single local TTCB received $M(s, n+1)$. Then, the local TTCB that received the message started to R-broadcast a message with *higherseqVector[s]=n+1*. If it also crashed during that R-broadcast, some local TTCBs could receive confirmations while other did not; this would cause some local TTCBs to R-deliver $M(s, n)$, while others would not. Therefore, the protocol tolerates a single local TTCB crash during a reliable broadcast. □

Theorem 4 *The Timely Reliable Broadcast protocol tolerates Bd local TTCB crashes (assuming AN6).*

Proof: Consider that the local TTCB s starts R-broadcasting $M(s, n+1)$ and crashes. The worst case happens when just Bd local TTCBs receive the message. We want to

prove that the protocol tolerates Bd crashes. Since the sender already crashed only $Bd-1$ local TTCBs can still crash. Therefore, of Bd local TTCBs that received the message at least one does not crash and sends *higherseqVector[s]=n+1* in the next message it R-broadcasts. All non-crashed TTCBs receive that R-broadcast and R-deliver $M(s,n)$. □

Theorem 5 *The Timely Reliable Broadcast protocol has the property of Timeliness and the constant $T_{broadcast}$ is given by:*

$$\begin{aligned} T_{broadcast} = \; & (WCET_{send} + T_{send} + T_r + WCET_{receive}) + (T_s) + \\ & (T_s + WCET_{send} + T_{send} + T_r + WCET_{receive}) + \pi \end{aligned} \tag{A.2}$$

Proof: The first component of this delay is the maximum time for the local TTCBs to receive the message R-broadcasted, $M(s,n)$ (first pair of brackets). The sender sends the next message, $M(s,n+1)$, T_s after $M(s,n)$. Therefore, the first two components of the formula give the maximum time for the local TTCBs to receive $M(s,n+1)$. The third component of the formula gives the extra time for the non-crashed local TTCBs to receive a message with $n+1$ associated with s and to R-deliver $M(s,n)$. If there is a constant $T_{broadcast}$ then the protocol has the property of Timeliness. □

Theorem 6 *For the crash-tolerant TBA service protocol, T_{TBA} is given by:*

$$T_{TBA} = T_s + T_{broadcast} \tag{A.3}$$

Proof: After the last process proposes or *tstart* expires, the TBA service takes at most a send period T_s to R-broadcast the last propose. Then, all local TTCBs take at most $T_{broadcast}$ to R-deliver the message. □

A.4 Byzantine reliable multicast protocol

This section proves that the BRM-M protocol is a reliable multicast and tolerates f failures out of $f + 2$ processes. In fact the protocol tolerates any number of faulty processes but the problem is vacuous if there are less than two correct processes. In those situations, the protocol definition does not impose any particular behavior.

In Section 4.1 we exemplified the cases in which a process was failed. Here we recall those cases as a set of conditions. A process is failed (or not correct) if:

- F1. The process does not follow the protocol or it crashed.
- F2. The process can not communicate with the TTCB or is impersonated by an attacker, e.g., if the attacker managed to capture the process' pair (eid, secret).
- F3. An attacker manages to falsify a MAC that should have been created by the process, e.g., if the attacker discovers one of the processes' symmetric keys.
- F4. The process can not send or receive successive copies of a message because its communication is systematically disrupted by an attacker.
- F5. The process does not get the result of a TBA because the TTCB discarded that result (the TTCB discards results after some time).

BRM-M is a reliable multicast protocol if it satisfies the properties of Validity, Agreement and Integrity as defined in Section 4.2.1.

Theorem 7 *If a correct process multicasts a message M then a correct process eventually delivers M (Validity).*

Proof: This property refers to a correct sender and a correct recipient. If a correct sender multicasts a message then a correct recipient in $group(M)$ (i.e., in $elist$) will eventually receive it, since F1, F2 and F4 are false. This will happen either due to the message sent in the first multicast (line 4) or to a retransmission (line 28). The correct recipient

also receives the correct $H(M)$, since F1, F2 and F5 are false and the TBA service gives correct results. After receiving a message and the hash, the correct recipient will eventually deliver the message (F1 and F2 are false). This will happen either because phase 1 was completed successfully (lines 11-12), or because it will eventually leave the loop (possibly after $Od + 1$ message multicasts) and end phase 2 (line 32). □

Theorem 8 *If a correct process delivers a message M then all correct processes eventually deliver M (Agreement).*

Proof: First, let us prove that if a correct recipient p delivers a message M then all correct recipients in $group(M)$ eventually receive M. If p is correct then it follows the protocol (F1 is false) and it delivers M only after: (1) receiving ACKs from all recipients in $group(M)$, i.e., in $elist$ (lines 8-12 and 25-26); or (2) sending $Od + 1$ copies of the message to every recipient from which it did not receive an ACK (lines 27-29 and 31). If p receives an ACK from another recipient then, either the ACK was genuine (sent by the recipient) or fake. If the ACK was fake then the corresponding recipient is not correct (condition F3) and therefore the property of Agreement does not apply to this process. In case (2), if p sends $Od + 1$ copies of a message to a recipient then either that recipient receives the message or it is failed and the property does not apply (F4). Therefore, if p delivers M then all correct recipients receive M.

Now we have to prove that if a correct recipient p' receives M then it eventually delivers M. If p' is correct it follows the protocol (condition F1) and it manages to communicate with the TTCB (condition F2). Since process p delivered M, then it must have obtained a correct $H(M)$ from the TTCB TBA. Therefore, since condition F5 is false, p' can also get $H(M)$ from the TBA. After receiving M and checking that it is the message corresponding to $H(M)$, p' will eventually deliver the message.

This proves that if a correct recipient delivers M then all correct recipients deliver M. A correct sender always delivers the message so this proves that the protocol verifies the Agreement property. □

Theorem 9 *For any message M, every correct process delivers M at most once, and only if M was previously multicast by $sender(M)$ (Integrity).*

Proof: For all messages with the same pair $(elist, tstart)$, every correct process runs a single instance of the protocol code. Additionally, an instance of the protocol always returns after delivering a message (lines 12 and 32). Therefore, every correct process delivers a message M at most once. Any correct process not in $group(M)$, i.e., not in $elist$, can not get $H(M)$ from the TTCB, therefore it can not deliver M. Now let us prove the second part of the property. The process $sender(M)$ is the process whose eid is the first in $elist$. The value returned by the TBA is the $H(M)$ proposed by the first element in $elist$ (decision TBA_RMULTICAST in line 7), i.e., by $sender(M)$ since it is correct (F2). Therefore, the value of $H(M)$ returned by the TBA is always the value proposed by the sender. Consequently, a correct process can deliver a message M only if M was previously multicast by $sender(M)$, since a correct process follows the protocol (F1) and checks if the hash of the message it received is equal to $H(M)$ (assuming the hash function is collision resistant). □

Theorem 10 *BRM-M tolerates the failure of f processes out of $n \geq f + 2$.*

Proof: Theorems 7–9 do not impose any limit on the number of processes that can fail. Therefore there is no such limit. The limit of $n \geq f + 2$ expresses only the notion that the problem is vacuous if there are not at least two correct processes. □

A.5 Block consensus protocol

This section proves the correctness of the Block Consensus protocol in Algorithm 4. The protocol is correct if it satisfies its definition, given by the properties of Validity, Agreement and Termination in Section 5.2.1. Here we consider the assumptions in Section 5.1 and in Section 5.2.2. We consider that the protocol is executed by n processes and that no more than $f = \lfloor \frac{n-1}{3} \rfloor$ can fail.

