



Project no.: IST-FP6-STREP - 027513
Project full title: Critical Utility InfrastructurAL Resilience
Project Acronym: CRUTIAL
Start date of the project: 01/01/2006 **Duration:** 36 months
Deliverable no.: D10
Title of the deliverable: Preliminary Specification of Services and Protocols

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Contractual Date of Delivery to the CEC:	17/01/2008
Actual Date of Delivery to the CEC:	17/01/2008
Organisation name of lead contractor for this deliverable	FCUL
Editor(s):	Nuno Neves ² , Paulo Verissimo ²
Author(s):	Anas Abou El Kalan ⁴ ; Amine Baina ⁴ ; Hakem Beitollahi ⁵ ; Alysson Bessani ² ; Andrea Bondavalli ³ ; Miguel Correia ² ; Alessandro Daidone ³ ; Geert Deconinck ⁵ ; Yves Deswarte ⁴ ; Fabrizio Garrone ¹ ; Fabrizio Grandoni ³ ; Henrique Moniz ² ; Nuno Neves ² ; Tom Rigole ⁵ ; Paulo Sousa ² ; Paulo Verissimo ²
Participant(s):	(1) CESI-R; (2) FCUL; (3) CNR-ISTI; (4) LAAS-CNRS; (5) KUL; (6) CNIT
Work package contributing to the deliverable:	WP4
Nature:	R
Dissemination level:	PU
Version:	004
Total number of pages:	121

Abstract

This document describes the preliminary specification of services and protocols for the Crutial Architecture. The Crutial Architecture definition, first addressed in Crutial Project Technical Report D4 (January 2007), intends to reply to a grand challenge of computer science and control engineering: how to achieve resilience of critical information infrastructures, in particular in the electrical sector.

The definitions herein elaborate on the major architectural options and components established in the Preliminary Architecture Specification (D4), with special relevance to the Crutial middleware building blocks, and are based on the fault, synchrony and topological models defined in the same document. The document, in general lines, describes the Runtime Support Services and APIs, and the Middleware Services and APIs. Then, it delves into the protocols, describing: Runtime Support Protocols, and Middleware Services Protocols.

The Runtime Support Services and APIs chapter features as a main component, the Proactive-Reactive Recovery Service, whose aim is to guarantee perpetual execution of any components it protects.

The Middleware Services and APIs chapter describes our approach to intrusion-tolerant middleware. The middleware comprises several layers. The Multipoint Network layer is the lowest layer of CRUTIAL's middleware, and features an abstraction of basic communication services, such as provided by standard protocols, like IP, IPsec, UDP, TCP and SSL/TLS. The Communication Support Services feature two important building blocks: the Randomized Intrusion-Tolerant Services (RITAS), and the Overlay Protection Layer (OPL) against DoS attacks. The Activity Support Services currently defined comprise the CIS Protection service, and the Access Control and Authorization service. Protection as described in this report is implemented by mechanisms and protocols residing on a device called *Crutial Information Switch* (CIS). The Access Control and Authorization service is implemented through PolyOrBAC, which defines the rules for information exchange and collaboration between sub-modules of the architecture, corresponding in fact to different facilities of the CII's organizations. The Monitoring and Failure Detection layer contains a preliminary definition of the middleware services devoted to monitoring and failure detection activities.

The remaining chapters describe the protocols implementing the above-mentioned services: Runtime Support Protocols, and Middleware Services Protocols.

Contents

Table of Contents	1
List of Figures	2
1 Introduction	5
2 Runtime Support Services and APIs	8
2.1 Proactive-Reactive Recovery Service	8
2.1.1 Overview	8
2.1.2 Model of the System	9
2.1.3 Service Description and Interface	10
3 Middleware Services and APIs	14
3.1 Multipoint Network	14
3.1.1 Internet Protocol	14
3.1.2 Internet Protocol Security	15
3.1.3 User Datagram Protocol and Transmission Control Protocol	16
3.1.4 Secure Socket Layer	18
3.2 Communication Support Services	19
3.2.1 Randomized Intrusion-Tolerant Services	19
3.2.2 Overlay Protection Against Denial-of-Service Attacks	23
3.3 Activity Support Services	37
3.3.1 CIS Protection Service	37
3.3.2 Access Control and Authorization Service	40
3.4 Monitoring and Failure Detection	48
3.4.1 Diagnosis Framework	48
3.4.2 Diagnosis in CRUTIAL	49

4	Runtime Support Protocols	53
4.1	Proactive-Reactive Recovery Protocols	53
4.1.1	System Model	53
4.1.2	The Protocols	53
5	Middleware Services Protocols	57
5.1	Multipoint Network	57
5.2	Communication Support	59
5.2.1	Randomized Intrusion-Tolerant Protocols	59
5.2.2	Overlay Network Protocols	71
5.3	Activity Support Protocols	78
5.3.1	CIS Protection Protocol	78
5.3.2	Access Control and Authorization Protocols	87
5.4	Monitoring and Failure Detection	99
5.4.1	Design Rationale	99
5.4.2	Services Specification	100
5.4.3	Diagnosing the Protection Service	102
5.4.4	Advantages and Limits of the PRRW Strategy	109
5.4.5	Direction for Improvements/Refinements	110
5.4.6	Architectural Modifications for the Detection of the Extended Set of Fault	112
6	Conclusions	114

List of Figures

1.1	CRUTIAL middleware.	6
2.1	Relationship between the rejuvenation period T_p , the rejuvenation execution time T_D , and k	11
2.2	PRRW architecture.	11
2.3	Recovery schedule (in an S_{ij} or R_i subslot there can be at most k parallel replica recoveries).	13
3.1	Countermeasure techniques against DoS attacks.	25
3.2	Rate control mechanism from high level view [41].	27
3.3	SoS architecture [54].	28
3.4	The OPL architecture.	29
3.5	Threat against the OPL architecture.	33
3.6	DoS attack in the static case.	33
3.7	DoS attack in the dynamic case.	34
3.8	DDoS attack in the dynamic case.	34
3.9	Impact of node joining/leaving for a) DoS and b) DDoS attacks. (X-axis: ρ ; Y-axis: percentage of absent nodes; Z-axis: probability of a successful search) . . .	35
3.10	Impact of the number of green nodes per application on the OPL performance. . .	36
3.11	Delay of the overlay network.	36
3.12	WAN-of-LANs connected by CIS.	38
3.13	The general functioning.	41
3.14	Scenario 1 exchanged commands.	42
3.15	Scenario 2 exchanged commands.	44
3.16	Scenario 3 exchanged commands.	46
3.17	Scenario 4 exchanged commands.	47
5.1	The RITAS protocol stack.	59
5.2	Flooding search in Type I protocols.	71
5.3	Chord routing algorithm.	72

5.4	Routing algorithm in Napster.	73
5.5	Hybrid decentralized indexing.	73
5.6	Intrusion-tolerant CIS architecture.	80
5.7	Scenario 1 users and services.	88
5.8	Scenario 2 users and services.	90
5.9	Arming Web Service Sequence Diagram.	94
5.10	Scenario 3 users and services.	95
5.11	Scenario 4 users and services.	96
5.12	Teleoperation Web Service Sequence Diagram.	98
5.13	System failure probability $P_F(t)$ over mission time t for different values of p_I . . .	105
5.14	System unavailability $P_U(t)$ over mission time t for different values of p_I	106
5.15	Impact of detection coverage c_M on failure probability $P_{FI}(t)$ due to invalid behavior for different values of p_I	107
5.16	Impact of detection coverage c_M on failure probability $P_{FO}(t)$ due to omissive behavior for different values of p_I	107
5.17	Impact of detection coverage c_M on system failure probability $P_F(t)$ for different values of p_I	107
5.18	Impact of detection coverage c_M on system unavailability $P_U(t)$ for different values of p_I	108
5.19	System failure probability $P_F(t)$ for different system configurations at mission time $t=2628$	109
5.20	System unavailability $P_U(t)$ for different system configurations at mission time $t=2628$	109

1 Introduction

This document describes the preliminary specification of services and protocols for the Crutial Architecture. The Crutial Architecture definition, first addressed in Crutial Project Technical Report D4 (January 2007), intends to reply to a grand challenge of computer science and control engineering: how to achieve resilience of critical information infrastructures, in particular in the electrical sector.

The scope considered spans the threats against computers and control computers, not the physical infrastructures themselves. The focus are systems with great socio-economic value, such as utility systems like electrical, gas or water, or telecommunication systems.

In the above-mentioned report, we laid down the basis for our work, whose final objectives are:

- Ensuring acceptable levels of service and, in last resort, the integrity of systems themselves, when faced with threats of several kinds. Doing so in an **automatic** and **adaptive** way.
- Taking into account the **hybrid** composition of those infrastructures: operational network, called generically SCADA, devoted to the physical processes; corporate intranet, where usual departmental services and clients reside; Internet, through which intranet users get to other intranets and/or the outside world; interconnections between all of these, including SCADA-Internet.

Intrusion tolerance, a workhorse of the Crutial approach, advocates the use of redundancy to ensure that a system still delivers its service correctly even if some of its components are compromised. Typical protocols, said of 'Byzantine resilience', or 'Byzantine fault tolerance', only operate correctly if at most a specified number f out of the n available replicas are compromised. However, given a sufficient amount of time, a malicious and intelligent adversary can find ways to compromise more than f replicas and collapse the whole system. The problem can be minimized if the replicas are rejuvenated periodically, using a technique called *proactive recovery*, to remove the effects of malicious attacks/faults. In fact, if the rejuvenation is performed sufficiently often, then an attacker is never able to corrupt enough replicas to break the system, and the latter operates perpetually. These paradigms outline the background of the approach of Crutial to resilience.

In this document, we present a preliminary specification of services and protocols of the Crutial Architecture. The definitions herein elaborate on the major architectural options and components established in the Preliminary Architecture Specification (D4), with special relevance to the CRUTIAL middleware building blocks, and are based on the fault, synchrony and topological models defined in the same document. The document, in general lines, describes the Runtime Support Services and APIs, and the Middleware Services and APIs. Then, it delves into the protocols, describing: Runtime Support Protocols, and Middleware Services Protocols.

The Runtime Support Services and APIs chapter features as a main component, the Proactive-Reactive Recovery Service, whose aim is to guarantee perpetual execution of any components it protects.

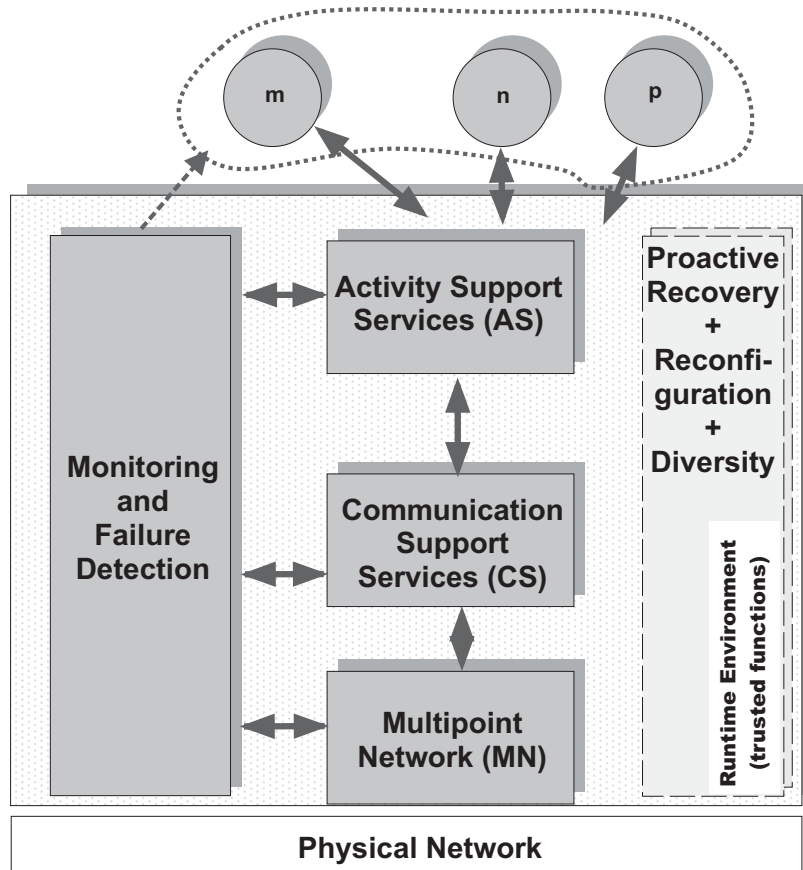


Figure 1.1: CRUTIAL middleware.

In Crutial we investigated limitations of existing approaches to intrusion-tolerant proactive recovery, and proposed a very complete scheme addressing them, which we named *proactive-reactive recovery*. Our first observation is that protecting oneself from timing attacks by using asynchronous models, and fulfilling periodic recoveries, are incompatible goals. To address this issue, we propose an innovative scheme based on a hybrid sync-asynchronous architecture, called *proactive resilience*. Our second observation is that one should allow correct replicas that detect or suspect that some replica is faulty, to accelerate the recovery of this replica. It is known that perfect Byzantine failure detection is impossible to attain in a general way. In consequence, dealing with imperfect failure detection is the most complex aspect of the proactive-reactive recovery service presented.

The Middleware Services and APIs chapter describes our approach to intrusion-tolerant middleware (see Figure 1.1). The middleware comprises several layers.

The Multipoint Network layer is the lowest layer of CRUTIAL’s middleware, and features an abstraction of basic communication services, such as provided by standard protocols, like IP, IPsec, UDP, TCP and SSL/TLS.

The Communication Support Services feature two important building blocks: the Randomized Intrusion-Tolerant Services (RITAS), and the Overlay Protection Layer (OPL) against DoS attacks. The Randomized Intrusion-Tolerant Services (RITAS) are organised as a stack of ran-

domized intrusion-tolerant protocols, supporting applications who depend on intrusion-tolerant broadcast and agreement. These protocols, being randomized, overcome the impossibility result in asynchronous settings established in [38] (also called the FLP result), but present a significant performance improvement over previous protocols of the same class. In recent years DoS attacks have become one of the most serious security threats to the Internet. Today Internet protocols have become an emerging technology for remote control of industrial applications, and as such, vulnerable to the same kind of attacks. We address an overlay protection layer for DoS attacks on top of the normal infrastructure, for solving DoS Problem.

The Activity Support Services currently defined comprise the CIS Protection service, and the Access Control and Authorization service.

The CRUTIAL reference architecture models the whole infrastructure architecture as a WAN-of-LANs. This topology allows simple solutions to hard problems such as legacy control subnetworks, and interconnection of critical and non-critical traffic, by defining realms with different levels of trustworthiness. The CIS Protection service protects realms from one another, i.e., a LAN from another LAN or from the WAN, thus allowing us to deal both with outsider and insider threats. Protection as described in this report is implemented by mechanisms and protocols residing on a device called *CRUTIAL Information Switch* (CIS).

The Access Control and Authorization service is implemented through PolyOrBAC, which defines the rules for information exchange and collaboration between sub-modules of the architecture, corresponding in fact to different facilities of the Critical Information Infrastructures (CII). Each organization specifies its security policy according to OrBAC. As organizations are interconnected through CIS, each CIS regroups mechanisms to define security policy of systems that compose each LAN (local and collaboration policies), and it also regroups mechanisms for collaboration: to make these LANs capable of collaboration and offering services to each other.

The Monitoring and Failure Detection section contains a preliminary definition of the middleware services devoted to monitoring and failure detection activities. Diagnosis in Cru-tial should occur at different components at different architectural levels, and as such, the classical framework has been extended: several components need to be monitored and several deviation detection mechanisms need to be in place, errors observed in different components must be correlated. Likewise, given that we are dealing with a complex infrastructure, methods for distributed diagnosis are mandatory, with a distinction between local and global detection and diagnosis.

The remaining chapters describe the protocols implementing the above-mentioned services: Runtime Support Protocols, and Middleware Services Protocols.

2 Runtime Support Services and APIs

2.1 Proactive-Reactive Recovery Service

This section describes the proactive-reactive recovery service and its interface. The protocols used to implement this service are presented in Section 4.1. Before describing the proactive-reactive recovery service, we start by motivating the necessity of such a service, and by explaining the system model in which the proactive-reactive recovery service is based.

2.1.1 Overview

One of the most challenging requirements of distributed systems being developed nowadays is to ensure that they operate correctly despite the occurrence of accidental and malicious faults (including security attacks and intrusions). In the context of CRUTIAL, this problem is specially relevant for an important class of systems that are employed in mission-critical applications such as the SCADA systems used to manage critical infrastructures like the Power grid. One approach that promises to satisfy this requirement and that gained momentum recently is *intrusion tolerance* [90]. This approach recognizes the difficulty in building a completely reliable and secure system and advocates the use of redundancy to ensure that a system still delivers its service correctly even if some of its components are compromised.

A problem with “classical” intrusion-tolerant solutions based on Byzantine fault-tolerant replication algorithms is the assumption that the system operates correctly only if at most f out of n of its replicas are compromised. The problem here is that given a sufficient amount of time, a malicious and intelligent adversary can find ways to compromise more than f replicas and collapse the whole system.

Recently, some works showed that this problem can be solved (or at least minimized) if the replicas are rejuvenated periodically, using a technique called *proactive recovery* [69]. These previous works propose intrusion-tolerant replicated systems that are resilient to any number of faults [17, 101, 15, 61, 80]. The idea is simple: replicas are periodically rejuvenated to remove the effects of malicious attacks/faults. Rejuvenation procedures may change the cryptographic keys and/or load a clean version of the operating system. If the rejuvenation is performed sufficiently often, then an attacker is unable to corrupt enough replicas to break the system. Therefore, using proactive recovery, one can increase the resilience of any intrusion-tolerant replicated system able to tolerate up to f faults/intrusions: an unbounded number of intrusions may occur during its lifetime, as long as no more than f occur between rejuvenations. Both the interval between consecutive rejuvenations and f must be specified at system deployment time according to the expected rate of fault production.

An inherent limitation of proactive recovery is that a malicious replica can execute any action to disturb the system’s normal operation (e.g., flood the network with arbitrary packets) and there is little or nothing that a correct replica (that detects this abnormal behavior) can do

to stop/recover the faulty replica. Our observation is that a more complete solution should allow correct replicas *that detect or suspect that some replica is faulty to accelerate the recovery of this replica*. We named this solution as *proactive-reactive recovery* and claim that it may improve the overall performance of a system under attack by reducing the amount of time a malicious replica has to disturb system normal operation without sacrificing periodic rejuvenation, which ensures that even dormant faults will be removed from the system. The key property of our approach is that, as long as the fault exhibited by a replica is *detectable*, this replica will be recovered as soon as possible, ensuring that there is always an amount of system replicas available to sustain system's correct operation.

We recognize that perfect Byzantine failure detection is impossible to attain in a general way, since what characterizes a malicious behavior is dependent on the application semantics [32, 31, 7, 45]. However, we argue that an important class of malicious faults can be detected, specially the ones generated automatically by malicious programs such as virus, worms, and even botnets. These kinds of attacks have little or no intelligence to avoid being detected by replicas carefully monitoring the environment. However, given the imprecisions of the environment, some behaviors can be interpreted as faults, while in fact they are only effects of overloaded replicas. In this way, a reactive recovery strategy must address the problem of (possible wrong) suspicions to ensure that recoveries are scheduled according to some fair policy in such a way that there is always a sufficient number of replicas for the system to be available. In fact, dealing with imperfect failure detection is the most complex aspect of the proactive-reactive recovery service presented in this section.

2.1.2 Model of the System

Recently, it was shown that proactive recovery can only be implemented with a few synchrony assumptions [81, 82]: in short, in an asynchronous system a compromised replica can delay its recovery (e.g., by making its local clock slower) for a sufficient amount of time to allow more than f replicas to be attacked. To overcome this fundamental problem, the proactive-reactive recovery service is based on a hybrid system model [89] in which the system is composed of two parts, with distinct properties and assumptions, let us call them *payload* and *wormhole*.

Payload. Any-synchrony system with $n \geq af + bk + 1$ replicas P_1, \dots, P_n . This part can range from fully asynchronous to fully synchronous. At most f replicas can be subject to *Byzantine failures* in a given recovery period and at most k replicas can be recovered at the same time. The exact threshold depends on the application. For example, an asynchronous Byzantine fault-tolerant state machine replication system requires $n \geq 3f + 2k + 1$ while the CIS Protection Service presented in Section 3.3.1 requires only $n \geq 2f + k + 1$. If a replica does not fail between two recoveries it is said to be *correct*, otherwise it is said to be *faulty*. We assume fault-independence for payload replicas, i.e., the probability of a replica being faulty is independent of the occurrence of faults in other replicas. This assumption can be substantiated in practice through the extensive use of several kinds of diversity [68].

Wormhole. Synchronous subsystem with n local wormholes in which at most f local wormholes

can fail by crash. These local wormholes are connected through a *synchronous and secure control channel*, isolated from other networks. There is one local wormhole per payload replica and we assume that when a local wormhole i crashes, the corresponding payload replica i crashes together. Since the local wormholes are synchronous and the control channel used by them is isolated and synchronous too, we assume several services in this environment:

1. wormhole clocks have a known precision, obtained by a clock synchronization protocol;
2. there is point-to-point timed reliable communication between every pair of local wormholes;
3. there is a timed reliable broadcast primitive with bounded maximum transmission time [44];
4. there is a timed atomic broadcast primitive with bounded maximum transmission time [44].

One should note that all of these services can be easily implemented in the crash-failure synchronous distributed system model [92].

2.1.3 Service Description and Interface

The Proactive Resilience Model (PRM) The proactive-reactive recovery service builds on the Proactive Resilience Model (PRM) briefly introduced in CRUTIAL deliverable D4. The PRM addresses proactive recovery and defines a system enhanced with proactive recovery through a model composed of two parts: the proactive recovery subsystem and the payload system, the latter being proactively recovered by the former. Each of these two parts obeys different timing assumptions and different fault models, and should be designed accordingly. The payload system executes the “normal” applications and protocols. Thus, the payload synchrony and fault model entirely depend on the applications/protocols executing in this part of the system. For instance, the payload may operate in an asynchronous Byzantine way. The proactive recovery subsystem executes the proactive recovery protocols that rejuvenate the applications/protocols running in the payload part. This subsystem is more demanding in terms of timing and fault assumptions, and it is modeled as a distributed component called *Proactive Recovery Wormhole* (PRW).

The Proactive Recovery Wormhole (PRW) The distributed PRW is composed of a local module in every host called the local PRW, which may be interconnected by a synchronous and secure control channel. The PRW executes periodic rejuvenations through a periodic timely execution service with two parameters: T_P and T_D . Namely, each local PRW executes a rejuvenation procedure F in rounds, each round is initiated within T_P from the last triggering, and the execution time of F is bounded by T_D . Notice that if local recoveries are not coordinated, then the system may present unavailability periods during which a large number (possibly all) replicas are recovering. For instance, if the replicated system tolerates up to f arbitrary faults, then it will typically become unavailable if $f + 1$ replicas recover at the same time, even if no “real” fault occurs. Therefore, if a replicated system able to tolerate f Byzantine servers is enhanced with periodic recoveries, then availability is guaranteed by (*i.*) defining the maximum number of replicas allowed to recover in

parallel (call it k); and (ii.) deploying the system with a sufficient number of replicas to tolerate f Byzantine servers and k simultaneous recovery servers. Figure 2.1 illustrates the rejuvenation process. Replicas are recovered in groups of at most k elements, by some specified order: for instance, replicas $\{P_1, \dots, P_k\}$ are recovered first, then replicas $\{P_{k+1}, \dots, P_{2k}\}$ follow, and so on. Notice that k defines the number of replicas that may recover simultaneously, and consequently the number of distinct $\lceil \frac{n}{k} \rceil$ rejuvenation groups that recover in sequence. For instance, if $k = 2$, then at most two replicas may recover simultaneously in order to guarantee availability. This means also that at least $\lceil \frac{n}{2} \rceil$ rejuvenation groups (composed of two replicas) will need to exist, and they can not recover at the same time. Notice that the number of rejuvenation groups determines a lower-bound on the value of T_P and consequently defines the minimum window of time an adversary has to compromise more than f replicas. From the figure it is easy to see that $T_P \geq \lceil \frac{n}{k} \rceil T_D$.

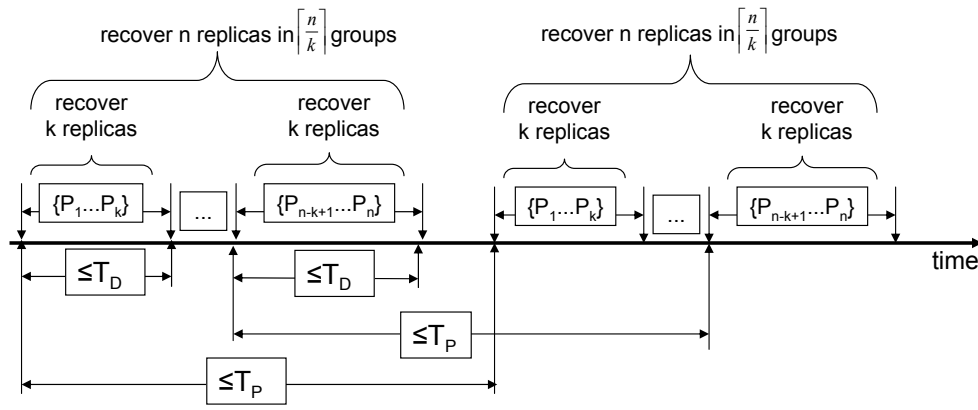


Figure 2.1: Relationship between the rejuvenation period T_P , the rejuvenation execution time T_D , and k .

The Proactive-Reactive Recovery Wormhole (PRRW) We extended the PRW to trigger both proactive and reactive recoveries and named the new component Proactive-Reactive Recovery Wormhole (PRRW). The PRRW is then the distributed component that offers the proactive-reactive recovery service. This service needs input information from the payload replicas in order to trigger *reactive recoveries*. This information is obtained through two interface functions: $W_suspect(j)$ and $W_detect(j)$. Figure 2.2 presents this idea.

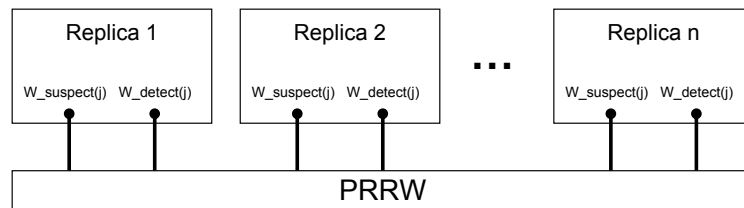


Figure 2.2: PRRW architecture.

A payload replica i calls $W_suspect(j)$ to notify the PRRW that the replica j is suspected of being failed. This means that replica i suspects replica j but it does not know for sure if it is really failed. Otherwise, if replica i knows without doubt that replica j is failed, then $W_detect(j)$

is called instead. Notice that the service is generic enough to deal with any kind of replica failures, e.g., crash and Byzantine. For instance, replicas may: use an unreliable crash failure detector [18] (or a muteness detector [32]) and call $W_suspect(j)$ when a replica j is suspected of being crashed; or detect that a replica j is sending unexpected messages or messages with incorrect content [7, 45], calling $W_detect(j)$ in this case.

If $f + 1$ different replicas suspect and/or detect that replica j is failed, then this replica is recovered. This recovery can be done immediately, without endangering availability, in the presence of at least $f + 1$ detections, given that in this case at least one correct replica detected that replica j is really failed. Otherwise, if there are only $f + 1$ suspicions, the replica may be correct and the recovery must be coordinated with the periodic proactive recoveries in order to guarantee that a minimum number of correct replicas is always alive to ensure the system availability. The quorum of $f + 1$ in terms of suspicions or detections is needed to avoid recoveries triggered by faulty replicas: at least one correct replica must detect/suspect a replica for some recovery action to be taken.