Theorem 11 *If all correct processes propose the same value* v*, then any correct process that decides, decides* v *(Validity).*

Proof: The theorem applies only when all correct processes (at least $2f + 1$) propose the same value v. The algorithm is a loop inside lines 3 to 10. All processes begin with the same *tstart* (assumption in Section 5.2.2) that works as the loop counter.

Each round of the loop, the correct processes call *TTCB_propose* and get the result of the TBA *out_dec* calling *TTCB_decide* (line 6). *out_dec* contains the value proposed by *more* processes before *tstart* (due to the decision function TBA_MAJORITY) and the two masks saying which processes proposed the value decided and which proposed any value before *tstart*. Each round can satisfy one of two cases, depending on the number of processes k that proposed before *tstart*:

Case 1 ($k < 2f + 1$): This case can be subdivided in another two. (Case 1a): If no $f + 1$ processes proposed the value decided, then the loop goes to the next round (line 10). (Case 1b): If $f + 1$ processes proposed the value decided then this value is v, since there are at most f failed processes (the theorem assumes all correct processes propose v). In the end of the round, the loop terminates since $f + 1$ proposed the same value (line 10). The value v is decided (line 11).

Case 2 ($k \geq 2f + 1$): Since there are at most f failed processes, the majority of processes that proposed are correct and the value decided by the TBA is v. The loop terminates (line 10) and v is decided (line 11).

Any correct process that decides, decides in cases (1b) or (2), therefore it decides v.
□

Theorem 12 *No two correct processes decide differently (Agreement).*

Proof: Two correct processes execute the same TBAs, since they start with the same *tstart* (Section 4) and TBA returns the same values to all processes (TTCB assumption).

Two correct processes exit the loop in the same round since they test the same condition (line 10) with the same results of TBA's. They return the same result for the same reason (line 11). □

Theorem 13 *Every correct process eventually decides (Termination).*

Proof: We start by showing that there is a round when *enough* processes manage to propose to the TBA before $tstart$. After a round where not enough processes proposed, accordingly to line 10, another round is scheduled, with a longer delay for processes to propose until *tstart*. The procedure is repeated, each time with a larger delay, until eventually there is a round where all non-failed processes have a chance to propose (assumption in Section 5.2.4). Since $n \geq 3f + 1$, even if f failed processes are mute, at least $2f + 1$ processes will have proposed, which satisfies the termination condition in line 10. The protocol can terminate earlier if there is an earlier round in which $f + 1$ processes manage to propose the same value. □

Theorems 11, 12 and 13 prove that the Block Consensus protocol satisfies its definition with $f = \lfloor \frac{n-1}{3} \rfloor$. However, the protocol also satisfies the properties of Validity and Agreement even if we allow a higher number of processes to be failed, more precisely, if $f = \lfloor \frac{n-1}{2} \rfloor$. The proofs are trivial modifications to the proofs of Theorems 11 and 12.

A.6 General consensus protocol

This section proves the correctness of the General Consensus protocol (Algorithm 5). The protocol is correct if it satisfies its definition, given by the properties of Validity, Agreement and Termination in Section 5.2.1. We make the same assumptions as for the Block Consensus protocol in the previous section, plus those in Section 5.2.3 and in Section 5.1 (communication model). We consider also that the protocol is executed by n processes and that no more than $f = \lfloor \frac{n-1}{3} \rfloor$ can fail. The same discussion made in relation to the Block Consensus protocol also applies here. The General Consensus protocol satisfies the properties of Validity and Agreement even if $f = \lfloor \frac{n-1}{2} \rfloor$.

Lemma 3 *If no more than $f = \lfloor \frac{n-1}{3} \rfloor$ processes fail and all correct processes propose the same value then the protocol does not change to phase 2.*

Proof: The change to phase 2 is tested in line 17. When $2f + 1$ processes proposed (first part of the condition), at least $f + 1$ processes proposed the same value (second part) since there are at least $2f + 1$ correct processes. Therefore, if the first part of the condition in line 17 is satisfied, the second is not, and the protocol does not change to phase 2. □

Theorem 14 *If all correct processes propose the same value* v*, then any correct process that decides, decides* v *(Validity).*

Proof: The theorem applies only when all correct processes propose the same value v, therefore the protocol does not change to phase 2 (Lemma 3). The phase 1 of the protocol is very similar to the Block Consensus protocol, therefore the proof that any correct process that decides, decides the same hash $H(v)$ follows from Theorem 11. If a process is correct then it eventually receives its own message with v (lines 6, 21). Therefore, any correct process that decides, decides v (lines 23, 26). □

Theorem 15 *No two correct processes decide differently (Agreement).*

Proof: The proof that no two correct processes decide different hashes is similar to Theorem 12. If two correct processes decide the same hash then they decide the same value due to the properties assumed for the hash function (lines 23 and 26, Section 5.2.3). □

Theorem 16 *Every correct process eventually decides (Termination).*

Proof: We base ourselves on the reasoning of the proof of Theorem 13, to ascertain that either all correct processes eventually terminate in phase 1 (line 19) or they change to phase 2 (line 17).

Let us now prove that all correct processes in phase 2 eventually decide. All correct processes multicast their values v_i to all others (line 6). Assuming the communication model, eventually every correct process receives the messages with the values v_i from all correct processes. Line 10 chooses the value v_j proposed by the process with index $(r \ mod \ n)$ in *elist* or the next one available. Again under the reasoning of the proof of Theorem 13, eventually $f + 1$ processes manage to propose the same $H(v_j)$, which is decided by the TBA. If a process has the value v_j in bag then it decides immediately (lines 19-20, 23-26). If a process p does not have the value v_j then it eventually receives it, since at least one correct process q has v_j (since $f + 1$ have it) and multicasts it (line 24-25). After receiving v_j, p decides it (lines 21-26). □

A.7 Membership service

This appendix proves that the membership service protocols satisfy the properties of Uniqueness, Validity, Integrity and Liveness (Section 6.2). Here we consider the system model presented in Section 6.1 and assumptions in Sections 6.2 and 6.3. In these proofs we use f^n to indicate the maximum number of sites allowed to fail in the view number n: $f^n = \lfloor \frac{|V^n|-1}{3} \rfloor$.

Lemma 4 *For all correct sites S_j that installed V_j^n and V_j^{n+1}, $V_j^{n+1} = V^{n+1}$.*

Proof: Correct sites in view V^n execute the VCA protocol (Alg. 7) to agree on the view changes to V^n that give the new view V^{n+1}. The *last TBA* of the VCA protocol decides the hash of the view changes to be applied to V^n. The proof can be divided in two cases:

Case 1 If at site S_j the hash decided in the last TBA is equal to *Hash(bag-changes-prop)* (Alg. 7, lines 10-19) then applying the changes in *bag-changes-prop* to V^n it gets V^{n+1} so it installs V^{n+1}.

Case 2 If that is not true, at least $2f^n + 1$ sites proposed for the last TBA (line 15) therefore at least $f^n + 1$ *correct* sites did it (since there are at most f^n failed sites). All correct sites will eventually multicast a CHANGES message (line 18), S_j will eventually receive one of them (line 22), terminate VCA (lines 24-25) and install V^{n+1}.

□

Theorem 17 *If views V_i^n and V_j^n are defined, and sites S_i and S_j are correct, then $V_i^n = V_j^n$ (Uniqueness).*

Proof: The proof is by induction on views. The group is created by site S_1 so the initial case is when site S_2 joins (view V^2). We assume S_2 manages to get a reliable copy of V_1 that contains only S_1 (Section 6.3.5). Then S_2 sends S_1 a *REQ_TO_JOIN* and waits for $f^1 + 1 = \frac{|V^1|-1}{3} + 1 = 1$ message with the new view information. Since there is only one site in V^1, no site is allowed to fail ($f^1 = 0$), therefore considering the communication model, S_2 gets a correct copy of V^2 and $V_1^2 = V_2^2$.

The proof that $(V_i^n = V_j^n) \Rightarrow (V_i^{n+1} = V_j^{n+1})$ (if neither of the sites exit the group) comes directly from Lemma 4.

Finally, considering that sites can join the group, we have also to prove that $V_i^{n+1} = V_j^{n+1}$ even if V_i^n was defined but S_j joined the group (V_j^n was not defined). We assume S_j manages to get a reliable copy of V^n. Then S_j multicasts a *REQ_TO_JOIN* and waits for $f^n + 1$ messages with the new view information. Considering the communication model and that there are at least $2f^n+1$ correct sites (and at most f^n failed), S_j eventually receives $f^n + 1$ identical copies of V^{n+1} and installs that view. □

Theorem 18 *If site S_i is correct and view V_i^n is defined, then $S_i \in V_i^n$ and, for all correct sites $S_j \in V_i^n$, V_j^n is eventually defined (Validity).*

Proof: The first part of the proof – that $S_i \in V_i^n$ – is trivial. Let us prove that for all correct sites $S_j \in V_i^n$, V_j^n is eventually defined. The case of the first view, V^1, is also

trivial. For all other views, V^n is installed after the sites in view V^{n-1} executed the VCA protocol. Therefore, all sites S_j belong to V_i^n for one of two reasons: $S_j \in V^{n-1}$ and S_j did not exit the group; or S_j joined the group to view V^n.

Case 1 $S_j \in V^{n-1}$ and S_j did not exit the group. Let us prove that all correct sites received at least $2f^{n-1}+1$ messages *(INFO, *, $Ev(S_k)$, *)* from different sites in view V^{n-1}. VCA terminated therefore at least $2f^{n-1}+1$ sites proposed for a last TBA (Alg. 7, line 15), so at least $f^{n-1}+1$ *correct* sites proposed for that TBA. For those correct sites to call TBA, they must have received at least $2f^{n-1}+1$ INFO messages about one event (Alg. 6, lines 17-21). At least $f^{n-1}+1$ of those messages were sent by correct sites so all correct sites will eventually receive them (communication model). A correct site eventually multicasts an *(INFO, *, $Ev(S_k)$, *)*, either because it "saw" the event or when it receives the $(f^{n-1}+1)^{th}$ message. Therefore, all correct sites eventually multicast the message and receive it at least $2f^{n-1}+1$ times, the minimum number of correct sites.

VCA is called with the smallest *valid-tstart* received in the INFO messages (Alg. 6, line 21). Let us now prove that the smallest *valid-tstart* received in any subset of these $2f^{n-1}+1$ messages, from different sites and for this view, is smaller or equal than the last TBA *tstart* (Alg. 7, line 10). A correct site multicasts all INFO messages for a view with the same *valid-tstart* (Alg. 6, lines 4, 8-9, 14-16). In any subset of $2f^{n-1}+1$ messages from different sites, at least $f^{n-1}+1$ were sent by correct sites. The last TBA happens when at least $2f^{n-1}+1$ sites manage to propose before *tstart* (Alg. 7, line 15), from which at least $f^{n-1}+1$ are correct. Since there are at least $2f^{n-1}+1$ correct sites, the intersection of these two sets of $f^{n-1}+1$ correct sites has at least one site. A correct site cannot propose with a *tstart* smaller than the *valid-tstart* it sends in its INFO messages for a view (Alg. 6, lines 8-10, 14-16). Therefore, the smallest *valid-tstart* a correct site receives in any subset of $2f^{n-1}+1$ INFO messages is smaller or equal to the first TBA when $2f^{n-1}+1$ sites manage to propose, i.e., the last TBA.

A site can receive INFO messages with several events $Ev(S_j)$ in the same view.

When it receives for the first time the $(2f^{n-1}+1)^{th}$ INFO message (from different sites) with the same event, it starts executing VCA with the smallest *valid-tstart* in these messages. It calls TBA once or more times, until it gets the result of the last TBA (Alg. 7, lines 8-15) and installs the view, i.e., V_j^n is defined.

Case 2 S_j joined the group. All correct sites in view V^{n-1} eventually install V^n since they are included in Case 1. Therefore they eventually send the new view information to all sites accepted to join (Alg. 6, line 26). When a site allowed to join receives $f^{n-1}+1$ of these messages it installs the new view.