It is worth to notice that the proactive-reactive recovery service is completely orthogonal to the failure/intrusion detection strategy used by a system. The proposed service only exports operations to be called when a replica is detected/suspected to be faulty. In this sense, any approach for fault detection (including Byzantine) [18, 32, 7, 45], system monitoring [25] and/or intrusion detection [27, 63] can be integrated in a system that uses the PRRW. The overall effectiveness of our approach, i.e., how fast a compromised replica is recovered, is a direct consequence of detection/diagnosis accuracy.

Ensuring Availability The proactive-reactive recovery service initiates recoveries both periodically (time-triggered) and whenever something bad is detected or suspected (event-triggered). As explained before, periodic recoveries are done in groups of at most k replicas, so no more than k replicas are recovering at the same time. However, the interval between the recovery of each group is not tight. Instead we allocate $\lceil \frac{f}{k} \rceil$ intervals for recovery between periodic recoveries such that they can be used by event-triggered recoveries. This amount of time is allocated to make possible at most f recoveries between each periodic recovery, in this way being able to handle the maximum number of faults assumed.

The approach is based on real-time scheduling with an *aperiodic server task* to model aperiodic tasks [83]. The idea is to consider the action of recovering as a resource and to ensure that no more than k correct replicas will be recovering simultaneously. As explained before, this condition is important to ensure that the system always stays available. Two types of real-time tasks are utilized by the proposed mechanism:

- task R_i : represents the periodic recovery of up to k replicas (in parallel). All these tasks have worst case execution time T_D and period T_P ;
- task A : is the aperiodic server task, which can handle at most $\lceil \frac{f}{k} \rceil$ recoveries (of up to k replicas) every time it is activated. This task has worst case execution time $\lceil \frac{f}{k} \rceil T_D$ and period $(\lceil \frac{f}{k} \rceil + 1)T_D$.

Task R_i is executed at up to k different local wormholes, while task A is executed in all wormholes, but only the ones with the payload detected/suspected of being faulty are (aperiodically) recovered. The time needed for executing one A and one R_i is called the *recovery slot* i and is denoted by T_{slot} . Every slot i has $\lceil \frac{f}{k} \rceil$ *recovery subslots* belonging to the A task, each one denoted by S_{ip} , plus a R_i . Figure 2.3 illustrates how time-triggered periodic and event-triggered aperiodic recoveries are combined.

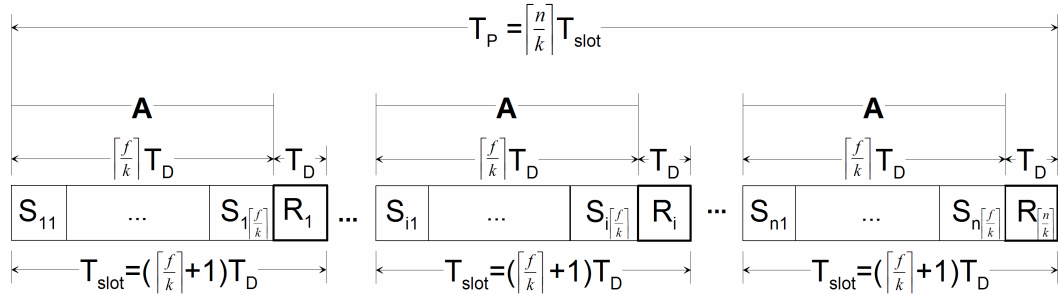


Figure 2.3: Recovery schedule (in an S_{ij} or R_i subslot there can be at most k parallel replica recoveries).

In the figure it is easy to see that when our reactive recovery scheduling approach is employed, the value of T_P must be increased. In fact, T_P should be greater or equal than $\lceil \frac{n}{k} \rceil (\lceil \frac{f}{k} \rceil + 1)T_D$, which means that reactive recoveries increase the rejuvenation period by a factor of $(\lceil \frac{f}{k} \rceil + 1)$. This is not a huge increase since f is expected to be small. In order to simplify the presentation of the algorithms, in the remaining of the report it is assumed that $T_P = \lceil \frac{n}{k} \rceil (\lceil \frac{f}{k} \rceil + 1)T_D$.

Notice that a reactive recovery only needs to be scheduled according to the described mechanism if the replica i to recover is only suspected of being failed (it is not assuredly failed), i.e., if less than $f + 1$ replicas have called $W_detect()$ (but the total number of suspicions and detections is higher than $f + 1$). If the wormhole W_i knows with certainty that replica i is faulty, i.e., if a minimum of $f + 1$ replicas have called $W_detect(i)$, replica i can be recovered without availability concerns, since it is accounted as one of the f faulty replicas.

3 Middleware Services and APIs

The Middleware Services and APIs chapter describes our approach to intrusion-tolerant middleware. The middleware comprises several layers, such as Multipoint Network, Communication and Activity Support, and Monitoring and Failure Detection.

3.1 *Multipoint Network*

Multipoint Network (MN) is the lowest layer of CRUTIAL's middleware. Its purpose is to offer a simple abstraction of the basic communication services provided by the underlying network infrastructure, which can then be utilized in an uniform way by the higher layers of the middleware. These services are said to be “basic” in the sense that they are implemented by standard protocols, like IP, IPsec, UDP, TCP and SSL/TLS. This section presents some of the protocols that can be integrated in the MN module, their services and APIs. The presentation is organized in terms of the two relevant layers of the TCP/IP reference model to which these protocols belong: Network and Transport. We skip the lowest layers for which there are many technologies and are too low level to be considered middleware: Ethernet (wired and wireless), SDH/ATM, Frame Relay, copper circuits, etc. Application layer protocols, like some protocols specific for critical infrastructures and industrial systems (MMS and ICCP), are also not described since they are seen at a higher level than the middleware.

3.1.1 Internet Protocol

The main service provided by the Network layer in the Internet is routing data packets – datagrams – from the source host to the destination host. Hosts are interconnected by special nodes called routers that inspect the datagrams to forward – or route – them to the next router or the destination. The format of the datagrams is defined by the most widely used Network layer protocol in the Internet, the Internet Protocol (IP). Nowadays, IP underlies most communication networks around the world, including the Internet, corporate networks and even some control networks, so it is important to give some insight about it.

The most important data in an IP datagram are the source and destination host addresses. A host or, more precisely, each host's network interface is identified by an IP address, which has 32 bits in IPv4, the current almost universally adopted version of IP. A shift to IPv6 is currently happening, although there is a high uncertainty about when it will end or even reach most of the Internet. IP also provides other services like fragmentation and reassembly of packets too big for the size of the packet transported by the physical network. IP does not ensure the reliability of the communication, i.e., datagrams can be dropped or duplicated.

IP can also be used to send messages to multiple destinations, something that is called IP multicast. This is important for CRUTIAL middleware since it involves multicast to several hosts,

e.g., for several CIS. The multicasted datagram is delivered to all members of a certain group with the same guarantees given by the regular IP datagrams: it is not guaranteed to reach all members, it is not guaranteed to arrive intact to all members and it is not guaranteed to arrive in the same order to all members, relative to other datagrams. Hosts can join and leave the group at any time, i.e., group membership is dynamic. Multicast groups cannot span the whole Internet since not all routers support this functionality. Typically there are “islands” of routers in the Internet that support it.

The classical API to IP is the sockets API, originally defined in Berkeley Unix. Several versions appeared since them, starting in other Unixes, and up to MS-Windows and Java, to give some examples. However, sockets are not usually used to send IP datagrams directly – so called raw sockets – but instead at transport level to send data over UDP or TCP, so more details are provided below. IP multicast is also typically used below UDP, and the same reasoning applies to the API.

3.1.2 Internet Protocol Security

Internet Protocol Security (IPSec) is an extension of IP that provides some level of security [52]. In its basic form, IP messages can be modified and its content read by anyone with access to the network, e.g., a hacker controlling a router. IPSec prevents this problem. IPSec has an important role in CRUTIAL since it is a basic mechanism to ensure security in the Network layer.

IPSec is designed to enhance the security of IPv4, providing interoperable, high-quality, cryptographically-based security. It offers several services, such as access control, connectionless integrity, data origin authentication, protection against replays, confidentiality (through encryption) and limited traffic flow confidentiality. Since these services are offered at the Network/IP layer, they can be used by any higher layer protocol, such as TCP, UDP, HTTP, etc. IPSec also supports negotiation of IP compression, motivated by the observation that encryption used within IPSec prevents effective compression by lower protocol layers.

IPSec is divided in two (sub)protocols, which may be applied alone or in combination with each other to provide the desired set of security properties at IP-level:

- Authentication Header (AH) – provides connectionless integrity, data origin authentication, and an optional anti-replay service.
- Encapsulation Security Payload (ESP) – provides payload confidentiality (using encryption) and limited traffic flow confidentiality. Optionally, it may also provide connectionless integrity, data origin authentication, and an anti-replay service.

Both AH and ESP are vehicles for access control, based on the distribution of cryptographic keys and the management of traffic flows relative to these security protocols. These protocols support two different modes of operation. At Transport mode, IPSec essentially protects

upper layer protocols (e.g., TCP). At Tunnel mode, the protocols are applied to tunneled IP packets, i.e., the IP datagrams themselves are sent through a secure tunnel. IPsec allows the user or the system administrator to control the granularity at which a security service is offered, allowing, for example, the creation of a single encrypted tunnel to carry all the traffic between two security gateways or a separate encrypted tunnel for each TCP connection between a pair of hosts communicating across these gateways. IPsec can be configured to protect only the integrity of the communication (preventing modifications) or the integrity and the confidentiality of the traffic.

Most IPsec implementations do not have an API that can be used by applications to transmit secure data, other than the socket API used for IP, UDP or TCP. IPsec works at the operating system level, and typically can only be configured by the system administrator. A system administrator can define the policy for IPsec on a host basis, determining the ways by which a host can connect securely to another.

3.1.3 User Datagram Protocol and Transmission Control Protocol

Network-level IP solves the problem of end-to-end communication between hosts. However, for implementing distributed applications, the problem that really has to be solved is slightly different: end-to-end communication between processes, since typically there are many processes running in each host. This is the problem solved by the Transport layer. In IP-based networks hosts are identified by IP addresses; inside a host, application are identified by ports (one or more), which are 16-bit numbers (range 0-65535). The standard Transport layer Internet protocols are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). Both protocols are used to support communication in critical infrastructures, so both are relevant for the CRUTIAL middleware.

UDP provides a datagram mode of packet-switched computer communication in an interconnected set of computer networks. Applications can send messages to other programs with a minimum set of guarantees using UDP. The key characteristics of the protocol are: it is transaction oriented, the delivery of messages is not ensured, nor is the order of message arrival, and there might be duplication of messages. UDP in fact is a thin layer on top of IP, which does not provide more guarantees, only adds information about the source and destination applications, i.e., the source and destination ports.

TCP, on the other hand, is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols supporting multi-network applications. Applications can send data using TCP, in a reliable way, to other programs on host computers attached to distinct but interconnected computer communication networks. TCP does not rely on the protocols below for reliability, but rather assumes that it can obtain a (potentially) unreliable datagram service from the lower level protocols, typically IP. A TCP connection serves to send a stream of data (not independent datagrams), which in practice is split in TCP segments. Reliability means that segments are delivered in the order they were sent and unmodified. In practice, these properties are ensured using a Cyclic Redundancy Check (CRC) to detect modifications, and retransmissions to recover from missing or corrupted segments. A disruption of the network can interrupt the delivery

of the stream of data if a timeout causes TCP to break the connection. TCP segments include the source and destination ports.

The CRC code in TCP segments is used to detect accidental modifications, e.g., due to line noise. However, in terms of security it does not protect the segments since a malicious hacker can modify the segment plus the CRC code to fit the segment modification. Malicious modifications have to be detected using Message Authentication Codes (MAC), like those provided by IPSec. In fact, reliable and secure end-to-end application communication can be implemented using TCP over IPSec.

TCP is a complex protocol, with several other mechanisms that are not discussed here. Examples are flow control (to prevent segments from being sent when the reception buffer has no space), slow start (to avoid contributing to network congestion when a TCP connection is established) and fast retransmit (to cause an earlier retransmission of missing segments).

The classical programming interface for TCP and UDP is the Berkeley Unix Socket API, although today there are many adaptations of this API available, like the Java sockets API, provided by the Java programming language. In what follows we consider the classical socket API. The three basic calls are:

- *socket()* – creates a socket, i.e., a communication endpoint with an IP address, a protocol (TCP or UDP) and a port (set by default);
- *bind()* – associates a specific IP address, protocol and port to the socket;
- *close()* – destroys a socket.

In the case of TCP there are a few specific calls related to establishing a connection between two machines: a server, that waits for connections, and a client, that makes connections. The calls are:

- *listen()* – executed in the server side to state the maximum number of connections that may be pending at a certain instant;
- *accept()* – blocks the server waiting for connections, or picks a pending connection;
- *connect()* – called by the client to establish a connection with a server.

There are several calls used to send and receive messages, such as *write()*, *sendto()*, *read()* and *recvfrom()*. To configure some parameters of the sockets there are calls like *ioctl()* and *setsockopt()* that can be used. For instance, *setsockopt()* can be used to add/remove a host to/from an IP multicast group. Finally, the *select()* call is often used for a server to block waiting for messages from several sockets, instead of only one. Alternatively, a server can be multithreaded and have one thread blocked waiting for messages in each port.

3.1.4 Secure Socket Layer

The Secure Socket Layer (SSL) [46, 40], later standardized as Transport Layer Security (TLS), is a security extension to TCP. It basically provides authentication of the hosts involved in the communication, and confidentiality and integrity of the communication. SSL/TLS is a modification of TCP. The initial handshaking is followed by a negotiation of the cryptographic algorithms to use and the creation of a session key. Authentication is based on public-key cryptography and digital certificates, and can be mutual (both peers authenticate themselves), one-way or simply not done. Integrity and (optionally) confidentiality of data are guaranteed using the session key, respectively by adding a MAC and encrypting the data. The security guarantees provided by SSL/TLS are similar to those provided by TCP over IPsec, except for the more powerful authentication scheme and the usual availability of a user-level API, something that is not common with IPsec.

SSL/TLS is provided by packets like OpenSSL and languages like Java. The basic APIs tend to be quite similar to the TCP sockets API. However there are usually a set of calls to define the location of the certificates, if confidentiality is turned on or off, to select which cryptographic algorithms should be used, etc.

3.2 Communication Support Services

This section presents the communication support services, which feature two main building blocks: the Randomized Intrusion-Tolerant Services, and the Overlay Protection Layer against DoS attacks. These services can be utilized for instance by the implementations of the activity support services, or by applications that need to have high levels resilience to accidental faults or malicious attacks.

3.2.1 Randomized Intrusion-Tolerant Services

With the increasing need of our society to deal with computer- and network-based attacks, the area of intrusion tolerance has been gaining momentum over the past few years. Arising from the intersection of two classical areas of computer science, fault tolerance and security, its objective is to guarantee the correct behavior of a system even if some of its components are compromised and controlled by an intelligent adversary.

A pivotal problem in fault- and intrusion-tolerant distributed systems is consensus. This problem has been specified in different ways, but basically it aims to ensure that n processes are able to propose some values and then all agree on one of these values. The relevance of consensus is considerable because it has been shown equivalent to several other distributed problems, such as state machine replication and atomic broadcast. Consensus, however, cannot be solved deterministically in asynchronous systems if a single process can crash (also known as the FLP impossibility result [38]). This is a significant result, in particular for intrusion-tolerant systems, because they usually assume an asynchronous model in order to avoid time dependencies. Time assumptions can often be broken, for example, with denial of service attacks.

Throughout the years, several techniques have been proposed to circumvent the FLP result, and among them randomization is particularly interesting because it requires no extra assumptions on the environment. *Randomized Intrusion-Tolerant Asynchronous Services* (RITAS) is a stack of randomized intrusion-tolerant protocols (also called Byzantine fault-tolerant protocols) for distributed systems. Figure 5.1 depicts the RITAS stack. It provides several services to applications who need to perform broadcasts and execute several flavors of consensus operations in a potentially malicious environment. All protocols in the stack rely on two standard Internet services that are abstracted by the MN: the IPsec Authentication Header protocol (AH) and the Transmission Control Protocol (TCP). These two protocols provide authenticated reliable communication channels for the rest of the stack.

The fundamental communication services that are offered by the stack are an *echo*, *reliable* and *atomic broadcast* among the members of the group of processes that implement the application. Based on these communication primitives, RITAS supports various kinds of agreement (or consensus) services. The most basic one is the *binary consensus*, which allows processes to agree on a single bit of data. There is also a *multi-valued consensus* service that can be employed to reach agreement on values of arbitrary length. Finally, the *vector consensus* service supports

agreements on vectors of values of arbitrary length.

3.2.1.1 Service Description and Interface

RITAS exports a simple API for applications who wish to access the protocols provided by the stack to build distributed systems services. The API revolves around the RITAS context data structure *ritas_t*, however, this data type is completely opaque to the application programmer. Functions provided by the API can be divided into two categories: context management and service requests. A typical RITAS session is composed by 4 basic steps executed by each process:

1. Initialize the RITAS context by calling *ritas_init()*.
2. Add the participating processes to the context by calling *ritas_proc_add_ipv4()*.
3. Call the communication and consensus protocols as many times as wished (however, functions are blocking and not thread-safe).
4. Destroy the RITAS context by calling *ritas_destroy()*.

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

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

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

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

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

```
int ritas_proc_add_ipv4(ritas_t *ctx, u_short id, u_char *ip, u_short port, u_char *key);
```

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

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

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

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

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

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

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

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

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

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

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

ritas_vc() runs vector consensus executions identified by *vcid*. The functions blocks until a decision is reached. The proposal value is passed as a pointer to a buffer *proposal* containing the value of length *prop_s*. The decision vector is stored in the buffer pointed by *vec*. The maximum length of data that this buffer can hold is indicated by *vec_s*. In case of success, the function returns the length of the decision vector in bytes; in case of failure returns -1. The decision vector can

be extracted into a data structure *ritas_vector_t* that makes it easier to process using the ancillary function *ritas_vector_extract()*.

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

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

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

ritas_ab_recv() delivers a message that was atomically broadcasted by some process in the group. The function blocks until a message is delivered. The message is stored in the buffer pointed by *buf*. The maximum length in bytes that the buffer can hold is indicated by *buf_s*. The function takes a pointer *abh* to a data structure *ritas_ab_header_t* where it is stored some meta-information about the delivered message such as its total order number. In case of success, the function returns the length of the message in bytes; otherwise it returns -1.

3.2.2 Overlay Protection Against Denial-of-Service Attacks

The Internet was designed for the minimal processing and best-effort forwarding of any packet, malicious or not. For attackers this architecture provides an uncontrolled network path to victims. DoS attacks exploit this to target mission-critical services. In recent years DoS attacks have become one of the most serious security threats to the Internet. This is because they may result in massive service disruptions and also because they have proven to be difficult to defend against. An estimate of worldwide DoS attack shows more than 12000 attacks on over 5000 distinct Internet hosts during a three week period in 2001 [62]. The 2004 CSI/FBI computer crime and security also shows DoS attacks are among most financially expensive security incidents [42].

Today, the Internet has become an emerging technology for remote control of industrial applications (e.g. power plants controllers) [21]. In such an environment, the communication path among application sites needs to be kept clear of interferences such as the ones created by DoS attacks: attacks that attempt to overwhelm the processing units or link capacity of the target site (or routers that are topologically close) by saturating with malicious packets.

Solving the network DoS problem is hard, given the fundamentally open nature of the Internet and the apparent reluctance of router vendors and network operators to deploy and operate new, potentially complex mechanisms. However, there are various approaches to solving DoS problems (see below). Though many DoS countermeasures have been proposed recently, it is not clear that any of them is able to stop Internet DoS attacks in the foreseeable future.

In this section, we address a protection layer for DoS attacks on top of today's existing IP infrastructure, where the communication is among application sites, located anywhere in the wide-area network, that have authorization to communicate with that location. This section focuses on how to design an Overlay Protection Layer (OPL) to solving DoS Problem.

Several researchers are exploring the use of overlay networks to tolerate DoS attacks [54, 84, 97]. The key idea is to hide application locations behind an overlay (proxy) network. Application sites can communicate with each other via the overlay network, where attackers cannot easily trace and locate the application sites to launch attacks. Location-hiding is an important component of a complete solution to DoS attacks. It gives application sites the capability to hide their IP addresses, thereby preventing DoS attacks, which depend on the knowledge of the victim's IP address. In general location-hiding schemes provide a "safety period" for application sites. In this section, we discuss how to design the overlay protection layer such that it is secure enough, given attackers who have a large but finite set of resources to perform the attacks. The attackers know the IP addresses of the nodes that participate in the overlay and also IP addresses of application sites. However, a few nodes have secret IP addresses in the OPL architecture.

We evaluate the OPL architecture performance by simulation and evaluate the likelihood that an attacker is able to prevent communication among application sites. Results show that even the attackers are able to launch massive attacks they are very unlikely to prevent successful communication. For example, in a static attack case (focused attack on a fixed set of nodes), DoS attacks completely are countered. In a dynamic attack case, if attackers can launch attacks upon 50% of nodes in the overlay, still 75% of communications among application sites are successful.

3.2.2.1 Background

This section presents a short overview of DoS problem, and describes some of the current defense techniques against it.

DoS Attacks

DoS attacks are a major security threat against availability in the Internet. In a DoS attack, attackers consume resources, on which either the applications or accesses to the applications depend, making the applications unavailable to the users. There are two classes of DoS attacks: application-level attacks and infrastructure-level attacks [97].

Application-level DoS attack: attackers attack through the application interface; for example, attackers overload an application by sending abusive workload, malicious requests which lead to application crash, extra CPU processing, system reboot, or general system slowing down. In this type, an attacker can also render a computing resource unavailable by modifying the system configuration (such as its static routing tables or password files). In fact attackers attack the system by exploiting weaknesses in the application software. This type of vulnerability typically originates in inadequate software assurance testing or negligent patching. Such DoS attacks are generally addressed through hardened security policies and authentication mechanisms. Application-level DoS attacks are application-specific and do not require the target application's IP address.

Infrastructure-level DoS attack: attackers directly attack the resource of the service infrastructure, such as the networks and hosts of the application service; for example, attackers send flood of bogus packets to saturate the target network. Infrastructure-level DoS attacks only require knowledge of application site addresses, i.e., IP addresses.

Distributed denial-of-service (DDoS) attacks are large scale DoS attacks which employ a large number of attackers distributed across the Internet. There are two steps in such attacks. First, attackers build large zombie networks by compromising many Internet hosts, and installing a zombie program on each. Second, attackers activate this large zombie network, directing them to DoS a target.

In the rest of this section we will focus on infrastructure-level DoS attacks. Therefore, whenever we say DoS attack, we refer to infrastructure-level DoS attack.

DoS Resilience Techniques

Figure 3.1 shows countermeasure techniques against the DoS problem. Countermeasure techniques are classified into reactive and proactive categories.

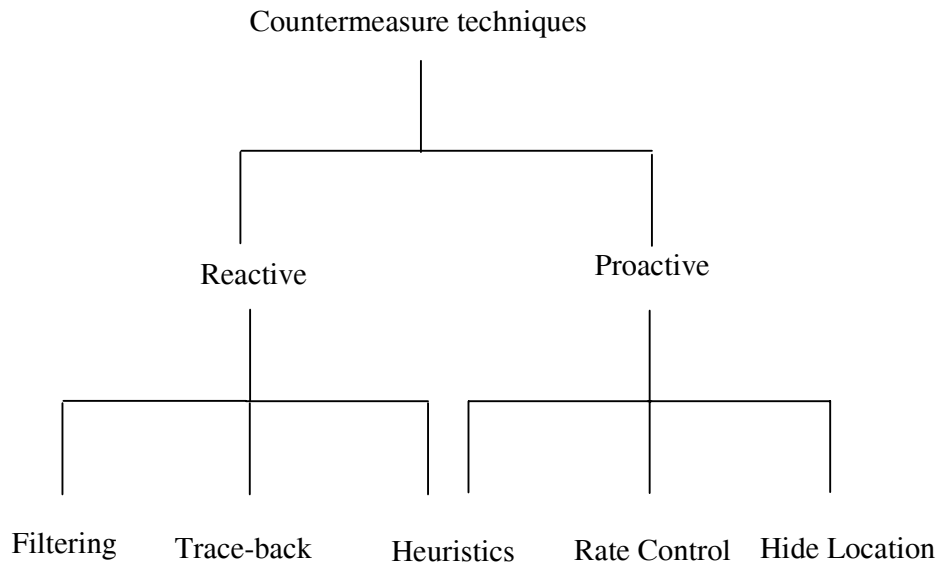


Figure 3.1: Countermeasure techniques against DoS attacks.

Reactive Techniques They monitor traffic at a target location, waiting for an attack to occur. After the attack is identified, typically via analysis of traffic patterns and packet headers, the countermeasure techniques are established (filtering techniques or source trace back techniques).

- **Filtering techniques:** in some cases the malicious packet flows can be identified on clearly-defined metrics, e.g., obviously wrong source address or other obvious errors in packet header. Such packet flows can be filtered at routers using some of the following techniques:
 - **Ingress filtering [36] :** routers check a packet for its source IP address, and block packets that come from an address beyond the routers' possible ingress address range. This requires a router to accumulate sufficient knowledge to distinguish between legitimate and illegitimate addresses, thus it is most feasible in customer networks or at the border of Internet Service Providers (ISP) where address ownership is relatively unambiguous.
 - **Distributed Packet Filtering (DPF) [70] :** it explores the power-law of Internet topology in source address validation. It can be distributed at core routers to proactively stop packet flows with obviously wrong source addresses, and meanwhile to reactively trace back the attacking sources. Empirical experiments show that DPF can efficiently identify spoofed address outside the autonomous system (AS) where the attackers reside.
 - **Source Address Validity Enforcement (SAVE) [57] :** in SAVE, symmetry between destination forwarding and source validation is explored to realize a protocol similar to Internet routing, but along the reverse direction for maintaining an incoming tree of authenticated sources. The protocol enables SAVE routers to filter malicious packet flows.

However, filtering techniques suffer from some problems:

1. Since filtering is rendered per-flow, routers must possess sufficient power to process a large number of flows simultaneously.
 2. The scalability concern in Internet core routers which could already be heavily loaded due to empirical experience obtained from per-flow based Internet.
 3. The accuracy with which legitimate traffic can be distinguished from the DoS traffic.
 4. The methods that filter traffic by known patterns or statistical anomalies in traffic patterns can be defeated by changing the attack pattern and masking the anomalies that are sought by the filter.
- **Source Traceback [77, 79]** : typically, traceback methods include packet-based marking, link testing, and verifying logging. Packet-based marking is normally comprised of two complementary components: a marking procedure executed by routers in the network and a path reconstruction procedure implemented by the victim. The routers augment IP packets with address marks en-route, then the victims can use information embedded in the IP packets to trace the attack back to the actual source. Instead of packet marking, an alternative method is to generate traceback information using separate IP control information such as link testing messages and verifiable logging messages.

However, the source trace-back defense techniques have some problems:

1. The trace-back mechanisms incur overhead in the form of control message processing, storage, and communication.
2. It is not quick and suffers a long delay to protect victim.
3. It does not address the DDoS problem.
4. Often the source of the attack is not real culprit but simply a node that has been remotely subverted by a cracker (zombie machine). The attacker can just start using another compromised node.

Passive Techniques These techniques do not require detection mechanisms. They prevent targets from DoS problem either by considering some controls in the network or transport layers (e.g., rate control techniques) or by using the interface layer (e.g., hide location techniques).

- **Rate Control Techniques [41, 99]** : in many cases, there is no clear boundary between DoS attack and insufficient service availability. Countermeasures based on rate control seek to enforce fairness in bandwidth allocation, thus minimize the damage caused by DoS attacks. In fact the idea of doing rate control is to identify the per-packet processing cost for different types of packets, and limited the flow rates such that the end server does not go into overload situation. Different rate limiting policies could be applied to different classes of traffic based on the resources they consume. Figure 3.2 shows rate control techniques from high level view.