□

Lemma 5 *If any correct site S_i receives $2f+1$ (INFO, *, $Ev(S_j)$, *) messages for the same view V^n and from different senders in the view, then at least one correct site in the view "saw" the event $Ev(S_j)$. The meaning of "saw" depends on the event: S_i detected the failure of S_j (event* Remv(S_j)*); S_i received a (LEAVE, S_j) message from S_j (event* Leave(S_j)*); or S_i received a (REQ_TO_JOIN, S_j, *) message from S_j (event* Join(S_j)*).*

Proof: In the proof we call simply *message* to a message *(INFO, *, $Ev(S_j)$, *)* and *n messages* to *n* of these messages received from different sites and for the view being considered.

A correct site multicasts a message for one of two reasons (Alg. 6, respectively lines 6-10 and 11-16): because it "saw" the event $Ev(S_j)$; because it received $f+1$ messages. Assume no correct site "saw" the event $Ev(S_j)$. If there are messages it is because a malicious site has sent it. Since there are at most f failed sites, no correct sites receives $f+1$ of messages, therefore no correct site sends messages, so no correct site receives $2f+1$ messages. □

Theorem 19 *If site $S_i \in V_i^n$ and V_i^{n+1} is not defined then at least one correct site detected that S_i failed or S_i requested to leave. If site $S_i \in V_i^{n+1}$ and V_i^n was not defined at S_i then at least one correct site authorized S_i to join (Integrity).*

Proof: First let us prove the first sentence. If site $S_i \in V_i^n$ but V_i^{n+1} is never defined then S_i exited the group. This is only possible if VCA made agreement on a *bag-changes-prop* with an event *Leave(S_i)* or *Remv(S_i)*. A correct site puts an event $Ev(S_j)$ in *bag-changes* when it receives the $(2f+1)^{th}$ message *(INFO, *, $Ev(S_j)$, *)* from different sites and for this view (Alg. 6, lines 17-18). Given Lemma 5, that happens only if at least one correct site "sees" the event, *Leave(S_i)* or *Remv(S_i)*. To "see" the *Leave(S_i)* means to receive a message *(LEAVE, S_i)* from S_i. To "see" the event *Remv(S_i)* means to detect the failure of S_i. This proves the first sentence. The proof of the second is similar so we skip it for brevity. □

Theorem 20 *If $\lfloor \frac{|V^n|-1}{3} \rfloor + 1$ correct sites detect that S_i failed or receive a request to join, or one correct site requests to leave, then eventually V^{n+1} is installed, or the join is rejected (Liveness).*

Proof: We prove the assertion for the site failure and skip the proofs for join and leave since they are similar. If $f^n + 1 = \lfloor \frac{|V^n|-1}{3} \rfloor + 1$ correct sites detect that S_i failed they multicast a total of $f^n + 1$ messages *(INFO, *, Remv(S_i))*. There are at least $2f^n + 1$ correct sites and all multicast these messages either because they detected the failure or because they received the mentioned f^n+1 messages. Therefore, all correct sites receive at least $2f^n + 1$ of these messages. If VCA is not running then it starts and eventually decides that view change, considering the time assumption made in Section 6.3.7. If VCA is already running that change can be decided or not. If not, we assume the event is re-issued in the next view and that it will be eventually agreed (Section 6.3.2). □

A.8 View-synchronous atomic multicast protocol

This appendix sketches proofs of the correctness of the view-synchronous atomic multicast protocol presented in Chapter 7. We consider the system model and the assumptions in Section 7.1.

Theorem 21 *If a correct site multicasts a message M, then some correct site in* group(M) *eventually delivers M (Validity).*

Proof: When a site wants to leave a group it also multicast a message to all members (Section 6.3.4. Therefore, the proof follows from Theorem 20. □

Theorem 22 *If a correct site delivers a message M, then all correct sites in* group(M) *eventually deliver M (Agreement).*

Proof: The proof also follows from Theorem 20. □

Theorem 23 *For any message M, every correct site p delivers M at most once and only if p is in* group(M)*, and if* sender(M) *is correct then M was previously multicast by* sender(M) *(Integrity).*

Proof: The proof follows trivially from the secure channels model (Section 5.1). □

Theorem 24 *If two correct sites install views V^n and V^{n+1} then both sites deliver the same messages in view V^n (View synchrony).*

Proof: Any correct site delivers a BAM-VS message only if the VCMDA protocol says so. New views are also installed after the execution of a VCA protocol, which is similar to VCMDA, therefore the proof follows from the proof of Uniqueness (Theorem 17). □

Theorem 25 *If two correct sites deliver two messages M_1 and M_2 then both sites deliver the two messages in the same order (Total order).*

Proof: If M_1 and M_2 are delivered in consequence of the result of the same VCMDA execution the proof is obvious (see Algorithm 9, line 23). The proof that a correct site eventually engages in an execution of VCMDA follows from Theorem 20. This has also the consequence that all correct sites execute the same sequence of VCMDAs in a view. Therefore, if M_1 and M_2 are delivered in two different executions of VCMDA, the property is also satisfied. □

References

ADELSBACH, A., ALESSANDRI, D., CACHIN, C., CREESE, S., DESWARTE, Y., KURSAWE, K., LAPRIE, J. C., POWELL, D., RANDELL, B., RIORDAN, J., RYAN, P., SIMMONDS, W., STROUD, R., VERÍSSIMO, P., WAIDNER, M., & WESPI, A. 2002. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*.

ADELSBACH, A., CREESE, S., HARRISON, R., PFITZMANN, B., SADEGHI, A., SIMMONDS, W., STEINER, M., STUBLE, C., & WAIDNER, M. 2003. *Final Report on Verification and Assessment. Project MAFTIA deliverable D22*.

ALVISI, L., MALKHI, D., PIERCE, E., & REITER, M. 1999 (Jan.). Fault Detection for Byzantine Quorum Systems (Extended Abstract). *In: Proceedings of the 7th IFIP International Working Conference on Dependable Computing for Critical Applications*.

AMES, S., GASSER JR., M., & SCHELL, R. 1983. Security Kernel Design and Implementation: An Introduction. *IEEE Computer*, **16**(7), 14–22.

AMIR, Y., DOLEV, D., KRAMER, S., & MALKHI, D. 1992 (July). Transis: A Communication Subsystem for High Availability. *In: Proceedings of the 22nd IEEE Fault-Tolerant Computing Symposium*.

AMIR, Y., ATENIESE, G., HASSE, D., KIM, Y., NITA-ROTARU, C., SCHLOSSNAGLE, T., SCHULTZ, J., STANTON, J., & TSUDIK, G. 2000 (Apr.). Secure Group Communication in Asynchronous Networks with Failures: Integration and Experiments. *Pages 330–343 of: Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*.

AMIR, Y., KIM, Y., NITA-ROTARU, C., SCHULTZ, J., STANTON, J., & TSUDIK, G. 2001 (Apr.). Exploring Robustness in Group Key Agreement. *In: 21th IEEE International Conference on Distributed Computing Systems*.

ARMSTRONG, J., CACHIN, C., CORREIA, M., COSTA, A., MIRANDA, H., NEVES, N.F., NEVES, N. M., PORITZ, J., RANDELL, B., RODRIGUES, L., STROUD, R., VERÍSSIMO, P., WAIDNER, M., & WELCH, I. 2001. *First Specification of APIs and Protocols for the MAFTIA Middleware. Project MAFTIA deliverable D24*.

ARMSTRONG, J., CACHIN, C., CORREIA, M., COSTA, A., MIRANDA, H., NEVES, N. F., NEVES, N. M., PORITZ, J. A., RANDELL, B., LUNG, L. C., RODRIGUES, L., STROUD, R. J., VERÍSSIMO, P., WAIDNER, M., & WELCH, I. S. 2002. *Complete Specification of APIs and Protocols for the MAFTIA Middleware. Project MAFTIA deliverable D9*.

ATENIESE, G., STEINER, M., & TSUDIK, G. 2000. New Multi-party Authentication Services and Key Agreement Protocols. *IEEE Journal of Selected Areas in Communications*, **18**(Mar.).

AVIZIENIS, A. 1985. The N-Version Approach to Fault Tolerant Software. *IEEE Transactions on Software Engineering*, **11**(12), 1491–1501.

AVIZIENIS, A., LAPRIE, J. C., & RANDELL, B. 2001. *Fundamental Concepts of Dependability*. Tech. rept. 01145. LAAS-CNRS, Toulouse, France.

BABAOĞLU, Ö. 1987. On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems. *ACM Transactions on Computer Systems*, **5**(3), 394–416.

BABAOĞLU, Ö., & DRUMMOND, R. 1985. Streets of Bizantium: Network Architectures for Fast Reliable Broadcasts. *IEEE Transactions on Software Engineering*, **11**(6), 546–554.

BABAOĞLU, Ö., DRUMMOND, R., & STEPHENSON, P. 1986 (July). The Impact of Communication Network Properties on Reliable Broadcast Protocols. *Pages 212–217 of: Proceedings of the 16th IEEE International Symposium on Fault-Tolerant Computing*.

BALDONI, R., HELARY, J., RAYNAL, M., & TANGUY, L. 2000 (June). Consensus in Byzantine Asynchronous Systems. *Pages 1–16 of: Proceedings of the International Colloquium on Structural Information and Communication Complexity*.

BARABANOV, M. 1997 (June). *A Linux-Based Real-Time Operating System*. M.Phil. thesis, New Mexico Institute of Mining and Technology.

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

BIRMAN, K., & JOSEPH, T. 1987a (Nov.). Exploiting Virtual Synchrony in Distributed Systems. *Pages 123–138 of: Proceedings of the 11th Symposium on Operating System Principles*.

BIRMAN, K., & JOSEPH, T. 1987b. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, **5**(1), 46–76.

BIRMAN, K. P. 1997. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall.

BOM, J., MARQUES, P., CORREIA, M., & PINTO, P. 1998 (June). QoS Control: an Application Integrated Framework. *In: Proceedings of the 1st IEEE International Conference on ATM*.

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

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

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

CACHIN, C., CORREIA, M., MCCUTCHEON, T., NEVES, N.F., PFITZMANN, B., RANDELL, B., SCHUNTER, M., SIMMONDS, W., STROUD, R., VERÍSSIMO, P., WAIDNER, M., & WELCH, I. 2001. *Service and Protocol Architecture for the MAFTIA Middleware. Project MAFTIA deliverable D23*.

CANETTI, R., GARAY, J., ITKIS, G., MICCIANCIO, D., NAOR, M., & PINKAS, B. 1999 (Mar.). Multicast Security: A Taxonomy and Efficient Constructions. *In: Proceedings of the INFOCOM '99*.

CASIMIRO, A., & VERÍSSIMO, P. 1999 (Apr.). Timing Failure Detection with a Timely Computing Base. *In: Proceedings of the European Research Seminar on Advances in Distributed Systems*.

CASIMIRO, A., & VERÍSSIMO, P. 2001 (Oct.). Using the Timely Computing Base for Dependable QoS Adaptation. *In: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*.

CASIMIRO, A., & VERÍSSIMO, P. 2002 (June). Generic Timing Fault Tolerance using a Timely Computing Base. *Pages 27–36 of: Proceedings of the 2002 International Conference on Dependable Systems and Networks*.

CASIMIRO, A., MARTINS, P., & VERÍSSIMO, P. 2000 (Sept.). How to Build a Timely Computing Base using Real-Time Linux. *Pages 127–134 of: Proceedings of the IEEE International Workshop on Factory Communication Systems*.

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

CASTRO, M., & LISKOV, B. 2000 (Oct.). Proactive Recovery in a Byzantine-Fault-Tolerant System. *Pages 273–288 of: Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*.

CASTRO, M., & LISKOV, B. 2001 (June). Byzantine Fault Tolerance Can Be Fast. *Pages 513–518 of: Proceedings of the IEEE International Conference on Dependable Systems and Networks*.

CESARE, S. 1998 (Nov.). *Runtime Kernel kmem Patching*. http://www.big.net.au/~ silvio/.

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

CHANDRA, T., HADZILACOS, V., & TOUEG, S. 1996. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, **43**(4), 685–722.

CHOCKLER, G., KEIDAR, I., & VITENBERG, R. 2001. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, **33**(4), 427–469.

CLOUTIER, P., MANTEGAZZA, P., PAPACHARALAMBOUS, S., SOANES, I., HUGHES, S., & YAGHMOUR, K. 2000 (Nov.). DIAPM-RTAI Position Paper. *In: Real-Time Linux Workshop*.

COLLBERG, C., THOMBORSON, C., & LOW, D. 1998a (May). Breaking Abstractions and Unstructuring Data Structures. *In: Proceedings of the IEEE International Conference on Computer Languages*.

COLLBERG, C., THOMBORSON, C., & LOW, D. 1998b (Jan.). Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. *In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

CORREIA, M., & PINTO, P. 1995 (Oct.). Low-Level Multimedia Synchronization Algorithms on Broadband Networks. *Pages 423–434 of: Proceedings of the 3rd ACM International Conference on Multimedia*.