However, the main problem of this approach is that its effectiveness is reduced when the number of attacking flows is large. Also if an attacker attacks with different types of packets, it cannot tolerate many attacks very well.

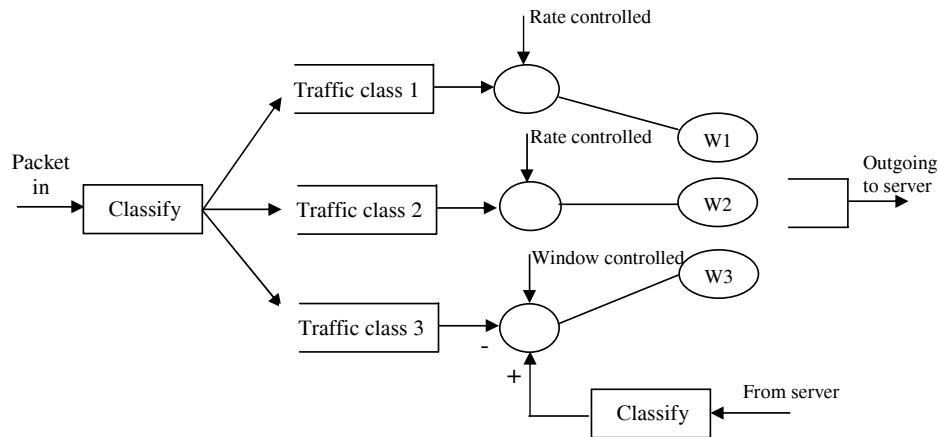


Figure 3.2: Rate control mechanism from high level view [41].

- **Location-Hiding Techniques** : location-hiding is an important component of a complete solution to DoS attacks. It gives application sites the capability to hide their locations and thereby preventing DoS attacks, which depend on the knowledge of their locations.

Overlay networks have been proposed as a means for location hiding. An overlay network is used to mediate all communications among application sites. As long as the mediation can be enforced, the overlay network is the only public interface for reaching an application site, and the application site cannot be directly attacked.

Many location-hiding mechanisms use overlay networks. As shown in [54, 84, 85, 97] it is feasible to hide an application's IP address using an overlay network, thereby enforcing overlay network mediation. Secure Overlay Services (SoS) [54] is an architecture that uses the overlay network for hiding locations. SoS uses filters combined by secret servlets to enforce all application access being mediated through the SoS network. In the SoS architecture, access requests will be authenticated by SOAP nodes and then routed via the Chord overlay network [86] to one of the beacon nodes and then to one of the servlets, which then forwards the requests to the target site which is protected via filters. Figure 3.3 shows the SoS architecture. However, the SoS architecture only simplifies the filtering roles around the target and reduces filtering processing time. Tolerating a large scale DoS attacks (e.g., DDoS) still is a challenge in the SoS architecture.

- **Heuristic Techniques** : there are some heuristics and creative approaches in both reactive and proactive areas against DoS attacks. Here we mention some of these approaches in brief.
 - **Replicated Servers** : one approach to mitigate DoS attacks against information carriers is to massively replicate the content being secured around entire network. To prevent access to the replicated information, an attacker must attack all replication points throughout the entire network- a task that is considerably more difficult than attacking a small number of, often co-located, servers. However, there are several reasons why replication is not always an ideal solution. For instance, if the information requires frequent updates, then it is hard to ensure large-scale coherency. Another concern is

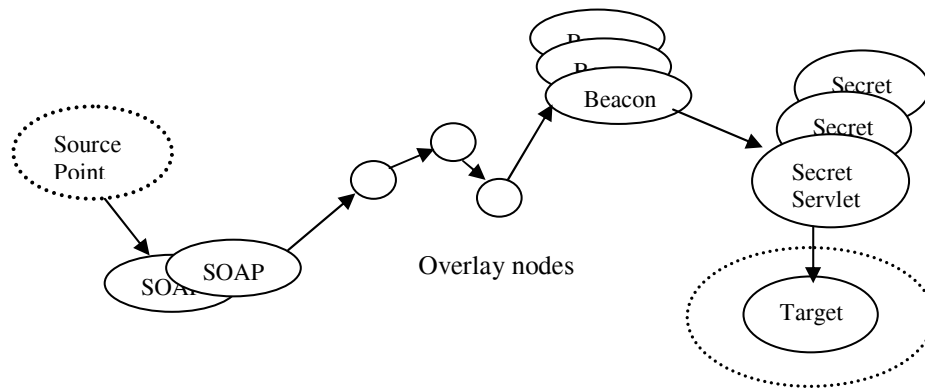


Figure 3.3: SoS architecture [54].

the security of the information. Also in applications such as power grid applications this approach is not suitable due to frequent changes in control operations.

- **AID Architecture [20]** : Chen and Chow describe global anti-DoS services, called AID, which ensures a registered client network the accessibility to any registered server as long as the client does not participate in the attack. The AID service is implemented as a distributed system consisting of a geographically dispersed AID station, for service registration and anti-DoS operations. When a server is under attack, it will signal its AID station, which propagates the information via the system to the other AID stations. Each AID station enforces all clients' traffic for the server into the IPsec tunnels.

In summary, each DoS countermeasure has its strengths and weaknesses. It is clear that none of the countermeasures will be the "silver bullet" that can stop DoS attacks immediately and efficiently. One solution can be combining the strengths of all effective solutions and let them compensate each other's weaknesses. Of course, this requires a general model to illustrate the shared features of all countermeasures.

3.2.2.2 The OPL Architecture and Operation Description

The OPL is a proactive approach to prevent DoS attacks. The goal of the OPL architecture is to make an intermediate interface by overlay nodes to hide the location of application sites during DoS attacks. OPL allows communication only among confirmed sites. It means that application sites have given each other a prior permission. Typically, this requires that any packet must be authenticated through the OPL architecture before the packet is allowed to be forwarded to the destination.

OPL is an overlay network, composed of nodes that communicate with one another atop the underlying network substrate. In overlay networks, nodes will perform routing functionality to deliver packets from one node in the overlay to another node of the overlay by a defined protocol. The set of overlay nodes of the OPL architecture is known to the public and also to the attackers.

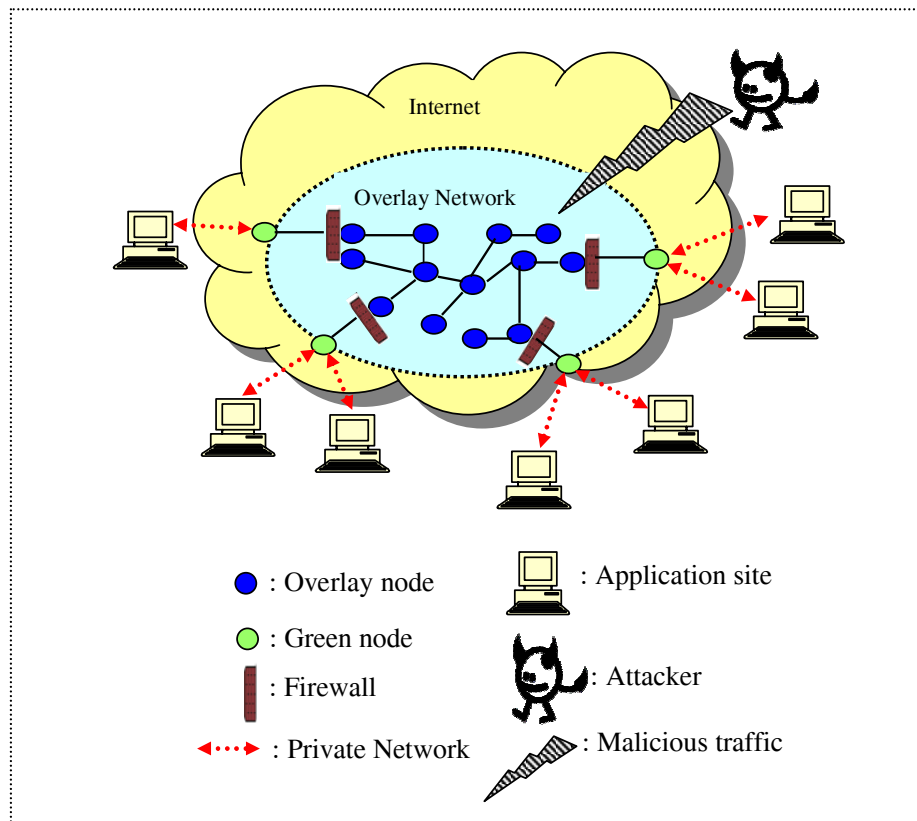


Figure 3.4: The OPL architecture.

However, certain nodes are kept secret from the public, that is certain roles that an overlay node may assume in the architecture are kept hidden. Attackers can launch DoS attacks from variety points of the Internet against application sites and overlay nodes. We assume that attackers cannot penetrate inside the overlay and so they cannot send malicious packets inside the overlay. Figure 3.4 shows a high-level overview of the OPL architecture that protects application sites from DoS attacks. In the following subsections the architecture is explained step by step.

Design Rationale

When there is no DoS attack, the application sites communicate normally via the Internet. In this case, the overlay is not used. In case of a DoS attack against an application site, the site switches to the overlay network. Fundamentally, the goal of the OPL architecture is to distinguish between authorized and unauthorized traffic. The former is allowed to reach the destination, while the latter is dropped. Hence, at a very basic level, we need the functionality of firewalls in the network that use authentication techniques to drop malicious packets. Authentication techniques can be solved by using traditional protocols such as IPsec, TLS, or by smart cards. Thereby attackers cannot penetrate inside the overlay; however they can launch DoS attacks against overlay nodes from variety points of the Internet. Although they can attack the overlay, the application sites are immune of attacks because the overlay is the initiated defense shield.

Any application site (e.g., application site "A") upon receiving a DoS attack disconnects itself from the Internet and signals its green node, which connects the application site "A" to the OPL system. Only the application site knows the location of green node. Hence the IP of green nodes is secret or the role of these nodes is kept hidden. The application site "A" signals its green node by a private network, while the green node communicates with overlay via the Internet. The application site "A" broadcasts a message via its green node to all other green nodes "connect to me via the overlay". Green nodes deliver this message to their application sites. Now, any application site that wants to communicate with the application site "A" connects to it via the overlay, while communicating with other (healthy) application sites via the Internet.

Green nodes These nodes are unknown nodes in the OPL architecture for the public in order to hide the location of application sites. The role of green nodes is kept secret and hence nobody knows which nodes are green nodes. The overlay network, along with the green nodes, holds the location of application sites secret. In fact green nodes are reference points in the architecture. If the green node of application site "A" is disclosed to attackers, the architecture cannot protect the application site "A" from DoS attacks any more; although it can protect other application sites.

A green node, upon receiving a packet from the overlay, forwards it to the destination via a private network. As the communication between green nodes and application sites is done by a private network, this part of the communication is completely secure. Although green nodes are connected to the overlay via the Internet, this part is secure too due to the secret location of green nodes.

A remark may be made that attackers can guess the routing path and then disclose the location of green nodes. The answer is no, because some overlay networks such as Chord, or CAN [72] have complicated routing mechanisms that will route packets to the destination efficiently, while utilizing a minimal amount of information about the identity of that destination. Overlay networks have dynamic nature and high level of connectivity. In these networks unlike the underlying network an edge is allowed between any pair of overlay nodes, hence overlay networks have flexibility and several choices to select a route, which complicates the job of attackers to determine the path taken within the overlay to a secret green node.

For having cheap private networks in any geographically zone a few green nodes are selected.

Attacking the Overlay Attackers can attack simultaneously the overlay from a variety of points of the Internet. However, these attacks have no influences on application sites. Overlay networks can tolerate these attacks due to their dynamic nature and high level of connectivity. Since a path exists between every pair of nodes, it is easy to recover from a breach in communication that is the result of an attack that shuts down a subset of overlay nodes. The recovery action simply removes those "shut down" nodes and then the overlay reconfigures itself (update hashing and routing tables). Furthermore, no overlay node is more important or sensitive than others.

If the overlay node that connects a green node to the overlay is attacked, it exits simply

from the overlay and the green node randomly connects to another node of the overlay.

Some Additional Points

The OPL architecture utilizes the Chord network as an overlay network [86]. The Chord network is a distributed protocol with N homogeneous overlay nodes that uses consistent hashing for routing. It maps an arbitrary identifier to a unique destination node that is an active member of the overlay by a hash function. Each overlay node maintains a list that contains $O(\log N)$ identities of other active nodes in the overlay.

Chord network has some valuable properties:

- In a Chord network, to find a key T from any node require $O(\log N)$ steps. In fact a Chord node only needs a small amount of routing information about other nodes.
- A Chord network never partitions. It means that if attackers attack the Chord network simultaneously and bring down many nodes, Chord easily reconfigures itself without partitioning. In fact Chord is able to route effectively even if only one node remains in the overlay.
- A Chord network is probably the most dependable network when compared to other overlay networks (see Section 5.2.2 for a discussion).

Application sites are connected to the overlay via their green nodes. In fact green nodes are bidirectional nodes that are used both for connecting to the overlay and delivering messages to the application sites.

3.2.2.3 Performance Evaluation

In this section we analyze the performance of the OPL architecture by simulation. Our evaluation determines the probability of successful searches and vice versa the probability of a successful attack. A search is successful if a path between any two arbitrary application sites can be found. Some assumptions:

- Attackers know the set of overlay nodes and can attack them.
- Attackers have a bounded and fixed amount of bandwidth to attack the architecture. For instance, attackers can attack maximum X nodes ($X < N$, N is the total number of overlay nodes) simultaneously.
- Attackers do not know which nodes are green nodes.
- Attackers cannot penetrate inside the overlay due to strong authentication techniques and firewalls.

- Each application site can access the overlay through access control techniques (via firewalls)
- Each geographical zone has two green nodes.

We created a simulator to evaluate the OPL architecture. To do this, we implement the Chord (overlay) network since it is the backbone of the OPL architecture.

- Implementing the Chord network: We have implemented the Chord network in an iterative style [86]. In the iterative style, a node resolving a lookup initiates all communications: it asks a series of nodes for information from their hashing tables, each time moving closer on the Chord ring to the desired successor. During each stabilization protocol step, a node updates its immediate successor and one other entry in its successor list or hashing table. Thus if a node's successor list and hashing table contain a total of k unique entries, each entry is refreshed once every k stabilization rounds. When the predecessor of node m changes, m notifies its old predecessor q about the new predecessor q' . This allows q to set its successor to q' without waiting for the next stabilization round.

The delay of each packet is exponentially distributed with an average of 50 ms. If a node m cannot contact another node m' within 500 ms, m concludes that m' has left or has been attacked (we consider same action for both leaving node and attacked node although we suppose different rates for these). If m' is an entry in m 's successor list or hashing table, this entry is removed. Otherwise m informs the node from which it learns that m' is gone.

- Implementing the attack toolkit: Attackers are placed outside the overlay. To implement an attack toolkit, we programmed in C++ the basic structure of Trinoo [30] to generate both DoS and DDoS attacks. In fact we implemented two basic procedures for the attack toolkit: daemon and master procedures. We have several daemon procedures that are controlled by a master procedure. Daemon procedures simply send malicious traffic to the targets at the given start time that is determined by the master procedure. We consider both static and dynamic attacks. In a static attack, attackers select a fixed set of overlay nodes to attack and when an attacked node is removed from the overlay, the attacker cannot redirect to another node. In the dynamic approach, attackers can attack any node and also can redirect attacks to other nodes.

In the first experiment we assume that the location of green nodes is never disclosed. In this case, if at least one node of overlay is alive, the probability of successful search is one. When all overlay nodes (100%) are dead, the probability of successful search is zero. This property of the architecture is true due to the second property of Chord. Note that Chord will be able to route effectively even if only one node remains in the overlay. In this case we can measure how important the threat is. Figure 3.5 plots threat against the OPL architecture for different values of N , the total number nodes of overlay system.

An obvious result of this figure is that resilience against DoS attacks grows linearly with overlay network size. In other words, a large overlay network provides better DoS resilience.

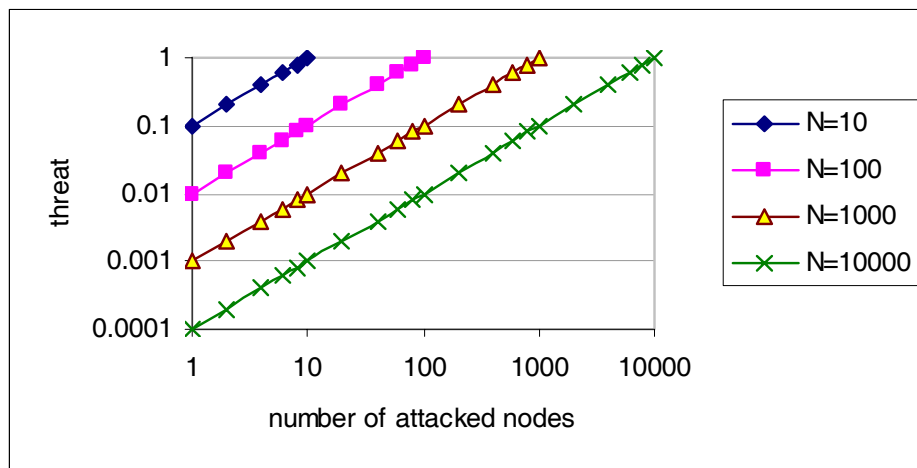


Figure 3.5: Threat against the OPL architecture.

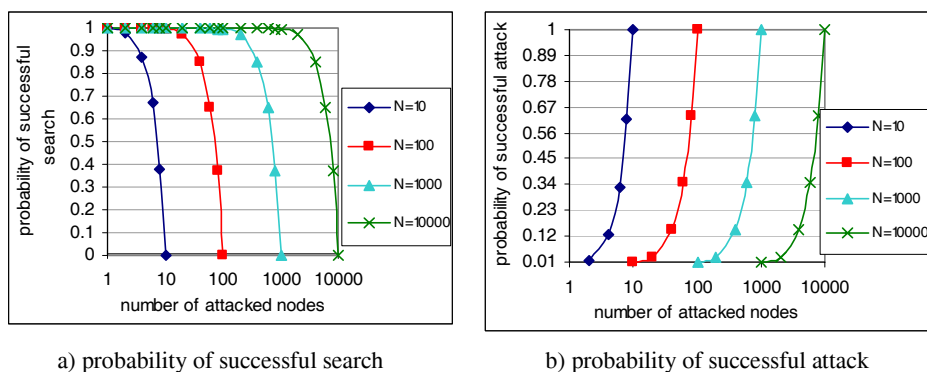


Figure 3.6: DoS attack in the static case.

In the next experiments we suppose that the location of green nodes may be disclosed randomly. Attackers attack the overlay nodes randomly and maybe some of these attacked nodes are green nodes.

The second experiment analyzes the likelihood of successful searches in the static case. Figure 3.6(a and b) shows the probability of a successful search and probability of successful attacks when the number of attacked nodes varies along the x-axis, respectively.

From these figures, we see that the likelihood of an attack successfully terminating communication between any two arbitrary application sites is negligible unless the attacker can simultaneously bring down a significant fraction of nodes in the network. For instance, if an attacker attacks 10% of overlay nodes simultaneously we have around 99% successful searches. If attackers bring down between 20% and 40% of overlay nodes, around 90% successful searches exist. Even if attackers bring down half of total nodes (50%), there are still 75% successful searches.

The third set of experiments evaluates the OPL architecture in a dynamic case. Previous experiment assumed that an attacker would select a fixed set of nodes to attack, and that OPL takes no action towards repairing the attack. The scenario of this set of experiments is that when OPL identifies an attacked node, that node is removed from the overlay. When an attacker identifies

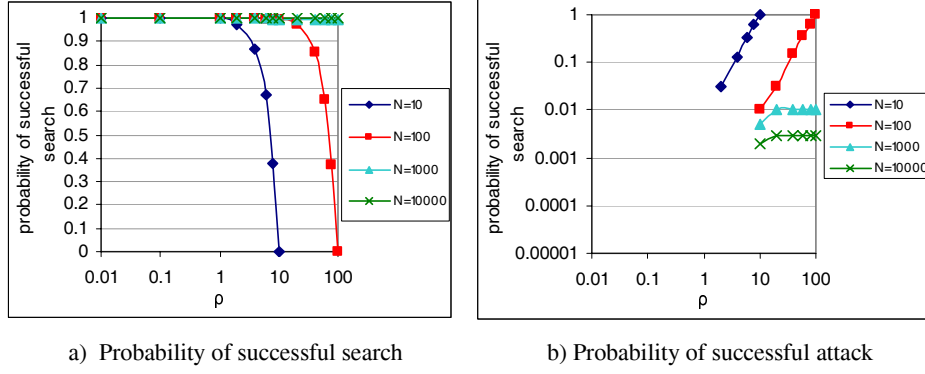


Figure 3.7: DoS attack in the dynamic case.

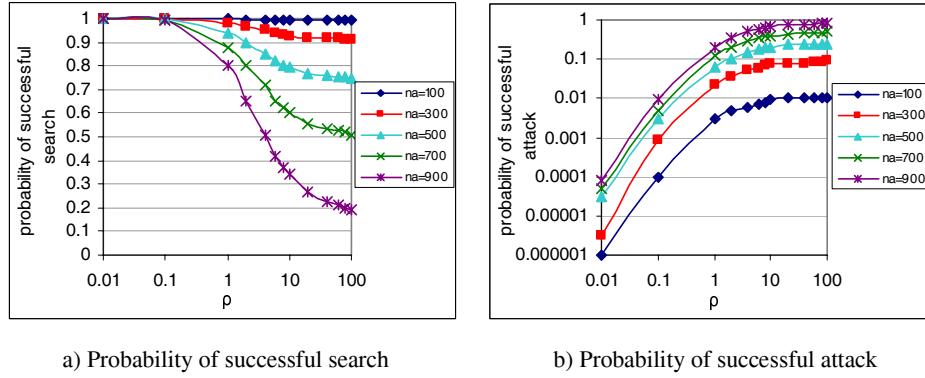


Figure 3.8: DDoS attack in the dynamic case.

that a node it is attacking no longer resides in the overlay, it redirects its attack towards a node that does still reside in the overlay. When the attacked node is removed, the attack against that node terminates and the node comes back to the overlay after D_r delay. D_r is a repair delay for reconfiguration. Also, there is an attack delay, D_a , that equals the difference in time between when an attacked node is removed from the overlay to the time when the attacker (realizing the node it is attacking has been removed) redirects the attack towards a new node in the overlay. We assume both D_a and D_r are exponentially distributed random variables with respective rates λ and μ .

We evaluate the OPL architecture in the dynamic case for both DoS and DDoS attacks. Figure 3.7 (a and b) plots the probability of successful searches and successful attacks respectively for DoS attack, where $\rho = \lambda/\mu$ varies along x-axis.

When $\rho \leq 1$ for any value of overlay nodes (N), attackers are least likely (0%) to deny the service. Also for large value of N (e.g. $N = 1000$) attackers do not have chance to deny the service. Even when ρ is 10 and N is 100, still we have about 100% successful search. We think in practice ρ is always less than 10. As a result, a DoS attack in the OPL architecture with repair is solved nearly completely.

Figure 3.8 (a and b) plots the probability of successful searches and successful attackers respectively for DDoS attack in the dynamic case. In this figure n_a is the maximum ability of attackers that can attack simultaneously. In figure, the simulation is done for $N = 1000$.

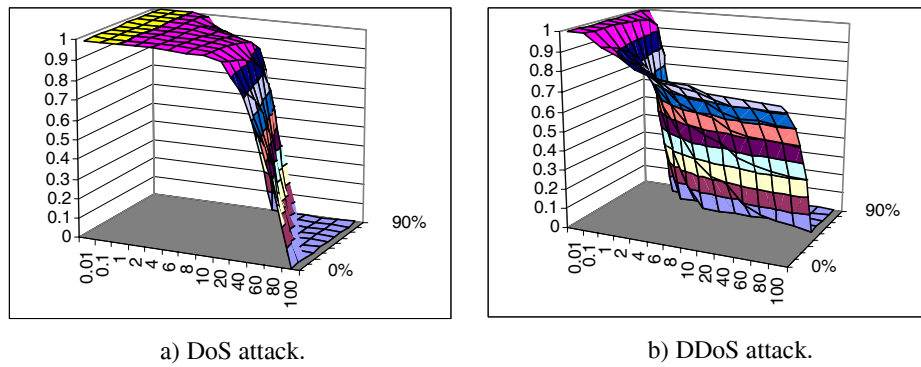


Figure 3.9: Impact of node joining/leaving for a) DoS and b) DDoS attacks. (X-axis: ρ ; Y-axis: percentage of absent nodes; Z-axis: probability of a successful search)

This figure shows that when an attack is distributed (DDoS), the fraction of time for which the attack is successful can be significant when a large fraction of nodes in the overlay is attacked, even when $\rho \leq 1$. From this figure we can understand that although DDoS attack is harder to tolerate than DoS attack, for $na \leq N/2$ it can be tolerated. For instance, when $na \leq N/2$ and $\rho \leq 10$ the probability of successful search is more than 70%. An interesting note of Figure 3.8 is that when ρ is increased above 10, the curves reach steady state and remain constant. It means that if ρ is increased to more than 100, the probability of successful attack will not change or change unnoticeably.

The fourth set of experiment evaluates the impact of node joining/leaving on the OPL architecture. Node joining/leaving is a normal action in overlay networks, especially in a Chord network. Surely this action has an impact on the OPL architecture against DoS attacks. It is clear that when more overlay nodes are absent, the resilience against DoS attacks will be worse and vice versa. Figure 3.9 (a and b) plots a 3-D view of node joining/leaving impact on OPL for both DoS and DDoS respectively, where the x-axis shows $\rho = \lambda/\mu$, the y-axis shows the average percentage of absent nodes of the overlay at any time and the z-axis shows probability of successful search.

As one can see from the figure, node joining/leaving has a significant impact on the OPL performance when it is under DDoS attack. In fact the probability of successful search degrades significantly when the percentage of absent nodes is increased in the overlay, while in the DoS attack it is barely degraded. For instance, the probability of successful search is more than 0.9 when $\rho \leq 1$ for any value of absent nodes, while for DDoS attack it is more than 0.9 only when $\rho \leq 0.01$.

The fifth set of experiments is used to analyze the impact of the number of green nodes on the OPL architecture. In Figure 3.10, we hold N fixed at 1000. We vary the number of green nodes along the x-axis and again plot the probability of successful search on the y-axis for different values of na . From this figure we see that likelihood of an attack successfully terminating communication between application sites is decreased by increasing the number of green nodes in each zone.

The sixth experiment (Figure 3.11) shows that although a large overlay provides a better resilience against a DoS attack, it requires more delay for routing too. In fact the delay is increased

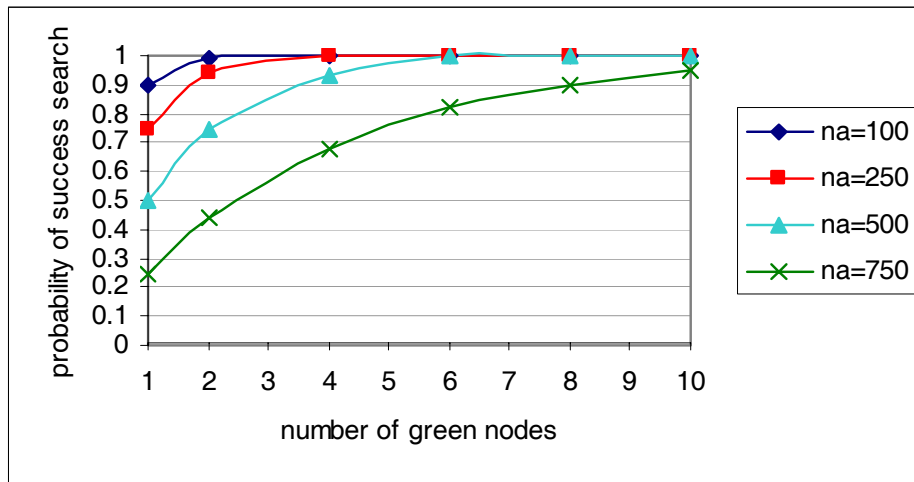


Figure 3.10: Impact of the number of green nodes per application on the OPL performance.