CORREIA, M., VERÍSSIMO, P., & NEVES, N. F. 2001a (Apr.). The Architecture of a Secure Group Communication System Based on Intrusion Tolerance. *Pages 17–22 of: Proceedings of the 21st International Conference on Distributed Computing Systems Workshops (International Workshop on Applied Reliable Group Communication)*.

CORREIA, M., VERÍSSIMO, P., & NEVES, N. F. 2001b. *The Design of a COTS Real-Time Distributed Security Kernel (Extended Version)*. DI/FCUL TR 01–12. Department of Computer Science, University of Lisbon.

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

CORREIA, M., LUNG, L. C., NEVES, N. F., & VERÍSSIMO, P. 2002b (Oct.). Efficient Byzantine-Resilient Reliable Multicast on a Hybrid Failure Model. *Pages 2–11 of: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*.

CORREIA, M., NEVES, N. F., LUNG, L. C., & VERÍSSIMO, P. 2003a. *Low Complexity Byzantine-Resilient Consensus*. Submitted for publication.

CORREIA, M., NEVES, N. F., LUNG, L. C., & VERÍSSIMO, P. 2003b (August). *Low Complexity Byzantine-Resilient Consensus*. DI/FCUL TR 03–25. Department of Informatics, University of Lisbon.

CORREIA, M., NEVES, N. F., LUNG, L. C., & VERÍSSIMO, P. 2003c. *WIT-GCS – A Wormhole-based Intrusion-Tolerant Group Communication System*. Submitted for publication.

COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGLE, P., & GLIGOR, V. 2000 (Dec.). SubDomain: Parsimonious Server Security. *In: Proceedings of the USENIX 14th Systems Administration Conference*.

CRISTIAN, F. 1991. Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems. *Distributed Computing*, **4**(4), 175–187.

CRISTIAN, F., & FETZER, C. 1998. The Timed Asynchronous System Model. *Pages 140–149 of: Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*.

CRISTIAN, F., AGHILI, H., STRONG, R., & DOLEV, D. 1985. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Pages 200–206 of: Proceedings of the 15th IEEE International Symposium on Fault-Tolerant Computing*.

DEBAR, H., DACIER, M., & WESPI, A. 1999. Towards a Taxonomy of Intrusion Detection Systems. *Computer Networks*, **31**(8), 805–822.

DÉFAGO, X., SCHIPER, A., & URBÁN, P. 2000. *Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey*. TR DSC 2000-036. École Polytechnique Fédérale de Lausanne.

DENNING, D. 1987. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, **13**(2), 222–232.

DENNING, D. E., & NEUMANN, P. G. 1985. *Requirements and Model for IDES - A Real-Time Intrusion Detection Expert System*. Tech. rept. Computer Science Laboratory, SRI International, Menlo Park, CA.

DESMEDT, Y., & FRANKEL, Y. 1989. Threshold Cryptosystems. *Pages 307–315 of:* BRASSARD, G. (ed), *Advances in Cryptology–Crypto '89*. Lecture Notes in Computer Science, vol. 435. Springer-Verlag.

DESWARTE, Y., BLAIN, L., & FABRE, J. C. 1991 (May). Intrusion Tolerance in Distributed Computing Systems. *Pages 110–121 of: Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*.

DIFFIE, W., & HELLMAN, M. 1979. Privacy and Authentication: An Introduction to Cryptography. *Proceedings of the IEEE*, **67**(3), 397–427.

DIFFIE, W., & HELLMAN, M. E. 1976. New Directions in Cryptography. *IEEE Transactions on Information Theory*, **22**(6), 644–654.

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

DOUDOU, A., & SCHIPER, A. 1997. *Muteness Failure Detectors for Consensus with Byzantine Processes*. Tech. rept. 97/30. EPFL.

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

DUTERTRE, B., SAIDI, H., & STAVRIDOU, V. 2001 (June). Intrusion-Tolerant Group Management in Enclaves. *In: Proceedings of the IEEE International Conference on Dependable Systems and Networks*.

DUTERTRE, B., CRETTAZ, V., & STAVRIDOU, V. 2002 (May). Intrusion-Tolerant Enclaves. *Pages 216–226 of: Proceedings of the IEEE International Symposium on Security and Privacy*.

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

EASTLAKE, D., CROCKER, S., & SCHILLER, J. 1994 (Dec.). *Randomness Recommendations for Security*. Network Working Group, Request for Comments: 1750.

FISCHER, M. J. 1983. The Consensus Problem in Unreliable Distributed Systems (A Brief Survey). *Pages 127–140 of:* KARPINSKY, M. (ed), *Foundations of Computing Theory*. Lecture Notes in Computer Science, vol. 158. Springer-Verlag.

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

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

FRASER, T. 2000. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. *In: Proceedings of the IEEE Symposium on Security and Privacy*.

FRIEDMAN, R. 2002. *A Simple Coding Theory-Based Characterization of Conditions for Solving Consensus*. Tech. rept. CS-2002-06. Department of Computer Science, The Technion.

FRIEDMAN, R., & VAN RENESSE, R. 1996 (Oct.). Strong and weak virtual synchrony in Horus. *Pages 140–149 of: Proceedings of the 15th Symposium on Reliable Distributed Systems*.

GONG, L. 1994 (Nov.). New Protocols for a Third-Party-Based Authentication and Secure Broadcast. *Pages 176–183 of: Proceedings of the 2nd ACM Conference on Computer and Communications Security*.

GONG, L. 1997. Enclaves: Enabling Secure Collaboration over the Internet. *IEEE Journal of Selected Areas in Communications*, **11**(5), 567–575.

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

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

HARDJONO, T., CAIN, B., & DORASWAMY, N. 1999 (Feb.). *A Framework for Group Key Management for Multicast Security*. Internet Draft.

HOHL, F. 1998. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. *Pages 92–113 of:* VIGNA, G. (ed), *Mobile Agents and Security*. Lecture Notes in Computer Science, vol. 1419. Springer-Verlag.

HUAGANG, X. 2000 (Oct.). *Build a Secure System with LIDS*. http://www.lids.org.

ITOI, N., & HONEYMAN, P. 1999 (May). Smartcard Integration with Kerberos V5. *In: Proceedings of the USENIX Workshop on Smartcard Technology*.

KEIDAR, I. 2002 (June). Challenges in Evaluating Distributed Algorithms. *In: Proceedings of the International Workshop on Future Directions in Distributed Computing*.

KIECKHAFER, R., & AZADMANESH, M. 1995. Unified Approach to Synchronous and Asynchronous Approximate Agreement in the Presence of Hybrid Faults. *IEEE Transactions on Reliability*, **44**(4).