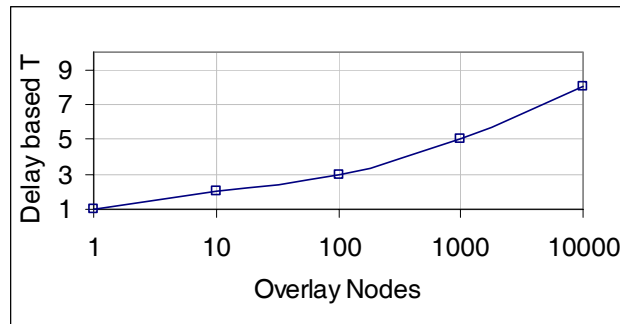


Figure 3.11: Delay of the overlay network.

proportionally to $O(\log N)$, where N is the total number of nodes in the overlay network (Chord property).

3.3 Activity Support Services

The Activity Support Services currently defined comprise the CIS Protection service, and the Access Control and Authorization service. In the next year, further services might be provided, namely the CIS Communication service.

3.3.1 CIS Protection Service

The CRUTIAL reference architecture to protect critical infrastructures [34, 91] models the whole infrastructure architecture as a WAN-of-LANs. This topology allows simple solutions to hard problems such as legacy control subnetworks, and interconnection of critical and non-critical traffic. Typically, a critical information infrastructure is formed by facilities, like power transformation substations or corporate offices, modeled as collections of LANs and interconnected by a wider-area network, modeled as a WAN, in the WAN-of-LANs model.

This architecture allows defining realms with different levels of trustworthiness. In this section we are interested in the problem of protecting realms from one another, i.e., a LAN from another LAN or from the WAN. However, given the ease of defining LANs in today's IP architectures (e.g., through virtual switched LANs), there is virtually no restriction to the level of granularity of protection domains, which can go down to a single host. In consequence, our model and architecture allow us to deal both with outsider threats (protecting a facility from the Internet) and insider threats (protecting a critical host from other hosts in the same physical facility, by locating them in different LANs).

Protection of LANs from the WAN or other LANs is made by a device called the CIS. A CIS provides two basic services: the *Protection Service (PS)* and the *Communication Service (CS)*. The PS ensures that the incoming and outgoing traffic in/out of the LAN satisfies the security policy of the infrastructure. The CS supports secure communication between CIS and, ultimately, between LANs. Although the CIS supports these two services, this section presents only the protection service, not the communication service (that is currently under development). Therefore, from now on “the CIS” means “the CIS *Protection Service*”. Figure 3.12 illustrates the use of CIS protecting several LANs of a critical infrastructure.

A CIS can not be a simple firewall since that would put the infrastructure at most at the level of security of current Internet systems, which is not acceptable since intrusions in those systems are constantly being reported [43, 51]. Instead, a CIS is a distributed protection device based on a sophisticated access control model and designed with intrusion-tolerant capabilities.

3.3.1.1 Specification of the Service

At the applications point of view, the CIS works mainly like a firewall, and thus is completely transparent to the critical infrastructure applications, that does not even have know that

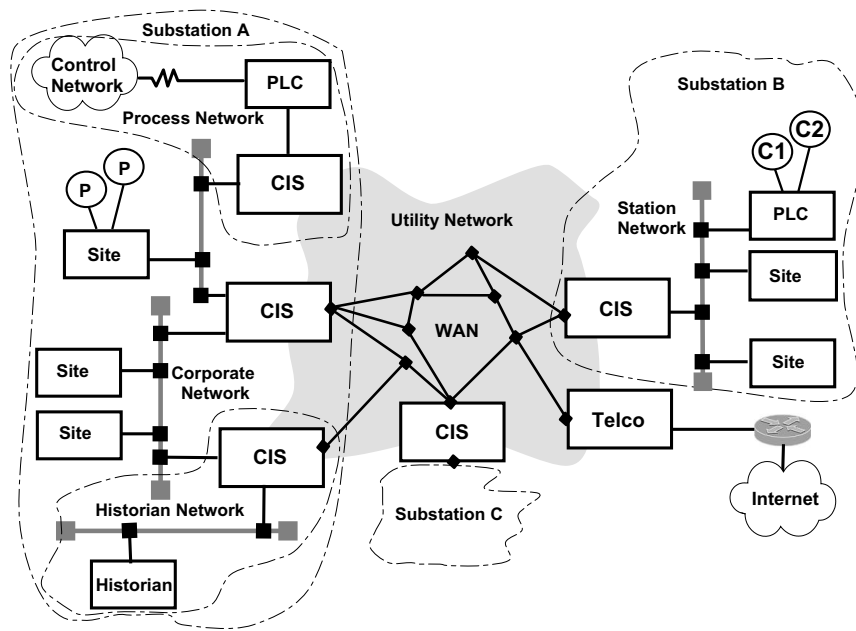


Figure 3.12: WAN-of-LANs connected by CIS.

there is some kind of sophisticated protection device inspecting their communication. The CIS captures packets, checks if they satisfy the security policy being enforced, and forwards the approved packets, discarding those that do not satisfy the policy. However, several other characteristics of the CIS make it a unique protection device.

- Distributed firewall :** CIS can be used in a distributed way, enforcing the same policies in different points of the network. An extreme case in the SCADA/PCS side is to have a CIS in each gateway interconnecting each substation network, and a CIS specifically protecting each critical component of the SCADA/PCS network. The concept is akin to using firewalls to protect hosts instead of only network borders [8], and is specially useful for critical information infrastructures given their complexity and criticality, with many routes into the control network that can not be easily closed (e.g., Internet, dial-up modems, VPNs, wireless access points) [14].
- Application-level firewall :** Critical infrastructures have many legacy components that were designed without security in mind, and thus do not employ security mechanisms like access control and cryptography [33]. Since these security mechanisms are not part of the SCADA/PCS protocols and systems, which *must still be protected*, protection must be deployed in some point between the infrastructure and the hosts that access it. The CIS has to inspect and evaluate the messages considering application-level semantics because, as already said, the application (infrastructure) itself does not verify it.
- Rich access control model :** Besides the capacity to inspect application-level messages, the CIS needs to support a rich access control policy that takes into account the multi-organizational nature of the critical infrastructures as well as their different operational states. Taking the Power System as an example, there are several companies involved in

generation, transmission and distribution of energy, as well as regulatory agencies, and several of these parties can execute operations in the power grid. Moreover, almost all Power System operation is based on a classical state model of the grid [37]. In each state of this model, specific actions must be taken (e.g., actions defined in a defense plan, to avoid or recover from a power outage) and many of these actions are not allowed in other states (e.g., a generator can not be separated automatically when the Grid is in its normal state). These two complex facets of access control in critical infrastructures require more elaborated models than basic discretionary, mandatory, or role-based access control. To deal with this, in the architecture of CRUTIAL we adopt a more elaborated model, OrBAC (Organization Based Access Control) [4]. It allows the specification of security policies containing permissions, prohibitions, obligations and recommendations, taking into account the role of the subject, who is part of an organization, the action it wants to execute, the target object of this action, and the context in which it is executed. An example: “In context ‘emergency’, operators from company C can execute maintenance operations on device D.”

- **Intrusion-tolerant firewall :** The level of security of current systems connected to the Internet is not adequate for the infrastructures we are concerned with, given their criticality. To improve the security and dependability of the CIS, it is designed to be intrusion-tolerant [90]: it is replicated in n machines and follows its specification as long as at most f of these machines are attacked and have their behavior corrupted. Obviously, such intrusion tolerance is only useful if there is independence in the way machines are corrupted. This independence of the corruptions or intrusions requires that the machines do not share the same vulnerabilities, since an intrusion is always the result of an attack that activates a vulnerability (or more). The usually accepted way to enforce this property is by having diversity in the machines [58, 68]. Therefore, the intrusion-tolerant CIS is designed with diversity in mind.

In the current stage of the project, we addressed mainly the problem of designing an intrusion-tolerant distributed firewall and its required protocols, which are described in Section 5.3.1.

3.3.2 Access Control and Authorization Service

Poly-organisation based access control (PolyOrBAC) handles the collaboration between the CII organizations, while controlling whether the interactions between these organizations comply with their expected behaviour. In fact, each component (organization/subsystem) of the CII could have its own security objectives and should be able to cooperate with the other components. Each organization contains its own resources, services, applications, operating system, objectives, functioning and security rules, and policy. Each organization could specify its security policy according to organization-based access control model (OrBAC) [49]. Different organizations could have common interests, common access, common policy, etc.

As organizations are interconnected through CIS, we believe that, in order to provide a controlled cooperation adapted to CII, each CIS regroups mechanisms to define security policy of systems that compose each LAN (local and collaboration policies), and it also regroups mechanisms for collaboration: to make these LANs capable of collaboration and offering services to each other.

PolyOrBAC uses OrBAC to specify local (as well as remote) access control policies (for each organization) as well as (collaboration) rules implying several organizations. In this way, the same rule, for example *Permission(organization,role,activity,view,context)*, could concern several internal and external accesses. On the other hand, Web Services (WS) are used to enforce collaboration.

In the example below (see Figure 3.13), organization B offers WS1, and organization A is interested in using WS1. This example shows the mutual negotiation of access rules for distant services. The organization B offers the Web service WS1, and Alice from organization "A" wishes to invoke WS1 from organization B. Since all organizations of the CII are connected by CIS (represented by Web interfaces), each CIS must contain mechanisms to implement the local security policy (of the organization that owns the CIS), collaboration mechanisms thanks to Web services should also be implemented (so that the various organizations can offer services and collaborate with each other). These policies and mechanisms must allow the authorized accesses to the resources and prevent the unauthorized accesses (accidental or malicious ones).

3.3.2.1 A Practical Description of WP1 Scenarios

In this section, we will describe the access control activity support services. In deliverable D4 [66], we proposed PolyOrBAC, a framework that ensures an access control specification for involved entities/organisations in a CII. PolyOrBAC handles the collaboration between the CII organisations, while controlling whether the interactions between these organizations comply with their expected behaviour [50]. In fact, each component or subsystem of the CII has its own security policy and should respect the global security policy and should be able to cooperate with the other components. For this aim, on the one hand PolyOrBAC uses the OrBAC to specify local security policies, on the other hand uses web services to handle the collaboration and resources sharing

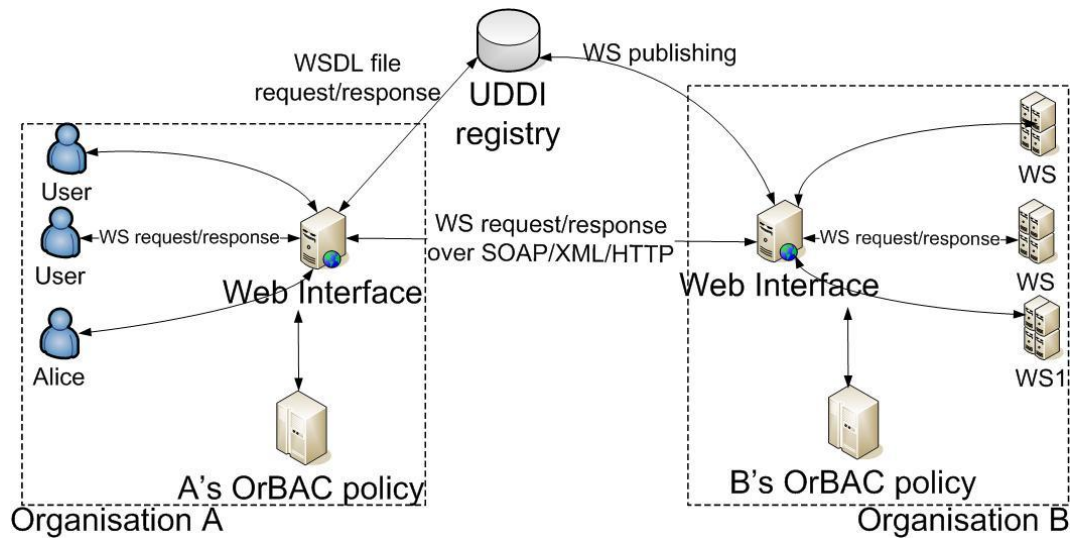


Figure 3.13: The general functioning.

processes.

This vision is important in the context of CRUTIAL where some components of the electrical and information infrastructure execute remote actions and access to resources from other partner organizations. The WP1 D2 deliverable illustrates this kind of accesses by presenting several functioning scenarios occurring in emergency and normal situations (cf. WP1-D2 Section 5.2: control system scenarios) [34].

In this Section we present the final-user vision by deriving the web services [94, 93, 95, 67] corresponding to these scenarios, while in Section 5.1 we use these Web Services to present PolyOrBAC access control protocols and thus, to enforce secure communications between CII.

Firstly, we study the scenario 2 (interaction between TSO and DSO operators under emergency conditions). This scenario involves two different operators the TSO (Transmission system operator) and the DSO (Distribution system operator), Then, we will apply the same methodology over scenarios 1, 3 and 4 of section 5 of D2.

3.3.2.2 Scenario 1: DSO Tele-operation (Emergency and Normal Conditions)

This scenario considers the possible cascading effects of ICT threats to the DSO communication channels among Area Control Centres and their tele-controlled Substations in presence of Power Contingencies, e.g. insufficient production or HV line unavailability (cf. Section 5.1 p77 D2).

This scenario considers the information flow between DSO Area Control Centers and their supervised HV/MV Substations, in both NORMAL and EMERGENCY Distribution Grid operating conditions.

A. Involved organisations:

We consider 2 important classes of organisations involved in the scenario 1:

- Area Control Centre (ACC) managed by the DSO.
- Substation (Contains an MCD-TU) at the level of the ACC.

B. Exchanged commands and signals:

Figure 3.14 shows the different exchanges of signals and commands between DSO ACC and DSO SS organisations.

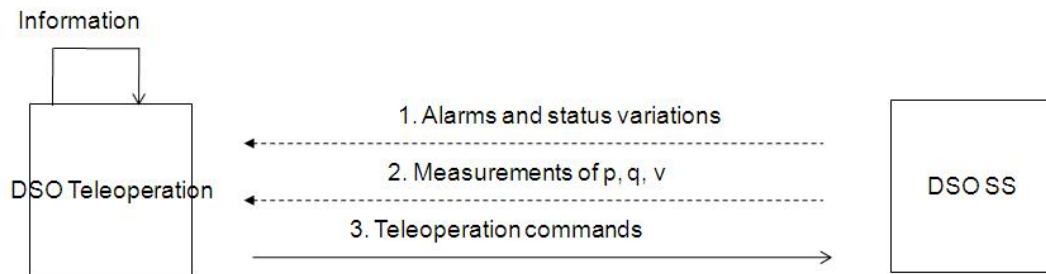


Figure 3.14: Scenario 1 exchanged commands.

1. DSO substations send measurements of P, Q, V and breakers positions towards DSO ACC.
2. DSO substations send alarms and status variations towards DSO ACC.
3. DSO ACC send commands to DSO substations.
4. DSO ACC updates information locally in order to align the databases and the operation capabilities at each ACC.

C. Web Services to implement:

In this scenario, we specify the Tele-operation web service as follows:

- o Provider: DSO SS.
- o Client: DSO ACC.

3.3.2.3 Scenario 2: Interaction Between TSO and DSO Operators in Emergency Conditions

This scenario considers the possible cascading effects due to ICT threats to the communication (security of the communications) channel among TSO/DSO Control Centers and MCD-TUs in emergency conditions (under-frequency or voltage instability). It is assumed that in emergency conditions the TSO is authorised by the DSO to activate defence plan actions consisting in the performance of load shedding activities on the Distribution Grid (cf. WP1-D2 Section 5.2 p88 D2). TSO RCC monitors the Electric Power System and elaborates some potentially emergency conditions that could be remedied with opportune load shedding commands applied to particular areas of the Grid. In order to actuate the defence action the TSO RCC chooses a subset of HV/MV Substations (SSs) from the list of SSs participating to the emergency plan, then sends the requests of preventively arming the MCD-TUs in these SSs to the interested DSO Area Control Centres (DSO ACCs). These requests are delivered through a communication channel between a TSO RCC and a DSO ACC. The DSO ACC provides for arming the required SSs, and returns their status to the TSO RCC. In case of detection of a real emergency situation the TSO MCD-TU (area sentinel) sends the command of load shedding to all the DSO MCD-TU participating to the emergency plan, but only the armed ones will be actually detached.

A. Involved Organizations:

By studying this scenario, we distinguish four important classes of organisations ¹ (cf. Figure 5-14 p89 D2) involved in the scenario 2:

- Regional Control Centre (RCC) managed by the TSO.
- Substation (Contains an MCD-TU) at the level of the RCC.
- Area Control Centre (ACC) managed by the DSO.
- Substation (SS) (Contains an MCD-TU) at the level of the ACC.

B. Exchanged signals and commands:

Figure 3.15 shows the different exchanges of signals and commands between TSO RCC, TSO SS, DSO ACC and DSO SS organisations.

1. All DSO SSs send different signals and measurements (power, voltage, frequency, etc) to their reference ACC.
2. All TSO SSs and Power Stations send different signals and measurements (power, voltage, frequency, etc) to their reference RCC.

¹An organisation can represent a server, a machine, an ACC, an RCC, an NCC, etc.

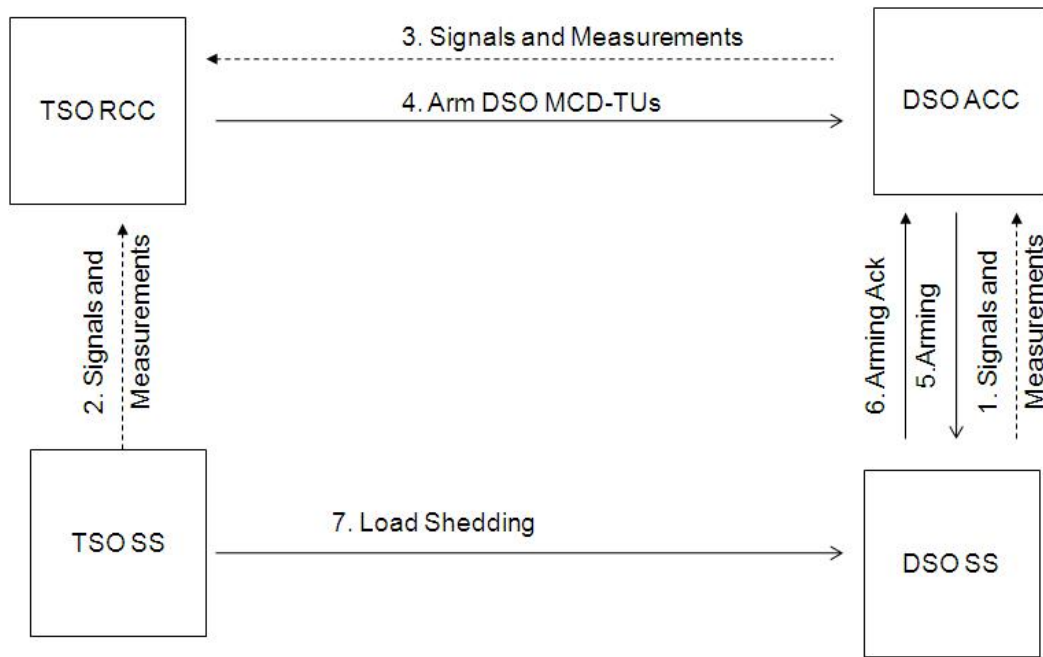


Figure 3.15: Scenario 2 exchanged commands.

3. DSO ACC sends different signals and measurements to TSO RCC.
4. The RCC elaborates the situation and the possible emergency manoeuvres then sends the requests of arming a set of SSs to DSO ACCs.

Arming Step:

5. The TSO RCC orders DSO ACCs to arm the SSs concerned by the possible emergency situation.
6. The DSO ACC sends the arming request to the involved SSs which arm their MCD-TU.
7. The DSO SSs send an acknowledgement to DSO ACC.

Then the TSO SSs send Load shedding request to the armed DSO SSs.

Load shedding step:

8. The TSO SS orders all the DSO SSs to perform the load shedding command over their MCD-TUs. Only the previously armed DSO SSs perform the load shedding over their MCD-TUs.

C. Web Services to implement:

In this scenario, 3 web services can be distinguished:

Arming Request:

- o Provider: DSO ACC.
- o Client: an EMS at the level of the TSO RCC.

Arming Activation:

- o Provider: DSO SS
- o Client: a virtual user at the level of the DSO ACC.

Load Shedding Activation:

- o Provider: DSO SS
- o Client: a virtual user at the level of the TSO SS.

3.3.2.4 Scenario 3: Integration of DSO Operation and Maintenance Functions

This scenario explores the integration of process control and corporate networks in a DSO Control Centre, evaluating the possible cascading effects of cyber attacks on the integrated architecture (cf. Section 5.3 p90 D2). Note that this scenario is still under development and there will be possible evolutions in the future. Scenario 3 assumes the use of a DSO intranet to access data for:

- Maintenance of MV and LV grids. The maintenance manager needs to know both the status of the Grid and the availability of the maintenance personnel, in order to carry out maintenance plans.
- Access to process information by other Corporate functions (administrative, management, metering, etc.)

A. Involved organisations:

We consider 2 important classes of organisations involved in the scenario 3:

- Area Control Centre (ACC) managed by the DSO.
- Substation (Contains an MCD-TU) at the level of the ACC.

B. Exchanged commands and signals:

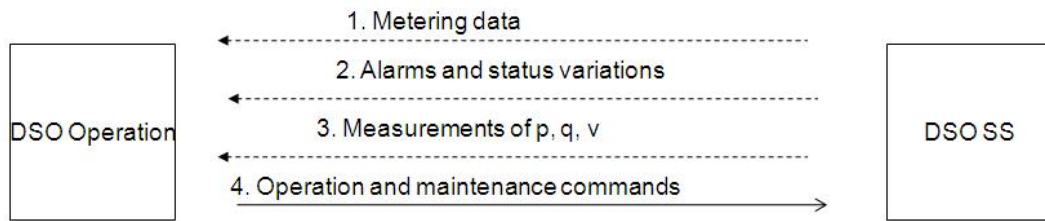


Figure 3.16: Scenario 3 exchanged commands.

Figure 3.16 shows the different exchanges of signals and commands between DSO ACC and DSO SS organisations.

1. DSO substations send metering data towards DSO ACC.
2. DSO substations send alarms and status variations towards DSO ACC.
3. DSO substations send measurements of P, Q, V and breakers positions towards DSO ACC.
4. DSO ACC or MCD-TUs send operation commands to DSO substations.

C. Web Services to implement:

In this scenario, we specify the Operation web service as follows:

- o Provider: DSO SS.
- o Client: DSO ACC.

3.3.2.5 Scenario 4: Maintenance of ICT Components of DSO

This scenario considers the institution of a DSO Centralized ICT maintenance service (cf. Section 5.4 p92 D2). Note that this scenario is still under development and there will be possible evolutions in the future. Scenario 4 assumes the presence of a central Control Center for the monitoring and control of the components of all the ICT systems of the Power Utility, a kind of Security Operational Center (SOC) having both Teleoperation and ICT competencies for providing:

1. Remote ordinary maintenance activities on the ICT components in a substation, including communication devices (e.g. routers, gateways, firewalls), Station Computers, Station level and bay level IEDs.
2. Continuous monitoring of the ICT equipment status, including security monitoring functions.

3. Repair actions on ICT network and automation equipment configurations.

A. Involved organisations:

We should distinguish between the DSO ACC which ensures the control and the monitoring, and the entity (organisation DSO- maintenance) that ensures the maintenance of all equipments. The maintenance concerns all equipments (from the ICT infrastructure), belonging to the ACC or the DSO substations.

We consider 2 important organisations involved in the scenario 4:

- The DSO-maintenance from the Area Control Centre (ACC) managed by the DSO.
- A substation (Contains an MCDTU) at the level of the ACC.

B. Exchanged commands and signals:

Figure 3.17 shows the different exchanges of signals and commands between DSO Maintenance and DSO SS organisations.

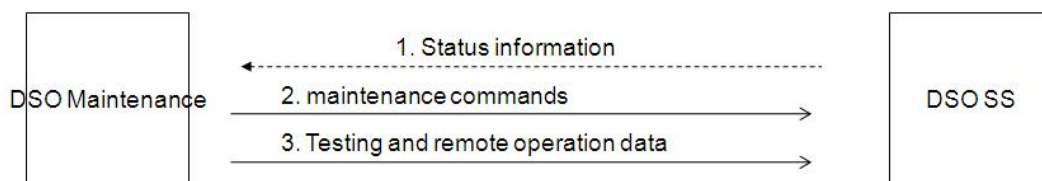


Figure 3.17: Scenario 4 exchanged commands.

1. DSO substations (ICT components of DSO) transmit status information to DSO ACC.
2. The DSO ACC delivers commands, and
3. Functional testing and remote operation data to ICT components of the DSO at the level of the DSO substations.

C. Web Services to implement:

In this scenario, we specify the Maintenance web service as follows:

- o Provider: DSO SS.
- o Client: DSO ACC.

3.4 *Monitoring and Failure Detection*

This section provides a preliminary definition of the middleware services devoted to monitoring and failure detection activities. Following a top-down approach, a preliminary schematization and description of the relevant activities is presented.

3.4.1 **Diagnosis Framework**

From a theoretical viewpoint, the framework for the diagnosis activity involves the following three actors [25]: the monitored component (the system component under diagnosis), the deviation detection mechanism (the entity introduced in the system to observe the external behavior of the monitored component and to judge whether it is suitable or not) and the state diagnosis mechanism (it has to guess the internal state of the monitored component, based on information collected overtime from the deviation detection mechanism). Two information flows are necessary: the first involves the monitored component and the deviation detection mechanism (the latter has to observe the former), the second involves the deviation detection mechanism and the state diagnosis mechanism (the latter has to collect information provided by the former). Each of the above information flows could be managed following a proactive or a reactive schema: in the proactive schema, the entity interested in fresh information has to ask for them, whilst in the reactive schema the entity that generates information has to send it to the entity interested in it. More interaction patterns can be found in [75].

When dealing with systems composed by several components at different architectural levels, the above framework needs to be extended: several components need to be monitored and several deviation detection mechanisms need to be in place. Given that evident or subtle dependencies can exist among the component functionalities, errors observed in different components could be correlated. All this requires the need for a correlation activity between the collection of errors and the declaration of a diagnosis.

In fact, in large, well designed and tested systems, with hardware components far from their wear out age, crude faults or malfunctions, easily and rapidly recognizable as such, tend to be really rare events; while subtle borderline conditions may still occur, whose very presence is difficult to detect, not to mention accurate recognition and treatment. Another difficulty stems from the observation that, in the less-than-simple systems, a binary (faulty/not_faulty, go/no_go) schematization of the error behavior is insufficient and possibly counter-effective on the availability. More often, in a mature system non-fatal malfunctions occur, which nevertheless require corrective action, albeit far from downing the entire affected (sub)system. Also from this point of view, then, finer recognition of borderline situations is needed, to enable flexible reaction. As an example, an over-temperature signal given by a CPU sensor may trigger a de-scheduling of low priority tasks, if the CPU is fully loaded, while it may be a symptom of worse problems if the CPU is only lightly loaded. Even if a binary decision scheme is deemed good enough, an approach more knowledgeable than just waiting for a single, unambiguous error signal would have a positive impact on the system dependability.