KIHLSTROM, K. P., MOSER, L. E., & MELLIAR-SMITH, P. M. 2001. The SecureRing Group Communication System. *ACM Transactions on Information and System Security*, **4**(4), 371–406.

KIHLSTROM, K. P., MOSER, L. E., & MELLIAR-SMITH, P. M. 2003. Byzantine Fault Detectors for Solving Consensus. *The Computer Journal*, **46**(1), 16–35.

LAMPORT, L. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, **21**(7), 558–565.

LAMPORT, L., SHOSTAK, R., & PEASE, M. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, **4**(3), 382–401.

LAMPSON, B. 1974. Protection. *Operating Systems Review*, **8**(1), 18–24.

LAPRIE, J.-C. 1991. Dependability Concepts. *In:* POWELL, D. (ed), *Delta-4 A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag.

LINCOLN, P., & RUSHBY, J. 1993. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. *In: Proceedings of the 23th IEEE International Symposium on Fault-Tolerant Computing*.

LUNG, L. C., CORREIA, M., NEVES, N. F., & VERÍSSIMO, P. 2003 (May). A Simple Intrusion-Tolerant Reliable Multicast Protocol using the TTCB. *Pages 649–663 of: 21º Simpósio Brasileiro de Redes de Computadores*.

LYU, M., CHEN, J., & AVIZIENIS, A. 1992 (Sept.). Software diversity metrics and measurements. *Pages 69–78 of: Proceedings of the Sixteen Annual International Computer Software and Applications Conference*.

MALKHI, D., & REITER, M. 1997a (May). Byzantine Quorum Systems. *Pages 569–578 of: Proceedings of the 29th ACM Symposium in Theory of Computing*.

MALKHI, D., & REITER, M. 1997b. A High-Throughput Secure Reliable Multicast Protocol. *The Journal of Computer Security*, **5**, 113–127.

MALKHI, D., & REITER, M. 1997c (June). Unreliable Intrusion Detection in Distributed Computations. *Pages 116–124 of: Proceedings of the 10th Computer Security Foundations Workshop*.

MALKHI, D., & REITER, M. 1998 (Oct.). Secure and Scalable Replication in Phalanx. *In: Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*.

MALKHI, D., REITER, M., & WOOL, A. 1997a (Aug.). The Load and Availability of Byzantine Quorum Systems. *Pages 249–257 of: Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*.

MALKHI, D., REITER, M., & WRIGHT, R. 1997b (Aug.). Probabilistic Quorum Systems. *Pages 267–273 of: Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*.

MALKHI, D., MERRIT, M., & RODEH, O. 1997c. Secure Reliable Multicast Protocols in a WAN. *Pages 87–94 of: International Conference on Distributed Computing Systems*.

MALKHI, D., REITER, M. K., TULONE, D., & ZISKIND, E. 2001 (June). Persistent Objects in the Fleet System. *In: Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*.

MALKI, D., & REITER, M. 1996 (June). A High-Throughput Secure Reliable Multicast Protocol. *Pages 9–17 of: Proceedings of the 9th IEEE Computer Security Foundations Workshop*.

MCHUGH, J. 2001. Intrusion and Intrusion Detection. *International Journal of Information Security*, **1**(1), 14–35.

MELLIAR-SMITH, P., MOSER, L., & AGRAWALA, V. 1990. Broadcast Protocols For Distributed Systems. **1**(1), 17–25.

MENEZES, A. J., OORSCHOT, P. C. VAN, & VANSTONE, S. A. 1997. *Handbook of Applied Cryptography*. CRC Press.

MEYER, F., & PRADHAN, D. 1987 (July). Consensus with Dual Failure Modes. *Pages 214–222 of: Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing*.

MIRANDA, H., PINTO, A., & RODRIGUES, L. 2001 (Apr.). Appia, a Flexible Protocol Kernel Supporting Multiple Coordinated Channels. *Pages 707–710 of: Proceedings of the 21st International Conference on Distributed Computing Systems Workshops*.

MITTRA, S. 1997 (Sept.). Iolus: A Framework for Scalable Secure Multicasting. *In: Proceedings of the ACM SIGCOMM*.

MOSER, L., AMIR, Y., MELLIAR-SMITH, P., & AGARWAL, D. 1994 (June). Extended Virtual Synchrony. *Pages 56–65 of: The 14th IEEE International Conference on Distributed Computing Systems*.

MOSER, L. E., & MELLIAR-SMITH, P. M. 1999. Byzantine-Resistant Total Ordering Algorithms. *Information and Computation*, **150**, 75–111.

MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., & LINGLEY-PAPADOPOULOS, C. A. 1996. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, **39**(4), 54–63.

MOSER, L. E., MELLIAR-SMITH, P. M., & NARASIMHAN, N. 2000 (Jan.). The Secure-Group Communication System. *Pages 507–516 of: Proceedings of the IEEE Information Survivability Conference*.

MOSTEFAOUI, A., RAJSBAUM, S., & RAYNAL, M. 2001 (July). Conditions on Input Vectors for Consensus Solvability in Asynchronous Distributed Systems. *Pages 152–162 of: Proceedings of the 33rd ACM Symposium on Theory of Computing*.

NATIONAL COMPUTER SECURITY CENTER. 1983 (Aug.). *Trusted Computer Systems Evaluation Criteria*.

NATIONAL COMPUTER SECURITY CENTER. 1987 (July). *Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria*.

POWELL, D. 1992 (July). Fault Assumptions and Assumption Coverage. *In: Proceedings of the 22nd IEEE International Symposium of Fault-Tolerant Computing*.

POWELL, D., SEATON, D., BONN, G., VERÍSSIMO, P., & WAESELYNK, F. 1988 (June). The Delta-4 Approach to Dependability in Open Distributed Computing Systems. *In: Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing*.

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

RAMASAMY, H. 2002. *A Group Membership Protocol for an Intrusion-Tolerant Group Communication System*. M.Phil. thesis, University of Illinois at Urbana-Champaign.

RAMASAMY, H., PANDEY, P., LYONS, J., CUKIER, M., & SANDERS, W. H. 2002 (June). Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems. *Pages 229–238 of: Proceedings of the International Conference on Dependable Systems and Networks*.

REISCHUCK, R. 1982 (Nov.). *A New Solution for the Byzantine General's Problem*. Tech. rept. RJ 3673. IBM Research Lab.

REITER, M. 1994 (Nov.). Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. *Pages 68–80 of: Proceedings of the 2nd ACM Conference on Computer and Communications Security*.