The same idea can be extended when dealing with a complex infrastructure made up by several sub-systems (nodes). In this case it is not practical to have a centralized diagnostic entity that has to gather, aggregate and correlate all the detections collected over time from the sub-systems; the centralized diagnostic entity should be ultra-reliable and communication links between it and all the monitored sub-systems should be guaranteed. Therefore, methods for distributed diagnosis are mandatory, where each sub-system decides independently about the system (e.g. which are the healthy sub-systems and which the faulty ones).

Considering a distributed system comprised by completely connected sub-systems, the Hybrid Fault-Effect Model [96] can be assumed, so that all fault classification is based on a local classification of fault-effects (to the extent permitted by the deviation detection mechanism of the sub-system itself) and on a global classification, thus developing a global opinion on the fault-effect. Diagnosis is thus performed using a two-phase approach on a concurrent, on-line and continual basis:

1. Local detection and diagnosis, aiming to diagnose the sub-system itself.
2. Global information collection and global diagnosis, obtained through exchange of local diagnosis. Since each sub-system may have a different perception of the errors observed on the remote sub-systems, each node has some private values (the results of private diagnosis on remote sub-systems) and the goal is to ensure consistent information exchange and agreement against Byzantine behavior. An agreement (or consensus) algorithm is thus needed in order to solve the problem.

3.4.2 Diagnosis in CRUTIAL

The CRUTIAL infrastructure is organized as a WAN-of-LANs, where each LAN is connected to the WAN by a CIS. Given that the computers inside the LANs cannot be modified/updated, all the diagnosis activity has to be performed inside the CIS. The following diagnosis scenarios arise:

- CIS self-diagnosis (local view): CIS monitors both itself (e.g. to diagnose hardware or software faults) and its LAN (e.g. to “measure” its level of trustworthiness).
- CIS distributed diagnosis (global view): CISs construct a common view about the “state” of a certain CIS in the infrastructure (e.g. related to the liveness and trustworthiness of a specific CIS).

3.4.2.1 The CIS Self-Diagnosis Service

From a local viewpoint the CIS is a sophisticated application level firewall (combined with equally sophisticated intrusion detectors) which is required to:

1. be intrusion tolerant,
2. prevent resource exhaustion providing perpetual operation,
3. be resilient against fault assumption coverage uncertainty providing survivability.

In order to comply with the above requirements, the CIS has a hybrid architecture and is replicated (with diversity) in n replicas. Each CIS replica is built using a synchronous and secure local wormhole and an asynchronous and insecure payload.

Two monitoring/failure detection scenarios arise:

1. Internal monitoring: monitoring performed inside a single replica, trying to detect local failures;
2. External monitoring: monitoring performed on the perceived behavior of the other replicas.

The internal monitoring, given the information system malfunctions introduced in [66], has to be done on the following components/services (so far, components/services that need to be monitored were not definitely identified):

- Hardware components (e.g. network interfaces, processing units, memory modules ...) which are supporting the replica. The monitoring activity on these components makes sense only when physical replication is used; in case of logical replication, these components need to be monitored in the host system running the replicas.
- Software components belonging to several architectural levels in the payload or in the operating system.

Several signals coming from many architectural levels are collected and processed over time: an example of signal coming from low architectural levels (OS) is related to a CPU fan that is working too slow or a temperature sensor that is signaling the CPU is too warm. An example of signal coming from a higher architectural level is an application-generated exceptions or error return code.

The internal monitoring activity has hence to identify compound system conditions which could require diverse corrective actions; for example, repeated application errors could be interpreted as manifestation of software aging requiring rejuvenation, or could be correlated with lower level signals (the CPU is too warm because the CPU fan is working too slow), requiring another kind of reconfiguration (e.g. replacing the CPU fan). The rationale behind internal monitoring and failure detection is to try to stop the replica before it starts to behave incorrectly.

The external monitoring is performed by each replica on the perceived behavior of the other replicas, given that a replica is not guaranteed to always behave correctly. The monitoring activity is performed at service level, so that each service is in charge of detecting whether its peers running in the other replicas seems correct or not. An examples of middleware service monitoring its peers on other replicas is the “Protection Service”.

3.4.2.2 LAN Diagnosis Service

The CIS monitors over time the nodes in its protected LAN in order to evaluate their trustworthiness. The evaluated trustworthiness level is used to request maintenance actions on the protected un-trustable node (e.g. replacing hardware, refreshing the software, changing passwords, ...).

A trustworthiness indicator for each protected node N is defined (it could be multi-dimensional) and modified based on the following detections:

- The instance of the security policy applied within the CIS itself to the outgoing traffic detects that N is trying to violate the security policy (e.g. trying to send something without being allowed to do it).
- the instance of the security policy running on a remote CIS detects that a message sent by N to one of its protected node was rejected. The CIS distinguishes whether an incoming packet really comes from a station computer (instead from an hacker in the WAN) using the “LAN Traffic Labeling” service (the CIS protecting the source node signs the label). The signed label is hence a proof of the source of the packet.

The LAN diagnosis service collects over time the above detections in order to evaluate the trustworthiness indicator of each protected node. If protected trustworthiness indicator of node N goes over a given threshold, the LAN diagnosis service alerts its peers about N being un-trustable (so that they can possibly take adequate countermeasures).

3.4.2.3 CIS Distributed Diagnosis Service

The several replicas that made up a single CIS are required to perform the same operations; this simplifies somewhat the task of checking their correctness on the run. Each single CIS, as seen from the WAN, is a different logical entity, in terms of actions, services and requests toward other CISs. In the ordinary information flux there is no simple comparison rule check that can be performed, to catch on the fly a mischievous partner. On the other hand, if a CIS becomes compromised, internal redundancy and resilient architecture notwithstanding, then necessarily the basic hypothesis on the fault occurrence has been broken: more than f replicas are out of order together. Of course, this is the catastrophic case, whose probability has to be lowered down to a target level by choosing proper redundancy figures. However, a local catastrophe (regarding a single LAN controlled by a compromised CIS) not necessarily should imply the downing of the entire system. In fact, on the WAN side, all CISs attempt to maintain a common view of two parametric descriptors its partners' health: Liveness and Trustworthiness.

Liveness is checked in two ways: i) passively, by monitoring normal network traffic from the target; ii) if the former is not frequent enough, exert a form of resilient ping, by means of a simple challenge/response protocol.

Trustworthiness is built up by checking the formal correctness of the messages coming from the target, as well from any access violation detected by the Protection Service.

4 Runtime Support Protocols

4.1 Proactive-Reactive Recovery Protocols

In this section we describe the protocols used to implement the proactive-reactive recovery service presented in Section 2.1. Before describing the protocols, we start by presenting the assumptions under which they were developed.

4.1.1 System Model

The proactive-reactive recovery protocols are executed inside the proactive-reactive recovery wormhole (PRRW). Note that, as explained in Section 2.1, the PRRW executes in a synchronous (real-time) subsystem and it is assumed the existence of a set of services:

1. local PRRWs' clocks have a known precision, obtained by a clock synchronization protocol;
2. there is point-to-point timed reliable communication between every pair of local wormholes;
3. there is a timed reliable broadcast primitive with bounded maximum transmission time;
4. there is a timed atomic broadcast primitive with bounded maximum transmission time.

In this context, the proactive-reactive recovery protocols are implemented as threads in a real-time environment with a *preemptive scheduler* where static priorities are defined from 1 to 3 (priority 1 being the highest). The protocols algorithms do not consider explicitly the clock skew and drift, since we assume that these deviations are small due to the periodic clock synchronization, and thus are compensated in the protocol parameters (i.e., in the time bounds for the execution of certain operations).

4.1.2 The Protocols

We developed two protocols to implement the proactive-reactive recovery service. The protocols are presented in Algorithm 1 and Algorithm 2. The first protocol is responsible by the main logic of the service, while the second one is specifically responsible by recovery subslot allocation according to what was explained in Section 2.1.3. We start with the description of Algorithm 1 and then Algorithm 2 is described.

Parameters and variables. Algorithm 1 uses six parameters: i , n , f , k , T_P , and T_D . The id of the local wormhole is represented by i ; n specifies the total number of replicas and consequently the total number of local wormholes; f defines the maximum number of faulty replicas; k specifies the maximum number of replicas that recover at the same time; T_P defines the maximum time

interval between consecutive triggers of the *recovery* procedure (depicted in Figure 2.3 - page 13); and T_D defines the worst case execution time of the recovery of a replica. Additionally, four variables are defined: t_{next} stores the instant when the next periodic recovery should be triggered by local wormhole i ; the *Detect* set contains the processes that detected the failure of replica i ; the *Suspect* set contains the processes that suspect replica i of being failed; and *scheduled* indicates if a reactive recovery is scheduled for replica i .

Algorithm 1 Proactive-reactive recovery service: main protocol

{Parameters}	{Periodic recovery thread with priority 1}
integer i {Id of the local wormhole}	procedure <i>proactive_recovery</i> ()
integer n {Total number of replicas}	5: <i>synchronize_global_clock</i> ()
integer f {Maximum number of faulty replicas}	6: $t_{next} \leftarrow global_clock() + (\lceil \frac{i-1}{k} \rceil T_{slot} + \lceil \frac{f}{k} \rceil T_D)$
integer k {Max. replicas that recover at the same time}	7: loop
integer T_P {Periodic recovery period}	8: wait until $global_clock() = t_{next}$
integer T_D {Recovery duration time}	9: <i>recovery</i> ()
{Constants}	10: $t_{next} = t_{next} + T_P$
integer $T_{slot} \triangleq (\lceil \frac{f}{k} \rceil + 1)T_D$ {Slot duration time}	11: end loop
{Variables}	procedure <i>recovery</i> ()
integer $t_{next} = 0$ {Instant of the next periodic recovery start}	12: <i>recovery_actions</i> ()
set <i>Detect</i> = \emptyset {Processes that detected me as failed}	13: <i>Detect</i> $\leftarrow \emptyset$
set <i>Suspect</i> = \emptyset {Processes suspecting me of being failed}	14: <i>Suspect</i> $\leftarrow \emptyset$
bool <i>scheduled</i> = <i>false</i> {Indicates if a reactive recovery is scheduled for me}	15: <i>scheduled</i> $\leftarrow false$
{Reactive recovery interface threads with priority 3}	{Reactive recovery execution threads with priority 2}
service $W_suspect(j)$	upon $ Detect \geq f + 1$
1: <i>send</i> ($j, \langle SUSPECT \rangle$)	16: <i>recovery</i> ()
service $W_detect(j)$	upon $(Detect < f + 1) \wedge (Suspect \cup Detect \geq f + 1)$
2: <i>send</i> ($j, \langle DETECT \rangle$)	17: if $\neg scheduled$ then
upon <i>receive</i> ($j, \langle SUSPECT \rangle$)	18: <i>scheduled</i> $\leftarrow true$
3: <i>Suspect</i> $\leftarrow Suspect \cup \{j\}$	19: $\langle s, ss \rangle \leftarrow allocate_subslot()$
upon <i>receive</i> ($j, \langle DETECT \rangle$)	20: if $s \neq \lceil \frac{i}{k} \rceil$ then
4: <i>Detect</i> $\leftarrow Detect \cup \{j\}$	21: wait until $global_clock() \bmod T_P = sT_{slot} + ssT_D$
	22: if $ Suspect \cup Detect \geq f + 1$ then
	<i>recovery</i> ()
	23: end if
	24: end if

Reactive recovery service interface. $W_suspect(j)$ and $W_detect(j)$ send, respectively, a SUSPECT or DETECT message to wormhole j , which is the wormhole in the suspected/detected node (lines 1-2). When a local wormhole i receives such a message from wormhole j , j is inserted in the *Suspect* or *Detect* set according to the type of the message (lines 3-4). The content of these sets may trigger a recovery procedure.

Proactive recovery. The *proactive_recovery*() procedure is triggered by each local wormhole i at boot time (lines 5-11). It starts by calling a routine that synchronizes the clocks of the local wormholes with the goal of creating a virtual global clock, and blocks until all local wormholes call it and can start at the same time. When all local wormholes are ready to start, the virtual

global clock is initialized at (global) time instant 0 (line 5). The primitive *global_clock()* returns the current value of the (virtual) global clock. After the initial synchronization, the variable t_{next} is initialized (line 6) in a way that local wormholes trigger periodic recoveries in groups of up to k replicas according to their id order, and the first periodic recovery triggered by every local wormhole is finished within T_P from the initial synchronization. After this initialization, the procedure enters an infinite loop where a periodic recovery is triggered within T_P from the last triggering (lines 7-11). The *recovery()* procedure (lines 12-15) starts by calling the abstract function *recovery_actions()* (line 12) that should be implemented according to the logic of the system using the PRRW. Typically, a recovery starts by saving the state of the local replica if it exists, then the payload operating system (OS) is shutdown and its code is restored from some read-only medium, and finally the OS is booted, bringing the replica to a supposedly correct state. The last three lines of the *recovery()* procedure re-initialize some variables because the replica should now be correct (lines 13-15).

Reactive recovery. Reactive recoveries can be triggered in two ways: (1) if the local wormhole i receives at least $f + 1$ DETECT messages, then recovery is initiated immediately because replica i is accounted as one of the f faulty replicas (line 16); (2) otherwise, if $f + 1$ DETECT or SUSPECT messages arrive, then replica i is at best suspected of being failed by one correct replica. In both cases, the $f + 1$ bound ensures that at least one correct replica detected a problem with replica i . In the suspect scenario, recovery does not have to be started immediately because the replica might not be failed. Instead, if no reactive recovery is already scheduled (line 17), the aperiodic task finds the closest slot where the replica can be recovered without endangering the availability of the replicated system. The idea is to allocate one of the (reactive) recovery subslots depicted in Figure 2.3. This is done through the function *allocate_subslot()* (line 19 – explained later). Notice that if the calculated subslot $\langle s, ss \rangle$ is located in the slot where the replica will be proactively recovered, i.e., if $s = \lceil \frac{i}{k} \rceil$, then the replica does not need to be reactively recovered (line 20). If this is not the case, then local wormhole i waits for the allocated subslot and then recovers the corresponding replica (lines 21-22). Notice that the expression *global_clock() mod T_P* returns the time elapsed since the beginning of the current period, i.e., the position of the current global time instant in terms of the time diagram presented in Figure 2.3 - page 13.

Recovery subslot allocation. Subslot management is based on accessing a data structure replicated in all wormholes through a timed total order broadcast protocol, as described in Algorithm 2. This algorithm uses one more parameter and one more variable besides the ones defined in Algorithm 1. The parameter T_Δ specifies the upper-bound on the delivery time of a message sent through the synchronous control network connecting all the local wormholes. Variable *Subslot* is a table that stores the number of replicas (up to k) scheduled to recover at each subslot of a recovery slot, i.e., *Subslot* $[\langle s, ss \rangle]$ gives the number of processes using subslot ss of slot s (for a maximum of k). This variable is used to keep the subslot occupation, allowing the local wormholes to find the next available slot when it is necessary to recover a suspected replica.

A subslot is allocated by local wormhole i through the invocation of *allocate_subslot()*, which timestamps and sends an ALLOC message using total order multicast (line 1) to all local wormholes and waits until this message is received (line 2). At this point, *local_allocate_subslot()* is called and the next available subslot is allocated to the replica (line 3). The combination of total order multicast with the sending timestamp T_{send} ensures that all local wormholes allo-

Algorithm 2 Proactive-reactive recovery service: slot allocation protocol.

{Parameters (besides the ones defined in Algorithm 1)}

integer T_Δ {Bound on message delivery time}

{Variables (besides the ones defined in Algorithm 1)}

table $Subslot[\langle 1, 1 \rangle \dots \langle \lceil \frac{n}{k} \rceil, \lceil \frac{f}{k} \rceil \rangle] = 0$ {Number of processes scheduled to recover at each subslot of a recovery slot}

procedure $allocate_subslot()$

1: $TO_multicast(\langle ALLOC, i, global_clock() \rangle)$

2: **wait until** $TO_receive(\langle ALLOC, i, t_{send} \rangle)$

3: **return** $local_allocate_subslot(t_{send})$

upon $TO_receive(\langle ALLOC, j, t_{send} \rangle) \wedge j \neq i$

4: $local_allocate_subslot(t_{send})$

procedure $local_allocate_subslot(t_{send})$

5: $t_{round} \leftarrow (t_{send} + T_\Delta) \bmod T_P$

6: $curr_subslot \leftarrow \lfloor \frac{t_{round}}{T_{slot}} \rfloor + 1, \lfloor t_{round} \bmod T_{slot} \rfloor + 1$

7: **loop**

8: $curr_subslot \leftarrow next_subslot(curr_subslot)$

9: **if** $Subslot[curr_subslot] < k$ **then**

10: $Subslot[curr_subslot] \leftarrow Subslot[curr_subslot] + 1$

11: **return** $curr_subslot$

12: **end if**

13: **end loop**

procedure $next_subslot(\langle s, ss \rangle)$

14: **if** $ss < \lceil \frac{f}{k} \rceil$ **then**

15: $ss \leftarrow ss + 1$

16: **else if** $s < \lceil \frac{n}{k} \rceil$ **then**

17: $ss \leftarrow 0; s \leftarrow s + 1$

18: **else**

19: $ss \leftarrow 0; s \leftarrow 0$

20: **end if**

21: **return** $\langle s, ss \rangle$

upon $(t_{round} \leftarrow (global_clock() \bmod T_P) \bmod T_{slot} = 0$

22: **if** $\lfloor \frac{t_{round}}{T_{slot}} \rfloor = 0$ **then**

23: $prev_slot \leftarrow \lceil \frac{n}{k} \rceil$

24: **else**

25: $prev_slot \leftarrow \lfloor \frac{t_{round}}{T_{slot}} \rfloor$

26: **end if**

27: $\forall p, Subslot[\langle prev_slot, p \rangle] \leftarrow 0$

cate the same subslots in the same order. The local allocation algorithm is implemented by the $local_allocate_subslot(T_{send})$ function (lines 5-13). This function manages the various recovery subslots and assigns them to the replicas that request to be recovered. It starts by calculating the first subslot that may be used for a recovery according to the latest global time instant when the ALLOC message may be received by any local wormhole (lines 5-6), then it searches and allocates the next available subslot, i.e., a slot in the future that has less than k recoveries already scheduled (lines 7-21). Finally, in the beginning of each recovery slot, all the subslots of the previous recovery slot are deallocated (lines 22-27).

5 Middleware Services Protocols

The Middleware Services Protocols chapter presents the protocols that are being used to implement our intrusion-tolerant middleware. They include the standard protocols which are abstracted by the Multipoint Network, and the protocols specifically designed for the Communication and Activity Support, and Monitoring and Failure Detection.

5.1 *Multipoint Network*

The services provided by the Multipoint Network layer were already described in Section 3.1. The basic service is the provision of secure channels, which support the secure communication among two peers. These channels essentially enforce a set of attributes of the communication. The attributes that are relevant from a security point of view, are the following: authenticity, integrity, confidentiality, and reliability. In what follows we describe each of these attributes, how they are enforced and by which protocol (e.g., IP, IPsec, TCP or SSL/TLS).

Suppose a secure channel connects two peers P and Q . Authenticity is the measure in which what Q receives was sent by P . Authenticity means protection from forgery and is obtained by authentication mechanisms. There are two relevant kinds of authentication for this layer: unilateral - P authenticates Q or vice-versa, but not both; bilateral - P authenticates Q and vice-versa. When a secure channel is established, authentication can be done using a passwords or a public-private key pair. For instance, in SSL/TLS, one of the peers (unilateral authentication), or both (bilateral authentication), sends to the other a certificate signed by a Certification Authority with its public key, plus some data signed with its private key. Using this information the other peer, say Q , can verify if P is who it says it is. During the communication, the messages have also to be authenticated. An obvious solution is for P to sign all the messages it sends to Q with its private key. However, public-key cryptography is usually slow, so messages are typically authenticated using a keyed-Hash Message Authentication Code, or HMAC, like HMAC-SHA1 [55]. To use this scheme, during the establishment of the connection between P and Q , a shared key is established (for instance, P generates the key and sends it encrypted with Q 's public key, so only Q can decrypt it). Then each message takes an HMAC, which is basically a cryptographic hash of the message concatenated with the shared key. The message can not be forged without the possession of the key.

Integrity means that a message sent by P can not be modified without Q detecting this modification. Integrity is usually enforced using an HMAC, just like described. Both IPsec and SSL/TLS use this scheme.

Confidentiality means that only Q can read the messages sent by P on the channel that connects both. Confidentiality can be enforced by encrypting the communication using a symmetric cryptography algorithm, like AES or IDEA. For this to be possible, P and Q need to share a key, which is often obtained during the connection establishment, as described above.

Reliability means that a message sent by P is eventually delivered to Q . This attribute is

related to integrity since modifications to messages have to be detected to ensure that the message delivered is the message sent. However, detection is not enough, there is also the need to retransmit the message when the message is discarded, either because it was corrupted or because it was lost in the network. After having a scheme to enforce the integrity of the communication, the problem of ensuring the reliability in a system prone to security problems is identical to the problem of ensuring the reliability when only accidental corruptions or losses occur. The most common solution is based on the idea of Q confirming the messages it receives from P , and P resending the messages for which it did not receive a confirmation after a certain timeout. Reliability is provided by SSL/TLS, which in fact are modifications of TCP that ensures reliability in systems prone only to accidental faults. IPsec does not provide reliability, but TCP over IPsec does.

5.2 Communication Support

This section presents the protocols developed to implement the RITAS services and the overlay network protection.

5.2.1 Randomized Intrusion-Tolerant Protocols

The RITAS protocol stack, depicted in Figure 5.1, provides a set of useful distributed system services. The stack is organized in the following way: At the bottom there are two broadcast primitives: *echo broadcast* and *reliable broadcast*. On top of the broadcast primitives is the most basic flavor of consensus: *binary consensus*. This is the only randomized protocol in the stack. On top of binary consensus there is the *multi-valued consensus* protocol, which allows the proposal of arbitrary values. Finally, at the top of the stack there are two protocols: *vector consensus*, and *atomic broadcast*. Each one of these protocols is thoroughly described in the next subsections.

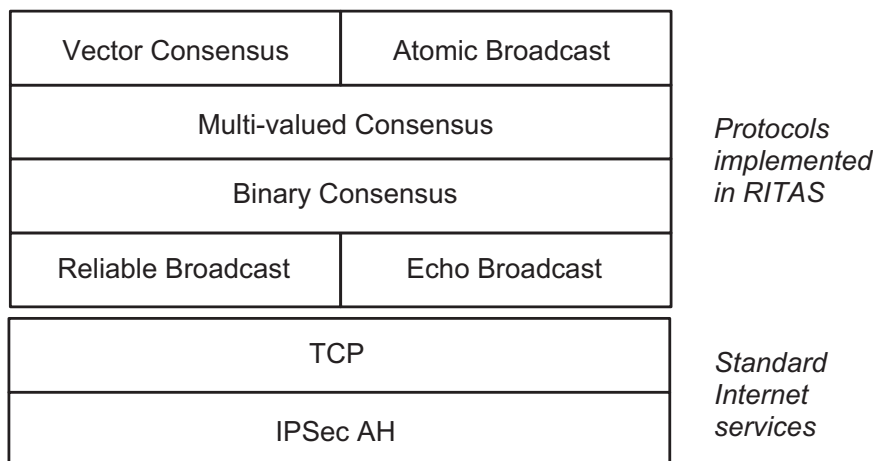


Figure 5.1: The RITAS protocol stack.

5.2.1.1 System Model

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

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

number of n processes.

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

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

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

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

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

5.2.1.2 Reliable Channels

All protocols in the stack rely on a authenticated reliable channel primitive, which can be implemented by the two layers at the bottom of Figure 5.1. These layers correspond to two standard Internet protocols that are abstracted by the MN (not shown in the figure): the IPsec AH protocol and TCP.

The channel primitive provides a point-to-point communication channel between a pair of correct processes with two properties: reliability and integrity. Reliability means that messages are eventually received, and integrity says that messages are not modified in the channel.

Formally, such a channel follows two properties:

- RC1 Reliability : If processes p_i and p_j are correct and p_i sends a message m to p_j , then p_j eventually receives m .
- RC2 Integrity : If p_i and p_j are correct and p_j receives a message m with $sender(m) = p_i$, then m was sent by p_i and m was not modified in the channel.¹

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

The primitive provided by the MN is called $RC_Broadcast(m)$, and it allows the broadcasting of a message m to all processes. In practice this is done by sending m to each channel that connects to any other process.

5.2.1.3 Reliable Broadcast

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

It is defined formally as follows:

- RB1 Validity : If a correct process p_i broadcasts a message m , then p_i eventually delivers m .
- RB2 Agreement : If a correct process p_i delivers a message m , then all correct processes eventually deliver m .
- RB3 Integrity : For any message m , every correct process delivers m at most once, and only if m was previously broadcasted by $sender(m)$.

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

The implemented reliable broadcast protocol was originally proposed in [12], and it is presented in Algorithm 3. An instance of the protocol, identified by $rbid$, starts with the sender broadcasting a message (INITIAL, m , $rbid$) to all processes. Upon receiving this message a process sends a (ECHO, m , $rbid$) message to all processes. It then waits for at least $\lfloor \frac{n+f}{2} \rfloor + 1$ (ECHO, m , $rbid$) messages or $f + 1$ (READY, m , $rbid$) messages, and then it transmits a (READY, m , $rbid$) message to all processes. Finally, a process waits for $2f + 1$ (READY, m , $rbid$) messages to deliver m . The broadcasts inside the protocol are made via the reliable channels.

5.2.1.4 Echo Broadcast

The *echo broadcast* protocol is a weaker and more efficient version of reliable broadcast. Its properties are somewhat similar, however, it does not guarantee that all correct processes deliver a broadcasted message if the sender is corrupt [88]. In this case, the protocol only ensures that the subset of correct processes that deliver will do it for the same message.

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

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

Algorithm 3 Reliable Broadcast protocol (for process p_i).

Function R_Broadcast (v_i , rbid)

INITIALIZATION:

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

TASK T0 (SENDER ONLY):

- 1: RC_Broadcast ($\langle \text{INITIAL}, v_i, \text{rbid} \rangle$);

TASK T1:

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

- EB2 Agreement 1 : If the sender is correct and a correct process p_i delivers a message m , then all correct processes eventually deliver m .
- EB3 Agreement 2 : If the sender is corrupt and a correct process p_i delivers a message m , then no correct process delivers $m' \neq m$.
- EB4 Integrity : For any message m , every correct process delivers m at most once, and only if m was previously broadcasted by $\text{sender}(m)$.

The implemented *echo broadcast* primitive was originally proposed in [88], and is a variant of the previously described *reliable broadcast* protocol. It is presented in Algorithm 4.

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

5.2.1.5 Binary Consensus

A *binary consensus* allows correct processes to agree on a binary value. Each process p_i proposes a value $v_i \in \{0, 1\}$ and then all correct processes decide on the same value $b \in \{0, 1\}$. In addition, if all correct processes propose the same value v , then the decision must be v .

Algorithm 4 Echo Broadcast protocol (for process p_i).**Function** E_Broadcast (v_i , $ebid$)

INITIALIZATION:

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

TASK T0 (SENDER ONLY):

- 1: RC_Broadcast ($\langle \text{INITIAL}, v_i, \text{rbid} \rangle$);

TASK T1:

- 1: **wait until** have been delivered at least one $\langle \text{INITIAL}, v, \text{rbid} \rangle$ or $\frac{n+f}{2} \langle \text{ECHO}, v, \text{rbid} \rangle$
- 2: RC_Broadcast ($\langle \text{ECHO}, v, \text{rbid} \rangle$);
- 3: **wait until** have been delivered at least $2f + 1 \langle \text{ECHO}, v, \text{rbid} \rangle$
- 4: **return** v ;

Binary consensus is formally defined by the following properties:

- BC1 Validity : If all correct processes propose the same value b , then any correct process that decides, decides b .
- BC2 Agreement : No two correct processes decide differently.
- BC3a Termination : Every correct process eventually decides.

Given the FLP impossibility result, there is no deterministic algorithm that can guarantee the termination property of consensus in our system model, which is completely asynchronous. The solution is to resort to a randomized model that guarantees the termination in a probabilistic way (as opposed to a deterministic way). As such, the termination property is changed to the following:

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

The implemented protocol is adapted from a randomized algorithm previously presented in [12]. The protocol has an expected number of communication steps for a decision of 2^{n-f} , and uses the underlying *reliable broadcast* as the basic communication primitive. The main advantage of this algorithm is that it does not use any cryptography whatsoever (although its dependence on a reliable communication channel, in practice, implies the use of a relatively cheap cryptographic hash function of some sort).

The protocol, which is presented in Algorithm 5, proceeds in 3-step rounds, running as many rounds as necessary for a decision to be reached. The first step (lines 2-9) of an execution of the protocol identified by $bcid$ starts when each process p_i (reliably) broadcasts its proposal v_i . Then waits for $n - f$ *valid* messages and changes v_i to reflect the majority of the received values. In the second step (lines 10-17), p_i broadcasts v_i , waits for the arrival of $n - f$ *valid* messages, and

Algorithm 5 Binary Consensus protocol (for process p_i).

Function B_Consensus (v_i , bcid)

```

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

```

if more than half of the received values are equal, v_i is set to that value; otherwise v_i is set to the undefined value \perp . Finally, in the third step (lines 18-27), p_i broadcasts v_i , waits for $n - f$ *valid* messages, and decides if at least $2f + 1$ messages have the same value $v \neq \perp$. Otherwise, if at least $f + 1$ messages have the same value $v \neq \perp$, then v_i is set to v and a new round is initiated. If none of the above conditions apply, then v_i is set to a random bit with value 1 or 0, with probability $\frac{1}{2}$, and a new round is initiated.

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

5.2.1.6 Multi-valued Consensus

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

Formally, it is defined as follows:

- MVC1 Validity 1 : If all correct processes propose the same value v , then any correct process that decides, decides v .
- MVC2 Validity 2 : If a correct process decides v , then v was proposed by some process or $v = \perp$.
- MVC3 Validity 3 : If a value v is proposed only by corrupt processes, then no correct process that decides, decides v .
- MVC4 Agreement : No two correct processes decide differently.
- MVC5 Termination : Every correct process eventually decides.

The implemented protocol is adapted from the multi-valued consensus proposed in [24]. It uses the services of the underlying *reliable broadcast*, *echo broadcast*, and *binary consensus* layers. The main difference from the original protocol is the use of echo broadcast instead of reliable broadcast at a specific point, and a simplification of the validation of the vectors used

Algorithm 6 Multi-valued Consensus protocol (for process p_i).

Function M_V_Consensus (v_i, cid)

```

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

to justify the proposed values. These changes grant greater efficiency to the protocol without compromising its correctness. The protocol is presented in Algorithm 6.

The protocol starts when every process p_i announces its proposal value v_i by reliably broadcasting a (INIT, v_i) message (line 1). The processes then wait for the reception of $n - f$ INIT messages and store the received values in a vector V_i (lines 2-3). If a process receives at least $n - 2f$ messages with the same value v , it echo-broadcasts a (VECT, v , V_i) message containing this value together with the vector V_i that justifies the value; otherwise, it echo-broadcasts the default value \perp that does not require justification (lines 4-9). The next step is to wait for the reception of $n - f$ *valid* VECT messages (line 10). A VECT message, received from process p_j , and containing vector V_j , is considered *valid* if one of two conditions hold: (a) $v = \perp$; (b) there are at least $n - 2f$ elements $V_i[k] \in \mathcal{V}$ such that $V_i[k] = V_j[k] = v_j$. If a process does not receive two *valid* VECT messages with different values, and it received at least $n - 2f$ *valid* VECT messages with the same value, it proposes 1 for an execution of the *binary consensus*, otherwise it proposes 0 (lines 11-16). If the binary consensus returns 0, the process decides on the default value \perp . If the binary consensus returns 1, the process waits until it receives $n - 2f$ *valid* VECT messages (if it has not done so) with the same value v and then it decides on that value (lines 17-22).

5.2.1.7 Vector Consensus

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

This problem is adapted from the problem of *interactive consistency*, defined for synchronous systems, to asynchronous systems [71]. While in interactive consistency the problem requires that the decision vector is composed by the values proposed by all correct processes, in vector consensus the requirement is that the decision vector is formed by a majority of values proposed by correct processes.

Vector consensus is formally defined by the following properties:

- VC1 Vector Validity : Every correct process that decides, decides on a vector V of size n :
 - $\forall p_i$: if p_i is correct, then either $V[i]$ is the value proposed by p_i or \perp .
 - at least $(f + 1)$ elements of V were proposed by correct processes.
- VC2 Agreement : No two correct processes decide differently.
- VC3 Termination : Every correct process eventually decides.

The implemented protocol is the one described in [24], which uses *reliable broadcast* and *multi-valued consensus* as underlying primitives. The protocol, which is presented in Algorithm 7,

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

Algorithm 7 Vector Consensus protocol (for process p_i).

Function Vector_Consensus (v_i , v_{cid})

```

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

5.2.1.8 Atomic Broadcast

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

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

- AB1 Validity : If a correct process broadcasts a message m , then some correct process eventually delivers m .
- AB2 Agreement : If a correct process delivers a message m , then all correct processes eventually deliver m .
- AB3 Integrity : For any identifier ID , every correct process p delivers at most one message m with identifier ID , and if $sender(m)$ is correct then m was previously broadcasted by $sender(m)$.
- AB4 Total order : If two correct processes deliver two messages m_1 and m_2 , then both processes deliver the two messages in the same order.

The implemented protocol was adapted from a proposal in [24]. The main difference from the original protocol is that it has been adapted to use multi-valued consensus instead of

vector consensus and to utilize message identifiers for the agreement task instead of cryptographic hashes. These changes were made for efficiency and have been proved not to compromise the correctness of the protocol. The protocol uses *reliable broadcast* and *multi-valued consensus* as primitives.

Algorithm 8 Atomic Broadcast protocol (for process p_i).

INITIALIZATION:

- 1: $R_delivered_i \leftarrow \emptyset$; {messages delivered by the reliable broadcast protocol}
- 2: $A_delivered_i \leftarrow \emptyset$; {messages delivered by the atomic broadcast protocol}
- 3: $aid_i \leftarrow 0$; {atomic broadcast identifier}
- 4: $num_i \leftarrow 0$; {message number}
- 5: $\forall j: B[j] \leftarrow 0$; {window start}
- 6: **activate task** (T1,T2);

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

- 7: R_Broadcast ($\langle A_MSG, num_i, m, i \rangle$);
- 8: $num_i \leftarrow num_i + 1$;

TASK T1:

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

TASK T2:

- 23: **upon** $\langle A_MSG, num_j, m, j \rangle$ is delivered by the reliable broadcast protocol
 - 24: $R_delivered_i \leftarrow R_delivered_i \cup \{ \langle A_MSG, num_j, m, j \rangle \}$;
 - 25: **end upon**
-

The atomic broadcast protocol, presented in Algorithm 8, is conceptually divided in two tasks: (1) the broadcasting of messages, and (2) the agreement over which messages should be delivered.

When a process p_i wishes to broadcast a message m , it simply uses the reliable broadcast to send a $(A_MSG, i, rbid, m)$ message where $rbid$ is a local identifier for the message (lines 7-8). Every message in the system can be uniquely identified by the tuple $(i, rbid)$.

The agreement task (2) is performed in rounds. A process p_i starts by waiting for A_MSG messages to arrive. When such a message arrives, p_i constructs a vector V_i with the identifiers of the received A_MSG messages and reliable broadcasts a (AB_VECT, i , r , V_i) message, where r is the round for which the message is to be processed (lines 10-11). It then waits for $n - f$ AB_VECT messages (and the corresponding V_j vectors) to be delivered and constructs a new vector W_i with the identifiers that appear in $f + 1$ or more V_j vectors (lines 12-13). The vector W_i is then proposed as input to the *multi-valued consensus* protocol and if the decided value W is not \perp , then the messages with their identifiers in the vector W can be deterministically delivered by the process (lines 14-16).

The protocol applies a window of messages to be delivered. Its purpose is to impose a limit on the identifiers that can be proposed to the multi-valued consensus primitive (line 14). This serves to ensure that processes will not indefinitely propose more identifiers while the messages with the identifiers within the window are not delivered by the atomic broadcast protocol. The variable B_j indicates the beginning of the window for process j and L is the window size. So, for example, if $B_j = 10$ and $L = 50$, task T2 will only consider reaching agreement on the order of messages whose identifier (j, num) has $10 \leq num < 50$.

5.2.2 Overlay Network Protocols

An overlay network is an attractive and cheap way to share resources and to increase computing power. In the last few years, overlay networks have rapidly evolved and emerged as a promising platform to deploy new applications and services in the internet. The main reasons for this is that overlay networks are self-organizing, have a decentralized nature, good scalability, efficient query search, and good resilience in the presence of node failures [3, 64]. However one of the main challenges of these systems is malicious faults (attacks) [26].

This section studies overlay network protocols and qualitatively analyzes the dependability of overlay protocols with reference to the following properties: scalability, reliability, security, availability and integrity.

5.2.2.1 Overlay Network Protocols

Four types overlay protocols are distinguished based on their degree of decentralization, topology, and routing mechanism of query search.

Type I - Purely unstructured decentralized : This overlay protocol is a distributed system without any centralized control. In such systems all nodes are equivalent in functionality, named servent (SERVer + cliENT). This means all nodes of a P2P system can act at the same time as server as well as a client. The logical P2P topology in these systems is often an unstructured mesh. The query is executed hop-by-hop through the mesh till success/failure or timeout. An example of these systems is Gnutella [19].

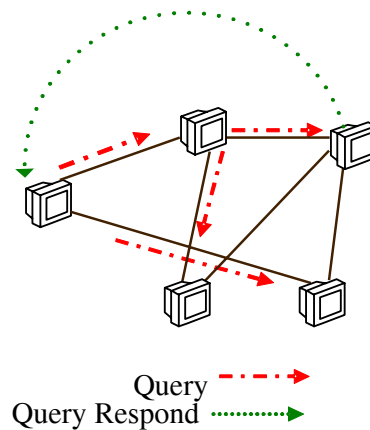


Figure 5.2: Flooding search in Type I protocols.

The routing algorithm of purely unstructured systems like Gnutella use flooding broadcast of queries for routing. In these systems each query from a peer is flooded (broadcasted) to directly connected peers, which themselves flood their peers and etc., until the request is answered or a maximum number of flooding steps occurs. In order to avoid loops, the nodes use the unique

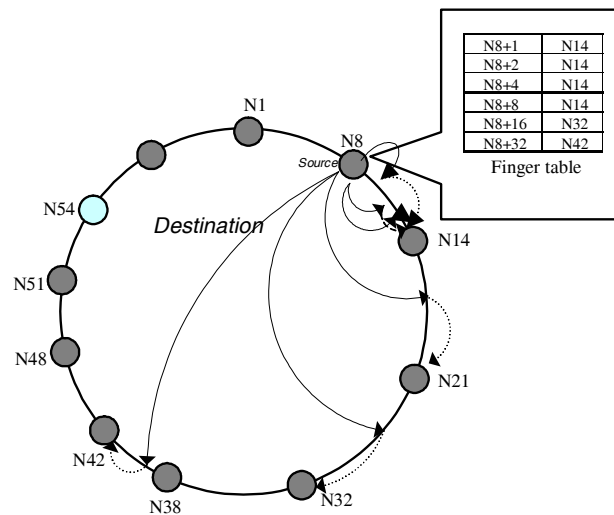


Figure 5.3: Chord routing algorithm.

message identifiers to detect and drop duplicated messages. When the resource is found in a certain node, it initiates a direct out-of-network for downloading, establishing a direct connection between the source and target node (see Figure 5.2).

Type II - Purely structured decentralized : They are similar to overlay networks of type I with the difference that the logical P2P topology is a structured topology such as a mesh, a ring, a d-dimension torus or a butterfly. These structured topologies are usually constructed using distributed hashing tables (DHT) techniques. The query is also executed hop-by-hop through the structured topology, and is sure to be successful after a deterministic number of hops in ideal case. Examples of this system are Chord [86], CAN [72], and Pastry [76].

The routing algorithm in these systems adds structure to the way information about resources are stored using distributed hash tables. With this method the queries can be efficiently routed to the node with the desired resource. This method also reduces the number of hops that must be taken to locate a resource. Among examples of these systems we have chosen the Chord routing algorithm (see Figure 5.3).

Type III - Hybrid centralized indexing : In these systems there is a central server that maintains information about registered users to the network. Each arriving node needs to actively notify the server, then other nodes only need to search peer's address at the server about its wanted objects. There is end-to-end interaction between two peer clients. Napster is an example of these systems [1].

In these systems a peer connects to a centralized directory server, which stores all information regarding location and usage of resources. Upon request from a peer, the central index will match the request with the best peer in its directory that matches the request. The best peer could be the one that is cheapest, fastest, nearest, or most available, depending on the user needs. Then the data exchange will occur directly between the two peers. Napster uses this method. Figure 5.4

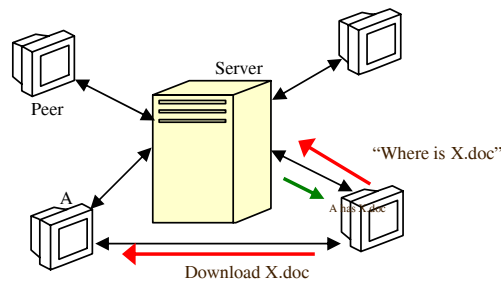


Figure 5.4: Routing algorithm in Napster.

shows routing in Napster as an example of these systems.

Type IV - Hybrid decentralized indexing : In these systems some nodes, called super-nodes, are central servers and they register users to the system and also facilitate the peer discovery process. In such systems peers are automatically elected to become super-nodes if they have sufficient bandwidth and good processing power. Like systems of type III there is end-to-end interaction (data exchange) between two peer clients. Figure 5.5 shows these systems. Kazaa [65] and Morpheus [98] are two decentralized indexing systems.

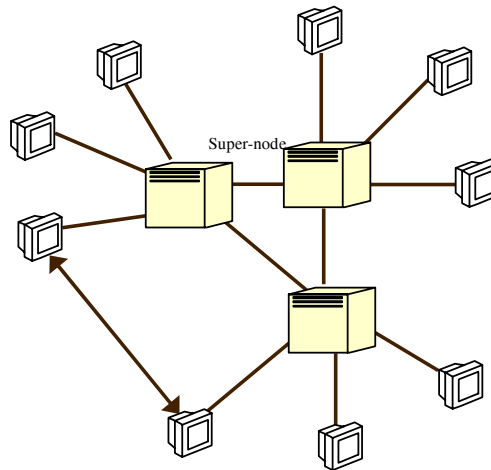


Figure 5.5: Hybrid decentralized indexing.

In these systems each super-node is associated with a set of nodes and every node connects to the super-node to which it belongs. When a search for a resource item is issued by a node, a lookup message will follow the path from the super-node of the node to the other super nodes; the operation will be repeated until success or until all paths are completely searched (see Figure 5.5).

5.2.2.2 Analyzing the Dependability of Overlay Network Protocols

The dependability of a system is that property of the system which allows reliance to be justifiably placed on the service it delivers [13]. In fact a system's dependability expresses

the expectations of its users regarding how the system meets their requirements. We consider dependability qualitatively in terms of the properties reliability, scalability, availability, security and integrity.

The evaluation has been performed using a 4-level grade score, assuming that grade "A" is the highest score (excellent), grade "B" is the good score, grade "C" is the medium score and grade "D" is the lowest one (bad).

Reliability is a measure of the continuous delivery of a proper service [13]. In principle, the absence of centralized control and coordination makes P2P systems robust with respect to failures that might occur at any peer. Faults at client peers don't usually affect system behavior because only local information is lost, but faults at server peers can result in data loss and denial of service.

- Type I : type I's topology is random and flat. All peers are equal and there is no single point of failure. In fact in this type as no one peer is more important than any other peer, the network is satisfactory reliable. Although all nodes are principally equal, in practice a few nodes become strategically more important than the others because of the high number of connections they manage. So if a peer with many connections fails, serious consequences for routing efficiency occur because several peers can become difficult or impossible to reach. We assign grade "C" to it.
- Type II : type II's topology is purely decentralized and there is no single point of failure. Furthermore, these systems store reference to a resource in the nearest peers instead only storing the next one. Moreover, each peer runs a stabilization algorithm that maintains correctness on references. Chord is a good example for it. So this property increases reliability of the system. We assign grade "A" to it.
- Type III : The centralized nature of this type makes it fragile with respect to faults. When the central server stops working, all services are not usable. To restore this type of system, client peers should upload information to the centralized server again. So the presence of a single point of failure makes this type of P2P systems unreliable. It has grade "D" for reliability.
- Type IV : Super nodes are selected at run-time based on their capabilities (CPU power, bandwidth, non-local IP, and etc.), hence the P2P network has a strong support for self-organization. It means that if a super-node goes down because of failure, another powerful peer becomes super node. This type is more reliable than type I, So reliability is satisfactory good and we assign grade "B" to it.

Scalability is the ability to operate without a noticeable decline in performance despite the change in the number of nodes that constitutes the system. In fact scalability is the degree of adaptability that a system exhibits with respect to increasing load situations [28]. P2P systems are dynamic systems and at any time a node can leave and join the system.

- Type I : in these systems every initial search query generates network traffic. When the number of peers participating in the network increase, the growing search queries (flood

search with TTL) negatively influence traffic (increase traffic). So increasing the number of nodes reduces performance of the system because of flooding overhead. So the system has a limitation in scalability. This type has grade "C" for scalability.

- Type II : As there are efficient query routing searches in these systems, a growing number of search queries does not negatively influence on traffic. Hence system has a good scalability. We assign grade "A" to it.
- Type III : In these systems a search query directly goes to the central server. By increasing the number of search queries, the server finally goes down due to an overload situation. So the central server easily becomes overloaded with requests because of performance and bandwidth limitations. Taking scalability into account, it is the least scalable topology of P2P systems. We assign grade "D" to it.
- Type IV : searching only in the subsets of the decentralized networks results in a serious reduction of the scope of flooding. Scalability in these systems is satisfactory mainly due to limited scope of network flooding, self-organizing and adequate number of super nodes. By having some super nodes, system does not go in overload situation easily. Furthermore if a super node goes down in an overload situation, another powerful peer becomes super node (self-organizing property). It has grade "B".

Availability is a measure of the frequency of periods of improper service [13]. We say a P2P is an available system if each query is answered and every response is adequately delivered in reasonable time (satisfying the user).

In P2P systems routing algorithms highly affect the user satisfaction level and availability of the system, because it directly responsible for searching time and also guarantees that search is accurate and feasible in reasonable time.

- Type I : as there is no algorithmic and structural mechanism for query search, system has weak availability. Search queries are based on flooding with time-to-live (TTL) value; hence there is no guarantee for finding results and no guarantee for quality of service and time to find. We assign grade "D" for availability of this group.
- Type II : Since the lookup is efficient and straightforward, the level of availability may seem to be high. However it depends on the consistency of information deployed among the nodes: joining and leaving peers invoke wide-area inconsistency. So availability is satisfactory for users and we assign grade "B" to it.
- Type III : in these systems the algorithm of resource discovery is simple. A peer asks a centralized directory for a search query (existence and location). The simplicity of resource discovery results in high availability provided that the central server is not overloaded. Searching covers the entire system scope without any communication overhead and the search results are delivered with minimum latencies. Grade "A" is suitable for this group.
- Type IV : furthermore decentralization, the search flooding reduces in the sub-networks of super nodes; hence response time is satisfactory short. This type has satisfactory availability in both search results and search time. Availability has grade "B" in this group.

Integrity is non-occurrence of improper alteration of information [13]. In P2P systems, integrity means keeping system state information coherent and up-to-date among all peers.

- Type I : Topology is random and dynamic, and the search process does not utilize any caches nor indices, hence data integrity is not an issue. On the other hand, there are no specific mechanisms to support up-to-date data in peers. We assign grade "D" to it.
- Type II : Although search query and lookup is optimized, leaving and joining peers make data integrity hard. To ensure efficient lookup, the routing information in all peers must be kept up-to-date by implementing stabilization protocol (it complicates the system). So the complexity of handling leaves and joins negatively influences the network's integrity. It has grade "C".
- Type III : The centralized topology provides a high level of integrity as all information related to the state of the system is concentrated in one place. Hence, keeping information up-to-date on participating peers and their resources is easy. We assign grade "A" to it.
- Type IV : Hybrid topology (decentralized and centralized) influences integrity of system. Peers frequently exchange information on their neighbors, and, moreover, are able to optimize their connections in terms of locality and workload. So dynamics positively influence the integrity of system. Frequent exchanges of information allow keeping lists of super-nodes up-to-date. We assign grade "B" to it.

Security is defined as a system's ability to manage and protect sensitive information [28]. Since in P2P systems the set of active peers is dynamic and peers don't trust each other, achieving a high level of security in P2P systems is more difficult than in client server systems. Traditional security mechanisms to protect data and systems from intruders and attacks, such as firewalls, cannot protect P2P systems since they are essentially globally distributed. These mechanisms can also inhibit P2P communication.

- Type I : Among four P2P types mentioned in this article, type I is the best type against attacks and malicious faults. It has a random distributed nature. All nodes are equal and there is no reference peer for attack. However, attackers can exploit the small-world network model to target peers that manage either numerous or important wide-range connections. In fact only a homogenous distribution of connections provides really no reference point for attackers. Main threats in this type are flooding, malicious content, virus spreading, attack on queries and DoS attack. However we assign grade "A" to it, in related to other approaches.
- Type II : This type is the worst type against attacks and malicious faults. It has systematic and algorithmic structure. Any peer can be a reference point for attackers. As an attacker gets access to a peer, he/she can start various attacks. Main attacks are routing poisoning, partitioning and virtualization into incorrect network when a new peer joins and contacts malicious peer, lookup and storage attack, inconsistency behaviors of peers, DoS attack and unsolicited responses to a lookup query. Hence we assign grade "D" to this type.

P2P types	Reliability	scalability	availability	Integrity	security
Type I	C	C	D	D	A
Type II	A	A	B	C	D
Type III	D	D	A	A	C
Type IV	B	B	B	B	

Table 5.1: Summary of dependability attributes for P2P types.

P2P types	Summary	SUM
Type I	$A + 2C + 2D$	7
Type II	$2A + B + C + D$	14
Type III	$2A + C + 2D$	11
Type IV	$4B + C$	13

Table 5.2: Summary of dependability grades for P2P types.

- Type III : In hybrid systems, attackers focus on centralized points. If an attacker gets access to the central server, performance degrades to zero. However, a centralized architecture might be more secure towards such attacks due to a centralized trusted server. In the other words it can prevent such attacks by employing firewalls like in client-server networks. In general we should use all protected mechanisms used in client-server network for this type of P2P networks. Main threats are DoS, man-in-the-middle, and Trojan attacks. We assign grade "C" to it.
- Type IV : In this type some P2P networks such as Farsite [5] implement Byzantine protocols to ensure a certain level of security and trust because the protocol won't allow the entire group to misbehave or fail unless malicious or faulty peers make up one third of the group's members. A drawback is that the Byzantine approach is effective only for small group of servers. However not all hybrid P2P types implement a Byzantine protocol. Main threats in this type are flooding, malicious or fake content, virus, etc. We assign grade "C" to this group.

Summary : Table 5.1 shows summary of grades that have been assigning to dependability attributes of P2P networks. Now suppose A has 5 points, B has 3 points, C has 1 point and D has no point. Table 5.2 shows the sum of grades. Type II such as Chord and CAN has highest score among the analyzed P2P types. So type II is the most dependable P2P network. Type IV has second rank in dependability attributes as it composes both decentralized and centralized natures. Type I is the least dependable P2P network.

5.3 Activity Support Protocols

5.3.1 CIS Protection Protocol

The main aspect that ensures a high level of dependability of the CIS is its intrusion tolerance, which is achieved through the (physical or virtual) replication of its components. The objective is to guarantee that valid and correct commands are delivered to a LAN, while invalid or conspicuous messages are blocked by the protection service. This should occur even if some of the CIS replicas are controlled by a malicious adversary. In this section we describe the design of the intrusion-tolerant CIS, starting with the design rationale, and then we define the algorithms it executes.

Several intrusion-tolerant services have been proposed in the literature (e.g., storage [17, 39, 61], certification authorities [73, 101], and DNS [16]), either based on Byzantine quorum systems (BQS) [60, 101] or state machine replication (SMR) [17, 78]. However, the CIS design presents two very interesting challenges that make it essentially different from those services. The first is that a firewall-like component has to be transparent to protocols that pass through it, so it can not modify the protocols themselves to obtain intrusion tolerance. This also means that recipient nodes will ignore any internal CIS intrusion tolerance mechanisms, and as such they cannot protect themselves from messages forwarded by faulty replicas not satisfying the security policy. This section shows that these challenges are not trivial and presents a solution that is based neither on BQS nor on SMR.

Although the CIS protection service is designed for critical infrastructures protection, any system/network could be protected by a CIS. However, the design takes into consideration the specific aspects of critical infrastructures since (i.) the solution must comply with legacy/standard components that cannot be easily replaced in critical infrastructures; and (ii.) the solution we propose is more costly and has less packet processing capacity than common (non-replicated) firewalls, so it is more adequate for protecting low-traffic critical facilities than high-bandwidth corporate networks.

5.3.1.1 Design Rationale

To understand the *rationale of the design* of the intrusion-tolerant CIS, consider the problem of implementing a replicated firewall between a non-trusted WAN and the trusted LAN that we want to protect. Further assume that we wish to ensure that only the correct messages (according to the deployed policy) go from the WAN side, through the CIS, to the *station computers*² in the LAN. A first problem is that the traffic has to be received by all n replicas, instead of only 1 (as in a normal firewall), so that messages can be evaluated by all replicas. A second problem is that up to f replicas can be faulty and behave maliciously, both towards other replicas and towards the station computers.

²Station computers in SCADA/PCS networked systems are the front-ends of control devices.

Our solution to the first problem is to use a device (e.g., an Ethernet hub) to broadcast the traffic to all replicas. These verify whether the messages comply with the OrBAC policy and do a vote, approving the messages if and only if at least $f + 1$ different replicas vote in favor. A message approved by the CIS is then forwarded to its destination by a randomly selected replica, so there is no unnecessary traffic multiplication inside the LAN. The way we deal with omissions in the broadcast is addressed later in this Section.

The second problem is usually addressed with masking protocols of the Byzantine type, which extract the correct result from the n replicas, despite f maliciously faulty: only messages approved by $f + 1$ replicas should go through (one of which must be correct since at most f can be faulty). Since the result must be sent to the station computers, either it is consolidated at the source, or at the destination.

The simplest and most usual approach is to implement a front-end in the destination host that accepts a message if: (1) $f + 1$ different replicas send it; or (2) the message has a certificate showing that $f + 1$ replicas approve it [11]; or (3) the message has a signature generated by $f + 1$ replicas using a threshold cryptography scheme [29]. These solutions would imply modifying the hosts' software. However, modifying the software of the SCADA/PCS system can be complicated, and the traffic inside the protected LAN would be multiplied by n in certain cases (every replica would send the message to the LAN), so this solution is undesirable.

So we should turn ourselves to consolidation at the source, and sending *only one, but correct, forwarded message*, in a way similar to an active replication scheme of Delta-4 [22]. However, what is innovative here is that source-consolidation mechanisms should be transparent to the (standard) station computers. Moreover, a faulty replica has access to the LAN (contrarily to the proposal of [22] where only the fail-silent adapters had access to the LAN) so it can send incorrect traffic to the station computers, which typically can not distinguish faulty from correct replicas. This makes consolidation at the source a hard problem.