REITER, M., BIRMAN, K., & GONG, L. 1992 (May). Integrating Security in a Group Oriented Distributed System. *Pages 18–32 of: Proceedings of the IEEE Symposium on Research in Security and Privacy*.

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

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

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

REITER, M. K., & BIRMAN, K. P. 1994. How to Securely Replicate Services. *ACM Transactions on Programming Languages and Systems*, **16**(3), 986–1009.

REITER, M. K., BIRMAN, K. P., & VAN RENNESSE, R. 1994. A Security Architecture for Fault-Tolerant Systems. *ACM Transactions on Computer Systems*, **12**(4), 340–371.

RIVEST, R. L., SHAMIR, A., & ADLEMAN, L. M. 1978. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, **21**(2), 120–126.

RODEH, O., BIRMAN, K., & DOLEV, D. 2001a. The Architecture and Performance of Security Protocols in the Ensemble Group Communication System. *ACM Transactions on Information and System Security*, **4**(3), 289–319.

RODEH, O., BIRMAN, K., & DOLEV, D. 2001b. Using AVL Trees for Fault Tolerant Group Key Management. *International Journal on Information Security*, **1**(2), 84–99.

RODRIGUES, L., & VERÍSSIMO, P. 2000. Topology-aware algorithms for large-scale communication. *Chap. 6, pages 127–156 of:* KRAKOWIAK, S., & SHRIVASTAVA, S. (eds), *Advances in Distributed Systems*. Lecture Notes in Computer Science, vol. 1752. Springer-Verlag.

SANDER, T., & TSCHUDIN, C. 1998. Protecting Mobile Agents Against Malicious Hosts. *Pages 44–60 of:* VIGNA, G. (ed), *Mobile Agents and Security*. Lecture Notes in Computer Science, vol. 1419. Springer-Verlag.

SCHIPER, A. 1997. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, **10**(Oct.), 149–157.

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

SHOUP, V., & RUBIN, A. D. 1996 (May). Session Key Distribution Using Smart Cards. *In:* SPRINGER-VERLAG (ed), *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques, Eurocrypt'96*. Lecture Notes in Computer Science, vol. 1070.

SMITH, S. W., PALMER, E. R., & WEIGART, S. 1998 (Feb.). Using a High-Performance, Programmable Secure Coprocessor. *In:* SPRINGER-VERLAG (ed), *Proceedings of the Second International Conference on Finantial Cryptography*. Lecture Notes in Computer Science.

STABELL-KULØ, T., ARILD, R., & MYRVANG, P. H. 1999 (May). Providing Authentication to Messages Signed with a Smart Card in Hostile Environments. *In: Proceedings of the USENIX Workshop on Smartcard Technology*.

STEINER, M., TSUDIK, G., & WAIDNER, M. 2000. Key Agreement in Dynamic Peer Groups. *IEEE Transactions on Parallel and Distributed Systems*, **11**(8), 769–780.

TALLEY, T., & JEFFAY, K. 1994 (Oct.). Two-Dimensional Scaling Techniques for Adaptive, Rate-Based Transmission Control of Live Audio and Video Streams. *In: Proceedings of the 2nd ACM International Multimedia Conference*.

TCPA. 2002 (Feb.). *Trusted Computing Platform Alliance. Main Specification Version 1.1b*. Tech. rept. TCPA.

TOBOTRAS, B. 1999. *Linux Capabilities FAQ 0.2*. ftp://ftp.guardian.no/pub/free/linux/capabilities/capfaq.txt.

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

TYGAR, J. D., & YEE, B. S. 1993 (Apr.). Dyad: A System for Using Physically Secure Coprocessors. *In: Proceedings of the Joint Harvard-MIT Workshop on Technological*

Strategies for the Protection of Intellectual Property in the Network Multimedia Environment.

VERÍSSIMO, P. 2003. Uncertainty and Predictability: Can They Be Reconciled? *Pages 108–113 of: Future Directions in Distributed Computing*. Lecture Notes in Computer Science, vol. 2584. Springer-Verlag.

VERÍSSIMO, P., & ALMEIDA, C. 1995. Quasi-Synchronism: a Step Away from the Traditional Fault-Tolerant Real-Time System Models. *Bullettin of the Technical Committee on Operating Systems and Application Environments*, **7**(4), 35–39.

VERÍSSIMO, P., & CASIMIRO, A. 2002. The Timely Computing Base Model and Architecture. *IEEE Transactions on Computers*, **51**(8), 916–930.

VERÍSSIMO, P., & RODRIGUES, L. 2001. *Distributed Systems for System Architects*. Kluwer Academic Publishers.

VERÍSSIMO, P., RODRIGUES, L., & BAPTISTA, M. 1989. AMp: A Highly Parallel Atomic Multicast Protocol. *Pages 83–93 of: Proceedings of the ACM SIGCOMM*.

VERÍSSIMO, P., RODRIGUES, L., & CASIMIRO, A. 1997. CesiumSpray: a Precise and Accurate Global Time Service for Large-scale Systems. *Journal of Real-Time Systems*, **12**(3), 243–294.

VERÍSSIMO, P., NEVES, N. F., & CORREIA, M. 2000a (Oct.). The Middleware Architecture of MAFTIA: A Blueprint. *In: Proceedings of the Third IEEE Information Survivability Workshop*.

VERÍSSIMO, P., CASIMIRO, A., & FETZER, C. 2000b (June). The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness. *Pages 533–542 of: Proceedings of the International Conference on Dependable Systems and Networks*.

VERÍSSIMO, P. E., NEVES, N. F., & CORREIA, M. P. 2003. Intrusion-Tolerant Architectures: Concepts and Design. *Pages 3–36 of:* LEMOS, R., GACEK, C., & ROMANOVSKY, A. (eds), *Architecting Dependable Systems*. Lecture Notes in Computer Science, vol. 2677. Springer-Verlag.

WALTER, C., SURI, N., & HUGUE, M. 1994. Continual On-Line Diagnosis of Hybrid Faults. *In: Proceedings of the 4th IFIP International Working Conference on Dependable Computing for Critical Applications*.

WANG, C., DAVIDSON, J., HILL, J., & KNIGHT, J. 2001 (July). Protection of Software-based Survivability Mechanisms. *Pages 193–202 of: Proceedings of the International Conference of Dependable Systems and Networks*.

WONG, C. K., GOUDA, M., & LAM, S. S. 1998 (Sept.). Secure Group Communications Using Key Graphs. *Pages 68–79 of: Proceedings of the ACM SIGCOMM*.

YODAIKEN, V., & BARABANOV, M. 1999. *RTLinux Version Two*. http://www.fsmlabs.com/archive/design.pdf.