The solution to the second problem (the existence of up to f faulty replicas) lies on using IPsec, a set of standard protocols that are expected to be generalized in SCADA/PCS systems, according to best practice recommendations from expert organizations and governments [87]. Henceforth, we assume that the IPsec Authentication Header (AH) protocol runs both in the station computers and in the CIS replicas. The basic idea is that station computers will only accept messages with a valid IPsec/AH Message Authentication Code (MAC), which can only be produced if the message is approved by $f + 1$ different replicas. However, IPsec/AH MACs are HMACs (*Keyed-Hashing for Message Authentication* [56]), generated using a shared key³ and a hash function, so it is not possible to use threshold cryptography. As the attentive reader will note, the shared key storage becomes a vulnerability point that can not be overlooked in a high resilience design, therefore, *there must be some secure component that stores the shared key and produces MACs for messages approved by $f + 1$ replicas*.

The requirement in the previous paragraph implies a set of trustworthy (secure) components immersed in non-trustworthy (Byzantine-on-failure) environment. Recent research points to the need for representing these scenarios where different fault models coexist, through hy-

³We assume that IPsec/AH is used with manual key management [53].

brid (and not homogeneous) distributed system models and architectures. In such architectures, *stronger* components also nick-named *wormholes*, provide services to the rest of the system following weaker assumptions, through a well-defined interface [89].

Figure 5.6 represents the intrusion-tolerant CIS architecture. Local wormholes (represented by the small W boxes) provide services for a secure voting protocol that produces a MAC for a message if at least $f + 1$ replicas approved it. Each CIS replica is deployed in a different operating system (e.g., Linux, FreeBSD, Windows XP), and the operating systems are configured to use different passwords and different internal firewalls (e.g., iptables, ipf). A second traffic replication device (see figure, right hand side) is used precisely for the replicas to receive whatever the others send to the LAN. This enables us to implement controls to reduce the probability of a message being forwarded by more than one replica.

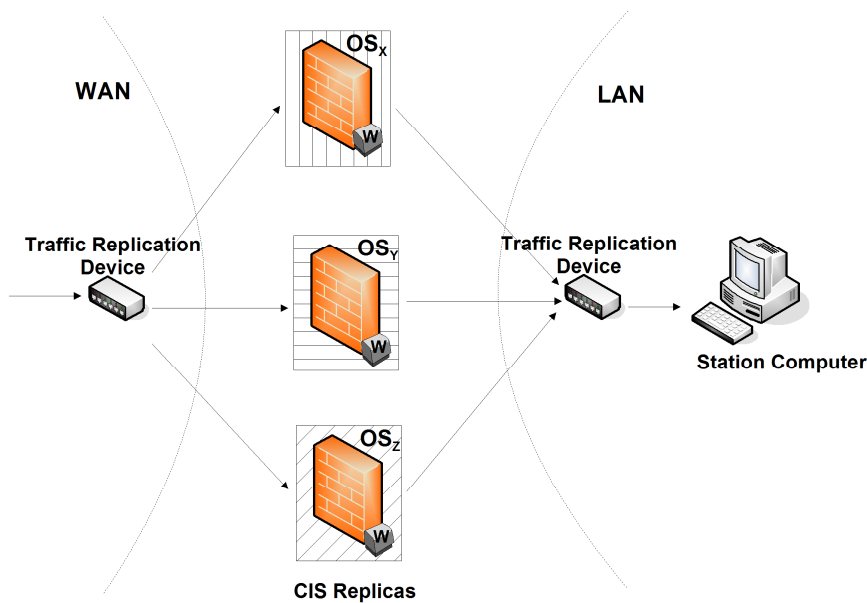


Figure 5.6: Intrusion-tolerant CIS architecture.

The CIS does not provide exactly-once semantics, i.e., messages can be lost when the traffic is high and the reception buffers of the replicas become full. This is not different from regular firewalls, except that the CIS operation is more complex so the throughput is expected to be lower. However, it is important to notice that almost all wide area control protocols, and specially the ones designed for Power Systems like ICCP [47] and IEC 61850 [48], are designed on top of TCP, which ensures reliable communication even if the network (in this case the CIS) loses messages.

5.3.1.2 System Model

The system is composed by n CIS replicas $CIS = \{CIS_1, \dots, CIS_n\}$. These replicas are deployed in the intersection between the WAN and the LAN in such a way that all data crossing the boundaries of one of these networks must pass through the CIS. The hybrid system model

encompasses two parts [89]: the *payload* and the *wormhole*.

Payload. *Asynchronous system* with $n \geq 2f + 1$ replicas in which at most f can be subject to *Byzantine failures*. If a replica does not fail during the execution of the CIS it is said to be *correct*, otherwise it is said to be *faulty*. Every CIS replica has a local clock that is assumed only to make progress. These clocks are not synchronized. We assume *fault independence* for the replicas, i.e., the probability of a replica be compromised is independent of another replica failure. This assumption is substantiated by the diversity mechanisms employed in the CIS replicas (different OS, passwords, and internal firewall), and its coverage can be made as high as desired, through additional kinds of diversity [68]. Finally, we assume also that the station computers can not be compromised⁴.

Wormhole. *Asynchronous secure tamper proof subsystem* $W = \{W_1, \dots, W_n\}$ in which at most $f_c \leq f$ local wormholes can *fail by crash*. We assume that when a local wormhole W_i crashes, the corresponding payload replica CIS_i crashes together. Each local wormhole stores two symmetric keys: K_W – shared between the wormholes and used for vote authentication; and K_{LAN} – shared between the station computers of the LAN and the local wormholes, such that station computers only accept messages authenticated with this key (the IPsec key).

Network. The assumptions underlying LAN and WAN communication are as follows. We consider that the messages arriving at CIS replicas both from the WAN and the LAN have *unreliable fair multicast* semantics, a trivial extension of the commonly assumed *fair links* abstraction [6, 59] to multicast: if a message is multicasted infinitely many times it will be received by all its correct receivers infinitely many times. The two primitives offered by this service are: $U\text{-multicast}(G, m)$, to multicast a message m to the group G , and $U\text{-receive}(G, m)$, to receive m that was multicast to G , where G can be either WAN or LAN. This is substantiated in practice by the traffic replication devices. We assume that *all* communication between replicas and other machines from the WAN and the LAN are based on these primitives. Additionally, all CIS replicas communicate through point-to-point reliable channels for voting approved messages. These channels can be implemented on the protected LAN or on a separate network (that can be a *Virtual LAN* configured on the LAN or WAN switches acting as traffic replication devices – see Figure 5.6).

Cryptography. Our protocols use a *collision-resistant hash function* H , which receives an arbitrarily long input and produces a fixed-length output in such a way that it is infeasible to find two messages with the same hash. Additionally, the HMACs used in IPSEC are assumed to inherit the collision resistance property from the hash functions in which they are based [56], i.e., that it is infeasible to find two messages that for a key K have the same MAC. A message m is signed with a key K by concatenating m with a MAC of m produced with K . We use m_σ to represent a message m signed with some key K , i.e., $m_\sigma = m.MAC(m, K)$.

⁴It is the trusted network that we aim to protect, exactly in the sense of preventing it from being compromised.

5.3.1.3 Service Properties

Before defining the service properties offered by the CIS, let us define the concept of legal message: a message is said to be *legal* if it is in accordance with the current deployed policy P . A message not in accordance with P is said to be *illegal*. Moreover, a message is said to be *processed* by the destination machine if its content is delivered to the application layer (e.g., the SCADA system). The basic properties offered by the CIS are the following:

- **Validity** : A legal message received by at least one correct CIS replica is forwarded to its destination machine;
- **Integrity** : An illegal message is never processed by its destination machine.

Notice that these two properties are sufficiently weak to be satisfied by a system with unreliable fair multicast communication and strong enough to ensure that only legal messages will be processed at LAN hosts.

5.3.1.4 Message Processing Protocol

Here we present the main protocol executed by the CIS to process messages incoming from the WAN to the protected LAN. The same algorithm is used to handle messages coming from the opposite direction (possibly with a more relaxed policy).

The policy verification is made in a *policy engine* accessed through the function *PolEng_verify*. We assume that all aspects of policy verification are encapsulated inside this component, which acts as an oracle that says if a message is legal or not.

Wormhole interface. The interactions between a replica CIS_i and its local wormhole W_i are made through a well defined interface that offers three services, invoked through the operations described in Table 5.3.

Operation	Return Type	Description
$W_create_vote(m)$	byte array	returns a MAC of message $\langle i, m \rangle$ produced with the wormhole shared key K_W
$W_sign(m, C_m)$	byte array	returns a MAC of message m produced with the shared key K_{LAN} , if C_m contains at least $f + 1$ votes (returned by $W_create_vote(m)$) from different replicas
$W_verify(m_\sigma)$	boolean	returns <i>true</i> if σ is a MAC of m produced with K_{LAN} , and <i>false</i> otherwise

Table 5.3: Wormhole services specification.

The Algorithm. The CIS replicas execute Algorithm 9 for processing incoming messages. The algorithm is composed by three code blocks (WAN message reception, LAN message reception,

and message retransmission) and all these blocks can be executed by different threads. We assume the existence of synchronization mechanisms that manage the concurrent access of the threads to the shared sets (e.g., execution of lines 1 and 2 is atomic). For readability, we chose to not include such mechanisms explicitly in the algorithm since they are not required for algorithm correctness.

T_{vote} is the single configuration parameter of the payload protocol and it defines the expected time required to receive, vote and sign a legal message. Additionally, the algorithm uses three variables: *Voting*, the set of messages being voted; *Pending*, the set of messages received and approved by the replica that were already signed by the wormhole but not yet forwarded to the station computer; and *TooEarly*, the set of correctly signed messages forwarded by some other replica but not yet received (from the WAN) by the replica.

Algorithm 9 CIS payload (replica CIS_i).

<pre> {Parameters} integer T_{vote} {Expected time to vote a message} {Variables} set $Voting = \emptyset$ {Messages being voted} set $Pending = \emptyset$ {Not yet forwarded messages} set $TooEarly = \emptyset$ {Messages forwarded before their arrival} {Code for WAN message reception and processing} upon $U\text{-receive}(WAN, m)$ 1: if $m_\sigma \in TooEarly$ then 2: $TooEarly \leftarrow TooEarly \setminus \{m_\sigma\}$ 3: else 4: if $PolEng_verify(m)$ then 5: $Voting \leftarrow Voting \cup \{m\}$ 6: $m_\sigma \leftarrow approve(m)$ 7: $Voting \leftarrow Voting \setminus \{m\}$ 8: $Pending \leftarrow Pending \cup \{m_\sigma\}$ 9: $waitRandom()$ 10: if $m_\sigma \in Pending$ then 11: $U\text{-multicast}(LAN, m_\sigma)$ 12: end if 13: end if 14: end if </pre>	<pre> {Code for LAN message reception and processing} upon $U\text{-receive}(LAN, m_\sigma)$ 15: if $m_\sigma \in Pending$ then 16: $Pending \leftarrow Pending \setminus \{m_\sigma\}$ 17: else if $W_verify(m_\sigma)$ then 18: $TooEarly \leftarrow TooEarly \cup \{m_\sigma\}$ 19: end if function $approve(m)$ 20: $vote_i \leftarrow W_create_vote(m)$ 21: $\forall CIS_j \in CIS, send(j, \langle VOTE, H(m), vote_i \rangle)$ 22: $C_m \leftarrow \emptyset$ 23: repeat 24: wait until $receive(j, \langle VOTE, H(m), vote_j \rangle)$ 25: $C_m \leftarrow C_m \cup \{vote_j\}$ 26: $\sigma \leftarrow W_sign(m, C_m)$ 27: until $\sigma \neq \perp$ 28: return m_σ {Periodic task for message retransmission} for each T_{vote} that $Voting \neq \emptyset$ 29: $\forall m \in Voting : U\text{-multicast}(WAN, m)$ </pre>
--	---

The algorithm begins when a replica receives a message coming from the WAN (lines 1-14). If the received message is not in the *TooEarly* set and it is legal, all correct replicas will approve it (line 4) and then invoke the *approve* function (which will be explained later) to vote and sign this message (line 6). While the message is being voted and signed, it is stored in the *Voting* set (lines 5 and 7). Then, the message is inserted in the *Pending* set (line 8) and it remains in this set until it is received from the LAN (lines 15-16)⁵. Finally, the replica waits for a random time interval (function *waitRandom* – line 9) and if the message is still in the *Pending* set, it is forwarded to the LAN (lines 10-11). The random waiting is implemented to avoid that all replicas forward the accepted message.

The algorithm contains several controls to deal with message losses, replica failures, and abnormal message ordering. The first of these controls deals with message omissions on the

⁵Recall that when a replica forwards a message to the LAN, it goes not only to the station computers but also to all CIS replicas.

WAN: when a replica receives and approves a message m , it stores it in the *Voting* set (line 5) before starting the vote and sign procedure. The message is removed from this set only when it is signed (line 7). For each T_{vote} time units that *Voting* is not empty, its content is multicasted to the other CISs (line 29). This ensures that the message being voted will eventually be received by other correct CIS replicas (due to the fairness assumption) and will then be voted. It is possible that some replica forwards a correctly signed message to the LAN without some other replicas having received it from the WAN. In order to deal with these “early messages” on the LAN and optimize CIS performance, we use the *TooEarly* set. When a not-pending legal message is received in the LAN, it will be stored in this set (lines 17-18) and stay there until it is received from the WAN (lines 1-2). The *Pending* and *TooEarly* sets are not necessary to satisfy the CIS properties. These sets are used with two goals: to reduce message duplication in the LAN (*Pending* set + *waitRandom* function), and to optimize CIS performance by avoiding policy verification and message approval when a message was already previously signed and forwarded by some other replica (*TooEarly* set). Moreover, given that messages arriving from the WAN and the LAN have unreliable semantics, these sets need to be periodically reset in order to avoid messages staying there forever.

The most important part of the algorithm is the *approve* function (lines 20-28), which comprises the steps executed to vote for and sign a legal message. The function begins with the replica calling the wormhole to build a vote for the message (line 20) and sending this vote to all other replicas (line 21). Each replica then waits to receive votes from other replicas until it manages to get a sufficient number of valid votes to make the wormhole produce a signature for the approved message (lines 23-27).

5.3.1.5 Wormhole Protocol

The implementation of the three services provided by the wormhole is presented in Algorithm 10. The replica id is stored inside the tamper proof memory of the local wormhole together with two symmetric keys that are used to authenticate different messages: the key K_W is used to authenticate vote messages that can be later verified by other wormholes; and the key K_{LAN} is used to sign approved messages to be forwarded to the protected LAN. These keys are used to authenticate messages using MACs.

Algorithm 10 Wormhole services (local wormhole W_i).

<pre> {Parameters} integer i {Replica id – for vote generation} key K_W {Wormholes key – for vote authentication} key K_{LAN} {Service key – for message authentication} {Services} service $W_create_vote(m)$ 1: return $MAC(\langle i, m \rangle, K_W)$ </pre>	<pre> service $W_sign(m, C_m)$ 2: if $\{v \in C_m : \exists j \text{ s.t. } v = MAC(\langle j, m \rangle, K_W)\} \geq f + 1$ then 3: return $MAC(m, K_{LAN})$ 4: else 5: return \perp 6: end if service $W_verify(m_\sigma)$ 7: if $MAC(m, K_{LAN}) = \sigma$ then return true else return false </pre>
--	---

The service $W_create_vote(m)$ uses the key K_W to generate the MAC for $\langle i, m \rangle$ (line 1). Since the id of the replica i cannot be modified by the payload and the key is secretly stored inside

the wormhole, it is impossible for a malicious payload to impersonate other replicas in the voting phase.

$W_sign(m, C_m)$ calculates the MAC σ of m using the shared key K_{LAN} if and only if the replica payload presents a certificate set C_m containing at least $f + 1$ valid votes produced by different replicas wormholes (lines 2-3). If there is no such number of valid votes, the wormhole returns the error value \perp (line 5).

Service $W_verify(m_\sigma)$ receives as input a message m allegedly signed with K_{LAN} and returns *true* if the MAC for m produced using K_{LAN} corresponds to σ , and *false* otherwise (line 7).

5.3.1.6 CIS Design Options

In this section we present some design options for the CIS protection service just described. Since these options make the CIS more costly, we decided not to include them in the main design, but to present these in a separate section.

Dealing with DoS Attacks from Malicious Replicas. A malicious CIS replica can flood the WAN and LAN networks with illegal messages aiming to delay the forwarding of legal messages. This kind of attack can degrade the performance of the CIS as a whole. More generically, intrusion-tolerant designs are typically vulnerable to DoS attacks given that they compromise the network fairness/reliability commonly assumed in Byzantine fault-tolerant algorithms (e.g., [17, 100, 101]).

Considering the CIS architecture described before, a practical way to deal with this problem is to integrate an intrusion detection system in the traffic replication devices in order to monitor the networks connecting the CIS replicas and issue alarms when some replica behaves maliciously. The response to these alarms could be done by a human operator or by some kind of automatic fail-safe system that could shutdown malicious replicas and notify the system administrator. There are available in the market switches with integrated intrusion detection systems, e.g., Cisco Catalyst 6500 and Nortel Ethernet Routing Switch 8600.

An alternative solution is to build a distributed secure wormhole, in which all replicas' local wormholes are connected by a secure network. With this control network, the whole voting protocol can be securely executed by the wormholes (in a crash-failure model), preventing malicious replicas from disturbing it. Moreover, with the local wormholes connected through a secure network, if a malicious replica floods the LAN with invalid messages, other correct CIS replicas can notify the distributed wormhole about this behavior. When the malicious replica's local wormhole knows that at least $f + 1$ other replicas suspect that its payload is faulty, it can shutdown the machine or recover it.

Supporting Statefull Firewalls. The CIS design presented in this section applies to stateless firewalls, which is the most common type of firewall used. However, some applications require statefull security policies, in which traffic is approved or denied taking into consideration the mes-

sages approved/denied in the past (e.g., some data message is only forwarded if some connection message was sent before).

The CIS policy engine can provide statefull semantics as long as one ensures that all messages are verified in the same order in all CIS replicas. In other words, we need an agreement protocol to ensure that all messages are evaluated in total order. This protocol could require either f more replicas and some timing assumptions (e.g., [17]) or the same number of replicas and a distributed secure and timely wormhole [23].

5.3.2 Access Control and Authorization Protocols

In Section 3.3.2, we have identified the web services of different CII's scenarios (scenarios 1, 2, 3, 4) of WP1 D2. In this section, we detail the invocations of the web services according to a PolyOrBAC security policy. The aim is thus to present the protocols of secure communications using PolyOrBAC web services. In this respect, we give PolyOrBAC rules ensuring the secure functioning of the different scenarios.

We briefly explain here the functioning of OrBAC derivation of rules, the derivation of permissions (i.e., instantiation of security rules) can be formally expressed as follows:

$$\begin{aligned}
 & \forall org \in Organisations, \forall s \in Subjects, \forall activ \in Activities, \forall o \in Objects, \forall r \in Roles, \forall a \in \\
 & Actions, \forall v \in View, \forall c \in Contexts, \\
 & Permission(org, r, v, activ, c) \wedge \\
 & Empower(org, s, r) \wedge \\
 & Consider(org, a, activ) \wedge \\
 & Use(org, o, v) \wedge \\
 & Hold(org, s, a, o, c) \\
 & \Rightarrow Ispermitted(s, a, o).
 \end{aligned}$$

This rule means that:

If a security rule specifies that:

- in org, role r can carry out the activity 'activ' on the view 'v' when the context 'c' is True, and
- in org, r is assigned to subject s, and
- in org, action a is a part of activity activ, and
- in org object o is part of view v, and
- the context c is True for the triple (org, s, a, o).

Then s is allowed to carry out a on o.

Example:

$\text{Permission}(\text{B}, \text{Accountant}, \text{Account}, \text{Consulting}, \text{Urgency}) \wedge$
 i.e. Organization B grants role Accountant Permission to perform activity Consulting on view Account within context Urgency.

$\text{Empower}(\text{B}, \text{Bob}, \text{Accountant}) \wedge$
 i.e. Organisation B empower subject Bob with the role Accountant.

$\text{Consider}(\text{B}, \text{OpenXMLFile}(), \text{Consulting}) \wedge$
 i.e. Organisation B considers that action OpenXMLFile() belongs to the activity Consulting.

$\text{Use}(\text{B}, \text{account1}, \text{Account}) \wedge$
 i.e. Organisation B uses object account1 as an Account.

$\text{Hold}(\text{B}, \text{Bob}, \text{OpenXMLFile}(), \text{account1}, \text{Urgency}) \wedge$
 i.e. in organization B, context Urgency is true between subject Bob, object account1 and action OpenXMLFile().

$\Rightarrow \text{Is permitted}(\text{Bob}, \text{OpenXMLFile}(), \text{account1})$
 i.e. subject Bob is allowed to carry out the action OpenXMLFile() on object account1.

5.3.2.1 Scenario 1: DSO Tele-operation (Emergency and Normal Conditions)

Figure 5.7 summarises the different web services, virtual users, and ws-images, and resources involved in the execution of scenario 1.

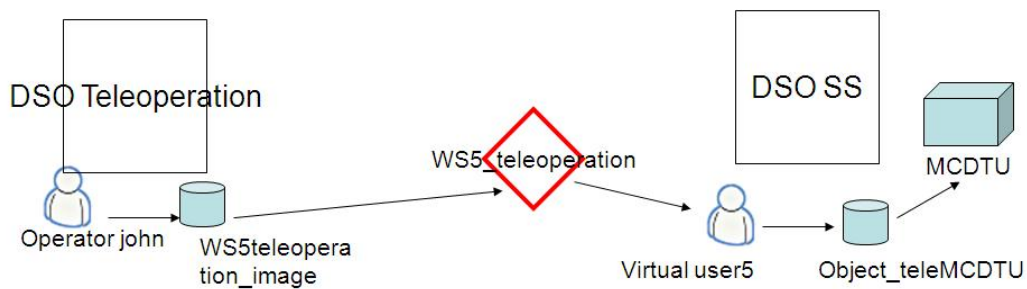


Figure 5.7: Scenario 1 users and services.

A. WS5-teleoperation

- o Provider: DSO SS.
- o Client: DSO ACC.

Execution Scenario:

- The operator John in DSO ACC performs tele-operation over the DSO SS.
- When operator John access (perform tele-operation over) WS5teleoperation-Image, the execution of WS5-teleoperation is automatically activated.
- WS5teleoperation-Image is the local representation of the remote WS5-teleoperation.
- WS5-teleoperation consists in ordering the virtual user5 in DSO SS to access (perform tele-operation request over) object object-teleMCDTU.
- The Virtual user5 is the representation of DSO ACC in DSO SS.
- Virtual user5 is created at the negotiation phase (for WS) between DSO ACC and DSO SS.

We present here the different OrBAC predicates that manage Access Control for the Tele-operation web service at the level of the organisation that invokes the service (DSO ACC) and the one that provides it (DSO SS).

OrBAC rules at DSO ACC

```
Permission(DSO ACC(org), DSO-role for ACC(role), Access(activity), ACC Distribution circuits(view), normal(context) )
Empower(DSO ACC(org), John(Subject), DSO-role for ACC(role))
Consider(DSO ACC(org), RWX(action), Access (activity))
Use(DSO ACC(org), WS5teleoperation-Image(object), ACC Distribution Circuits(view))
Hold(DSO ACC(org), John(Subject), RWX(action), WS5teleoperation-Image(object), emergency(context)).
Then is-permitted(John(Subject), RWX(action), WS1teleoperation-Image(object))
```

OrBAC rules at DSO SS

```
Permission(DSO SS(org), DSO-role for SS(role), Access (activity), DSO SS Distribution Circuits(view), normal(context))
Empower(DSO SS(org), virtual user5(Subject), DSO for SS(role))
Consider(DSO SS(org), RWX(action), Access (activity))
Use(DSO SS(org), object-teleMCDTU(object), DSO SS Distribution Circuits(view))
Hold(DSO SS(org), virtual user5(Subject), RWX(action), object-teleMCDTU(object), normal(context)).
Then is-permitted(virtual user5(Subject), RWX(action), object-teleMCDTU(object))
```

5.3.2.2 Scenario 2 : Interaction Between TSO and DSO Operators in Emergency Conditions

Figure 5.8 summarises the different web services, virtual users (representing distant users that request web services), and ws-images (local images of distant invoked web services), and resources involved in the execution of scenario 2.

A. WS1-armingrequest

- o Provider: DSO ACC.
- o Client: TSO RCC (an operator or a process).

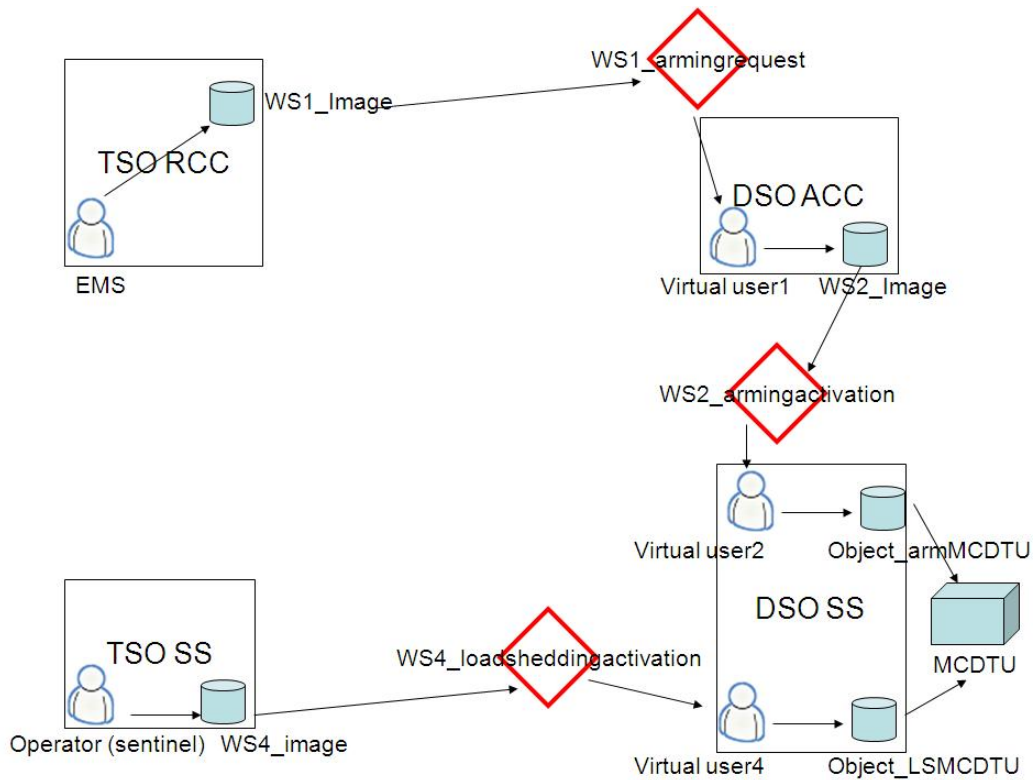


Figure 5.8: Scenario 2 users and services.

Execution Scenario:

- We assume that an EMS, an operator in TSO RCC (or a process running in TSO RCC) orders the DSO ACC to arm its DSO SS MCDTU.
- When the EMS accesses (perform arming request over) WS1-Image, the execution of WS1-armingrequest is automatically activated.
- WS1-Image is the local representation of the remote WS1-armingrequest.
- WS1-armingrequest consists in ordering the virtual user1 in DSO ACC to access (perform arming request over) object WS2-Image.
- The Virtual user1 is the representation of TSO RCC in DSO ACC. Virtual user1 is created at the negotiation phase (for WS) between TSO RCC and TSO SS.
- We recall that a virtual user is a user (temporarily or not) created to handle a remote request to a local resource.

We present here the different OrBAC predicates that manage Access Control for the Arming request web service at the level of the organisation that invokes the service (TSO RCC) and the one that provides it (DSO ACC).

OrBAC rules at TSO RCC

Permission(TSO RCC(org), TSO-role for RCC(role), Access(activity), RCC Distribution circuits(view), emergency(context))
 Empower(TSO RCC(org), EMS(Subject), TSO-role for RCC(role))
 Consider(TSO RCC(org), RWX(action), Access (activity))
 Use(TSO RCC(org), WS1-Image(object), RCC Distribution Circuits(view))
 Hold(TSO RCC(org), EMS(Subject), RWX(action), WS1-Image(object), emergency(context)).
 Then is-permitted(EMS(Subject), RWX(action), WS1-Image(object))

OrBAC rules at DSO ACC

Permission(DSO ACC(org), DSO-role for ACC(role), Access (activity), DSO ACC Distribution Circuits(view), emergency(context))
 Empower(DSO ACC(org), virtual user1(Subject), DSO-role for ACC(role))
 Consider(DSO ACC(org), RWX(action), Access (activity))
 Use(DSO ACC(org), WS2-Image(object), DSO ACC Distribution Circuits(view))
 Hold(DSO ACC(org), virtual user1(Subject), RWX(action), WS2-Image(object), emergency(context)).
 Then is-permitted(virtual user1(Subject), RWX(action), WS2-Image(object))

B. WS2-armingactivation

- o Provider: DSO SS.
- o Client: DSO ACC.

Execution Scenario:

- When virtual user1 access (performs arming activation over) WS2-Image, the execution of WS2-armingactivation is automatically activated.
- WS2-image is the local image of WS2-armingactivation.
- WS2-armingactivation consists in ordering a virtual user2 to access (perform the arming activation over) Object-armMCDTU in DSO SS.
- The Virtual user2 is the image of DSO ACC in DSO SS. Virtual user2 is created at the negotiation phase between DSO ACC and DSO SS.
- When virtual user2 access (perform arming activation over) Object-armMCDTU, the real physical command of arming activation is launched over the physical MCDTU.
- We can say that operator EMS from TSO RCC used virtual user1 and then virtual user2 as relays and performed the arming command over the DSO SS MCDTU.

We present here the different OrBAC predicates that manage Access Control for the Arming activation web service at the level of the organisation that invokes the service (DSO ACC) and the one that provides it (DSO SS).

OrBAC rules at DSO ACC

Permission(DSO ACC(org), DSO-role for ACC(role), Access (activity), DSO ACC Distribution Circuits(view), emergency(context))
 Empower(DSO ACC(org), virtual user1(Subject), DSO-role for ACC(role))
 Consider(DSO ACC(org), RWX(action), Access (activity))
 Use(DSO ACC(org), WS2-Image(object), DSO ACC Distribution Circuits(view))
 Hold(DSO ACC(org), virtual user1(Subject), RWX(action), WS2-Image (object), emergency(context)).
 Then is-permitted(virtual user1(Subject), RWX(action), WS2-Image(object))

OrBAC rules at DSO SS

Permission(DSO SS(org), DSO-role for SS(role), Access (activity), DSO SS Distribution Circuits(view), emergency(context))
 Empower(DSO SS(org), virtual user2(Subject), DSO for SS(role))
 Consider(DSO SS(org), RWX(action), Access (activity))
 Use(DSO SS(org), object-armMCDTU(object), DSO SS Distribution Circuits(view))
 Hold(DSO SS(org), virtual user2(Subject), RWX(action), object-armMCDTU(object), emergency(context)).
 Then is-permitted(virtual user2(Subject), RWX(action), object-armMCDTU (object))

D. WS4-loadsheddingactivation

- o Provider: DSO SS.
- o Client: TSO SS.

Execution Scenario:

- When Operator (Local function of sentinel) access (performs Load Shedding over) WS4-Image, the execution of WS4-loadsheddingactivation is automatically activated.
- WS4-image is the local image of WS4-loadsheddingactivation.
- WS4-loadsheddingactivation consists in ordering a virtual user4 to access (perform the load shedding over) Object-LSMCDTU in DSO SS.
- The Virtual user4 is the image of TSO SS in DSO SS. Virtual user4 is created at the negotiation phase between TSO SS and DSO SS since they are different.
- When virtual user4 access (perform load shedding over) Object-LSMCDTU, the real physical command of Load Shedding is launched over the physical MCDTU.
- Operator (Local function of sentinel) used virtual user4 as relays and performed the load shedding over the DSO SS MCDTU.

We present here the different OrBAC predicates that manage Access Control for the Load shedding request web service at the level of the organisation that invokes the service (TSO SS) and the one that provides it (DSO SS).

OrBAC rules at TSO SS

```

Operator (Local function of sentinel)
Permission(TSO SS(org), TSO-role for SS(role), Access (activity), TSO SS Distribution Circuits(view), emergency(context))
Empower(TSO SS(org), Operator (Local function of sentinel)(Subject), TSO-role for SS(role))
Consider(TSO SS(org), RW(action), Access (activity))
Use(TSO SS(org), WS4-Image(object), TSO SS Distribution Circuits(view))
Hold(TSO SS(org), Operator (Local function of sentinel)(Subject), RWX(action), WS4-Image (object), emergency(context)).
Then is-permitted(Operator (Local function of sentinel)(Subject), RWX(action), WS4-Image (object))

```

OrBAC rules at DSO SS

```

Permission(DSO SS(org), DSO-role for SS(role), Access (activity), DSO SS Distribution Circuits(view), emergency(context))
Empower(DSO SS(org), virtual user4(Subject), DSO for SS(role))
Consider(DSO SS(org), RWX(action), Access (activity))
Use(DSO SS(org), object-LSMCDTU(object), DSO SS Distribution Circuits(view))
Hold(DSO SS(org), virtual user4(Subject), RWX(action), object-LSMCDTU(object), emergency(context)).
Then is-permitted(virtual user4(Subject), RWX(action), object-LSMCDTU(object))

```

Figure 5.9 presents a sequence diagram summarising the different steps while invoking the Arming web service.

5.3.2.3 Scenario 3: Integration of DSO Operation and Maintenance Functions

Figure 5.10 summarises the different web services, virtual users, and ws-images, and resources involved in the execution of scenario 3. Note that this scenario is still under development and there will be possible evolutions of it.

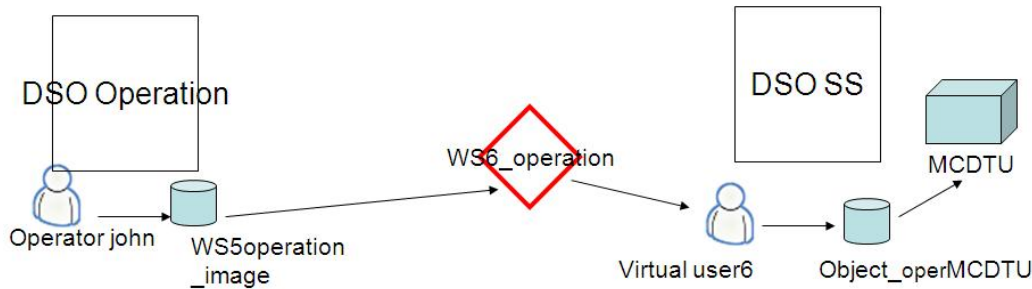


Figure 5.10: Scenario 3 users and services.

A. WS6-operation

- o Provider: DSO SS.
- o Client: DSO ACC.

Execution Scenario:

- The operator John in DSO ACC performs operation over the DSO SS.
- When operator John access (perform operation over) WS6operation-Image, the execution of WS6-operation is automatically activated.
- WS6operation-Image is the local representation of the remote WS6-operation.
- WS6-operation consists in ordering the virtual user6 in DSO SS to access (perform operation request over) object object-operMCDTU.
- The Virtual user6 is the representation of DSO ACC in DSO SS.
- Virtual user6 is created at the negotiation phase (for WS) between DSO ACC and DSO SS.

We present here the different OrBAC predicates that manage Access Control for the Operation web service at the level of the organisation that invokes the service (DSO ACC) and the one that provides it (DSO SS).

OrBAC rules at DSO ACC

```

Permission(DSO ACC(org), DSO-role for ACC(role), Access(activity),ACC Distribution circuits(view), normal(context) )
Empower(DSO ACC(org), John(Subject), DSO-role for ACC(role))
Consider(DSO ACC(org), RWX(action), Access (activity))
Use(DSO ACC(org), WS6operation-Image(object), ACC Distribution Circuits(view))
Hold(DSO ACC(org), John(Subject), RWX(action), WS6operation-Image(object), emergency(context)).
Then is-permitted(John(Subject), RWX(action), WS6operation-Image(object))

```

OrBAC rules at DSO SS

```

Permission(DSO SS(org), DSO-role for SS(role), Access (activity), DSO SS Distribution Circuits(view), normal(context))
Empower(DSO SS(org), virtual user6(Subject), DSO for SS(role))
Consider(DSO SS(org), RWX(action), Access (activity))
Use(DSO SS(org), object-operMCDTU(object), DSO SS Distribution Circuits(view))
Hold(DSO SS(org), virtual user6(Subject), RWX(action), object-operMCDTU(object), normal(context)).
Then is-permitted(virtual user6(Subject), RWX(action), object-operMCDTU(object))

```

5.3.2.4 Scenario 4: Maintenance of ICT Components of DSO

Figure 5.11 summarises the different web services, virtual users, and ws-images, and resources involved in the execution of scenario 4. Note that this scenario is still under development and there will be possible evolutions of it.

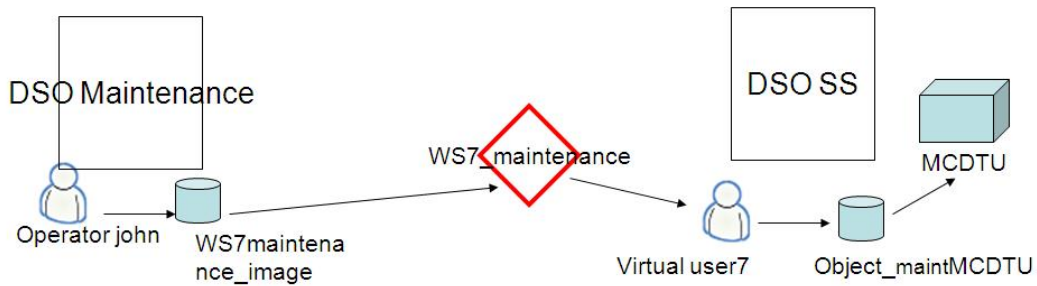


Figure 5.11: Scenario 4 users and services.

WS1-maintenance

- o Provider: DSO SS.
- o Client: DSO ACC.

Execution Scenario:

- The operator John in DSO ACC performs operation over the DSO SS.
- When operator John access (perform maintenance over) WS7 maintenance-Image, the execution of WS7-maintenance is automatically activated.
- WS7 maintenance-Image is the local representation of the remote WS7-maintenance.

- WS7- maintenance consists in ordering the virtual user7 in DSO SS to access (perform maintenance request over) object object-maintMCDTU.
- The Virtual user7 is the representation of DSO ACC in DSO SS.
- Virtual user7 is created at the negotiation phase (for WS) between DSO ACC and DSO SS.

We present here the different OrBAC predicates that manage Access Control for the Maintenance web service at the level of the organisation that invokes the service (DSO ACC) and the one that provides it (DSO SS).

OrBAC rules at DSO ACC

```
Permission(DSO ACC(org), DSO-role for ACC(role), Access(activity), ACC Distribution circuits(view), normal(context) )
Empower(DSO ACC(org), John(Subject), DSO-role for ACC(role))
Consider(DSO ACC(org), RWX(action), Access (activity))
Use(DSO ACC(org), WS7maintenance-Image(object), ACC Distribution Circuits(view))
Hold(DSO ACC(org), John(Subject), RWX(action), WS7maintenance-Image(object), emergency(context)).
Then is-permitted(John(Subject), RWX(action), WS7maintenance-Image(object))
```

OrBAC rules at DSO SS

```
Permission(DSO SS(org), DSO-role for SS(role), Access (activity), DSO SS Distribution Circuits(view), normal(context))
Empower(DSO SS(org), virtual user7(Subject), DSO for SS(role))
Consider(DSO SS(org), RWX(action), Access (activity))
Use(DSO SS(org), object-maintMCDTU(object), DSO SS Distribution Circuits(view))
Hold(DSO SS(org), virtual user7(Subject), RWX(action), object-maintMCDTU(object), normal(context)).
Then is-permitted(virtual user7(Subject), RWX(action), object-maintMCDTU(object))
```

Figure 5.12 presents a sequence diagram summarising the different steps while invoking the Teleoperation (Operation, or Maintenance) web service.

5.4 *Monitoring and Failure Detection*

This section contains a preliminary implementation of the monitoring and failure detection services described in Section 3.4. From a system-level viewpoint, monitoring and failure detection activities are organized as follows:

- Every CIS diagnoses itself over time, in order to judge the healthy/faulty status of its resources/services, primarily for local reconfiguration aims. This step is the result of the activity of the “CIS Self-Diagnosis” service.
- Every CIS (when asked for) declares its “health status” (full report, only relevant parts or a signature of them). Since CISs are not completely trusted by the others, they do not completely trust the declared “health status”, so they also try to build a private perception of the other CISs, basing on the possible direct relationships with them. The “declared status” and the “perceived statuses” could be conflicting (e.g. because of communication problems or because of deliberate and malicious causes). When CIS A needs to use some remote resources/services on CIS C, but A has no private perception of C, gossip can be applied: if A trusts B and B has a private perception of C, the private perception of C as seen by A can inherit the private perception of C as seen by B.
- When necessary, e.g. when the private perception is not enough for some reason, (pertinent groups of) CISs exchange among them their own private perceptions of a certain resource, in order to reach an agreement about that. The result of the agreement overrides the result of the private diagnosis.

5.4.1 *Design Rationale*

In the lifetime of a specific instance of a system, a rich flux of information relative to diverse aspects of the ongoing activities comes from a number of sources: hardware sensors (voltages, currents, power, temperatures, fan speeds, vibrations, accelerations); operating system messages (positive acknowledges, completion codes, warnings, various levels of exceptions, etc); communication subsystem messages (status, errors, timing, etc); application messages (normal exits, acquire/release system resources, interface errors, internal captured errors, etc.). Messages can be category-labeled accordingly; each category shows up as a unevenly spaced time series of data (whether sensor readings or more complex messages). The collection of all these time- and type-ordered data amounts to the innards’ behavior of the system.

However, the time series are essentially unlimited. More workable structures can be constructed as sets of time-limited sub-series, built taking care of choosing time cuts distant enough to capture significant behavior chunks.

A set $\{SG\}$ of such rough chunks can be set up initially from system logs, then kept up-to-date by slicing the behavior’s information flux.

The present design of an evolutive diagnosis subsystem is based on the following structures, built over the innard's behavior of the target system:

- A set of specification abiding behaviors $\{G\}$ where each element is an element taken from $\{SG\}$, possibly enriched with relevant input-output sequences, input values - internal state tuples, internal state evolution (partial) sequences, etc.; the element being characterized by being positively abiding by the specs.
- A dynamic collection of unsure behaviors $\{B\}$ where items from $\{SG\}$ not found in $\{G\}$ are collected on the run.
- A dynamic aggregation matrix $\{C\}$, where elements from $\{B\}$ are grouped, according to a lumping strategy based on a number of adaptable criteria (e.g., timing correlation, external stimuli presence, temperature, time of the day, etc.).

The above information structures are operated upon by specialized processes:

- A Collector process (or a concurrent set thereof) gathers the myriad of low-level (and high-level as well) lively signals from the system, updating mainly the set $\{B\}$ (but also the set $\{G\}$, upon e.g. system maintenance/update);
- Recognizer process(es) similarly act upon the set $\{C\}$, looking for patterns (associations) already known to be alarming symptoms, and building new ones whenever a "higher level" error signal overrides this background detection activity. A few available techniques for pattern recognition over non-periodic time series are under scrutiny.

The "higher level" implies a hierarchy in the misbehavior signals: in fact, elements in $\{B\}$ should be intended as deviating patterns not severe enough to require immediate action; it is common sense that situations do arise bad enough to require corrective action, without much pondering. In such a case, the current chunk is added to the accumulated knowledge of bad situations, after being properly formatted and labeled.

5.4.2 Services Specification

The services interface specification is common both in the local view and in the global view; type/values do differ in each instance.

Collector: The collector service collects detection information (errors) from a specific service/-component. The detection information is collected following a specified interaction paradigm (e.g. proactively, reactively or event triggered); the collected information complies with a specified format and semantics. The interaction paradigm, the format and the semantics of the collected information all depend on the service/component.

The service detects whether the interaction pattern is violated.

Parameters:

- name: id of the monitored service/component
- interaction paradigm: this is the way the collector collects detection information (proactive, reactive, periodic, other)
- detection format: format of the collected detection information
- detection semantics: semantics of the collected detection information

Normalizer . The normalizer translates the detection information collected by a “Collector” in a normalized format specified by the parameters.

The normalizer is used by the “Aggregator” as a source of normalized information.

Parameters:

- name: id of the collector providing non normalized detection information
- interaction paradigm: this is the way the collector produces information to be normalized
- detection format: format of the detection information received in input
- detection semantics: semantics of the detection information received in input
- normalized format: format of the normalized detection information
- normalized semantics: semantics of the normalized detection information

Specification:

The “Normalizer” translates the information collected by the “Collector” following the specified format and semantics for both input and output. The translation function must be injective (possibly biunique) in order to avoid ambiguity in the normalization.

The service detects whether the interaction pattern is violated and whether the format of the received information is violated.

Aggregator: The aggregator service assesses the status of a monitored component; the aggregator applies a diagnostic function to normalized detection information gathered overtime from a specific component/service. Several diagnostic functions can be used (heuristic [10], probabilistic [25]).

Parameters:

- name: id of the normalized collector providing detection information to the aggregator
- interaction paradigm: this is the way the collector produces information to be gathered
- normalized format: format of the normalized detection information
- normalized semantics: semantics of the normalized detection information
- diagnostic function: function applied to the flow of normalized signals received as input by the aggregator
- alarms: specifications of the alarms triggered by the diagnostic function.

Specification:

The “Aggregator” filters the detection information normalized by the “Normalizer” using the specified diagnostic function.

The service detects whether the interaction pattern is violated and whether the format of the received information is violated.

Recognizer: The recognizer correlates aggregated detection information coming from several monitored components/services. The recognizer is the diagnosing entity at system level.

The Recognizer process assembles its results in messages with the following tentative format:

- <TYP> category of the message (e.g. fault/error/warning)
- <LVL> signal level (one dimensional) - from “debug” to “emergency” [74]
- <TIME> timestamp
- <LOC> physical localization (generally, down to the least replaceable unit)
- <SYS> information on the Operating System status or affected components
- <APP> the co-interested Application software
- <SPN> special annotations, local system instance dependent

where <LVL> gives a measure of the severity of the error reported, and thus it is actually the first field to be parsed, in case immediate action were to be taken. The format proposed in [2], the Universal Logger Messages, was not adopted because it carries too little information; on the other hand, the full complexity of the BSD Syslog Protocol described in [74] is not needed, and actually detrimental, in this peculiar context.

Correlation is performed at various levels (e.g. signal severity); correlation is flexible with respect to the severity of the collected signals: severe signals are quickly recognized and trigger appropriate notifications to the reconfiguration sub-system.

5.4.3 Diagnosing the Protection Service

The Protection Service (PS) (see also section 3.3.1) is a middleware service performing egress/ingress access control, implementing an instance of the global security policy. The PS captures packets that pass through the CIS, checking if these packets satisfy the security policy and forwarding to the destination node only packets satisfying the security policy (discarding the others).

The dependability of the PS is currently enhanced using the CIS Proactive-Reactive Recovery Service [9], taking advantage of the CIS replicated hybrid architecture.

Monitoring is performed at payload level, where each instance of the protection service checks whether other replicas behave correctly, triggering specific accusations when necessary. The following function calls are used by replica *i* to express accusations about replica *j*:

- “W_detect(j)”: replica i detects that replica j is faulty. This is the case in which replica i finds an illegal message coming from replica j ;
- “W_suspect(j)”: replica i suspects that replica j is faulty. This is the case in which replica i finds that replica j , being the leader, is not forwarding a legal message to the protected LAN.

Failure detection is performed at wormhole level, where accusations raised by the replicas are collected and interpreted on the basis of quorums:

- replica j is detected to be maliciously faulty if at least $f + 1$ “W_detect(j)” were collected; in this case, at least one correct replicas detected replica j to be maliciously faulty.
- replica j is suspected of being faulty if at least $f + 1$ accusations were collected; in this case, at least one correct replica raised a suspect about replica j being faulty.

A quantitative analysis of the recovery strategy was performed (more details can be found in [35]). The main results of the analysis are presented hereafter, in order to identify the recovery strategy weak points and open problems and propose enhancements.

5.4.3.1 PRRW Quantitative Analysis

This section presents a quantitative analysis of the PRRW strategy. The relevant figures of interest are identified and the relevant parameters are described; finally the results of the performed simulations are presented and discussed. Details about models and simulations are available in [35].

The quantitative analysis of the PRRW strategy aims to evaluate how effective is the compromise between proactive and reactive recovery actions. Proactive recoveries rejuvenate the replicas in predefined instants of time, without being based on any fault detection. This means that proactive recoveries treat all the faults, including also the latent and hidden ones, which cannot be treated in other way, but they recover also correct replicas, weakening the availability of the system. On the other side, reactive recoveries are scheduled only on replicas detected or suspected of being faulty; replicas not detected of being faulty are never recovered, even if they are actually faulty, weakening the dependability of the system. Acting on the detection/diagnosis aspects of the system, we impact on the reactive part and we are hence able to quantify the benefit obtained by the two kind of recovery.

The relevant measures of interest are system failure probability and system unavailability. The system fails at time t if one of the following conditions holds:

1. the number of invalid replicas gets over f ;
2. the system is unavailable since $t - T_O$, that is the system is unavailable for a period of time grater then T_O .

Let $P_{FI}(t)$ be the probability of the system being failed at time t because of condition 1, given that it was correctly functioning at time $t=0$. Let $P_{FO}(t)$ be the probability of the system being failed at time t because of condition 2, given that it was correctly functioning at time $t=0$. System failure probability, denoted by $P_F(t)$, is defined as the probability of the system being failed at time t , given that it was not failed at time $t=0$; system failure probability is obtained as

$$P_F(t) = P_{FI}(t) + P_{FO}(t).$$

The system is unavailable at time t if one of the following conditions holds:

1. the number of correct replicas is less than $f+1$ (quorums cannot be reached)
2. there are more than $f+1$ correct replicas, but the leader is omitting (legal messages are not forwarded).

Let $T_U(0, t)$ be the total time the system is not failed but is unavailable within $[0, t]$ because of one of the above conditions. Let $T_A(0, t)$ be the total time the system is not failed within $[0, t]$. System unavailability, denoted by $P_U(t)$, is defined as the probability of the system being unavailable at time t , given that it was correctly working at time $t=0$; system unavailability is obtained as

$$P_U(t) = T_U(0, t)/T_A(0, t).$$

All the model parameters and the basic values used for the evaluations are shown in table 5.4. The relevant parameter used are the following:

- Mission time t . This is the time during which the system is exercised since it starts to work. t varies in $[2628, 42048]$.
- Detection coverage c_M of malicious behavior. c_M is the probability of detecting an intruded replica, and hence the probability of reactively recovery an intruded replica. c_M varies in $[0, 1]$: if $c_M=0$, no intrusions are detected, whilst if $c_M=1$, all intrusions are detected. $c_M=1$ in the ideal case.
- Probability p_I of intrusion manifesting as a permanent invalid behavior. p_I varies in $[0, 1]$: if $p_I=0$, all intrusions manifest as a permanent omissive behavior; if $p_I=1$, all intrusions manifest as a permanent invalid behavior.
- Number n of system replicas in the system, maximum number f of corrupted replicas tolerated by the system itself and maximum number k of system replicas recovering simultaneously.

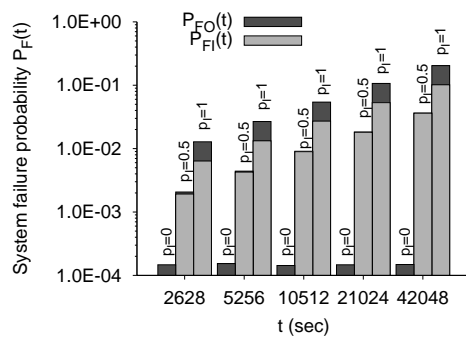
A first study was performed observing both system failure probability $P_F(t)$ and system unavailability $P_U(t)$ over mission time t . Three system configurations were evaluated for three different values of p_I (probability of intrusions manifesting as permanent invalid behavior).

Figure 5.13 shows how $P_{FI}(t)$ and $P_{FO}(t)$ change over mission time t . If $p_I=0$, $P_F(t)$ is constant as time increases (in this case $P_F(t)=P_{FO}(t)$, given that $P_{FI}(t)=0$). If $p_I=0.5$, $P_F(t)$ increases

Table 5.4: Parameters and their default values

name	value	meaning
t	2628	Mission time (sec)
n	4	Number of replicas in the system
k	1	Max number of replicas recovering simultaneously
f	1	Max number of corrupted replicas tolerated by the system
T_D	146	Time duration of a recovery operation (sec)
T_O	60	Duration of system omission before considering the system failed (sec)
λ_{ci}	$[1.9e^{-7}, 3.8e^{-7}]$	Crash rate, exponentially distributed. Each replica has a diverse crash rate (from 1 per 60 days to 1 per 30 days)
λ_{oi}	$[1.9e^{-6}, 3.8e^{-6}]$	Transient omission rate, exponentially distributed. Each replica has a diverse rate (from 1 per 6 days to 1 per 3 days)
λ_{eo}	$3.3e^{-2}$	Omission duration rate, exponentially distributed. A transient omission lasts about for 30 seconds
λ_{ai}	$[5.8e^{-5}, 1.2e^{-5}]$	Successful attack (intrusion) rate, exponentially distributed. Each replica has a diverse rate (from 5 per day to 1 per day)
p_I	0.5	Probability of intrusion manifesting as a permanent invalid behavior (if $p_I=0$ all intrusions manifest as permanent omissions)
c_M	0.7	Probability of detecting a malicious behavior

as time increases; the figure shows that $P_{FO}(t)$ is negligible with regard to $P_{FI}(t)$ ($P_{FO}(t)$ decreases to 0 as time increases). If $p_I=1$, $P_F(t)$ increases as time increases; the figure shows that $P_{FO}(t)$ is constant over time and that $P_{FI}(t)$ increases over time. The system with $p_I=0$ has the smallest values for $P_F(t)$ among the three system considered, whilst the system with $p_I=1$ has the largest values for $P_F(t)$ (at least two orders of magnitude with regard to the system with $p_I=0$).

Figure 5.13: System failure probability $P_F(t)$ over mission time t for different values of p_I .

If $p_I=0$, there isn't any invalid behavior triggering reactive recoveries, so almost all recoveries are periodic (reactive recoveries are triggered only on an omissive leader): in this case $P_F(t)$ shows to be almost constant over time. If $p_I \in \{0.5, 1\}$, there are invalid behaviors triggering reactive recoveries: in both the cases $P_F(t)$ shows to be increasing overtime mainly because of $P_{FI}(t)$; the larger is p_I , the larger is $P_F(t)$. In general it turns out that $P_F(t)$ increases over time mainly

because of the reactive recoveries triggered by invalid behavior.

Figure 5.14 shows how $P_U(t)$ changes over mission time t . If $p_I \in \{0, 0.5\}$, $P_U(t)$ decreases as time increases (both the settings show the same values for the same t). If $p_I=1$, $P_U(t)$ shows to be constant with regard to time; in this case $P_U(t)$ is larger than for the other values of p_I .

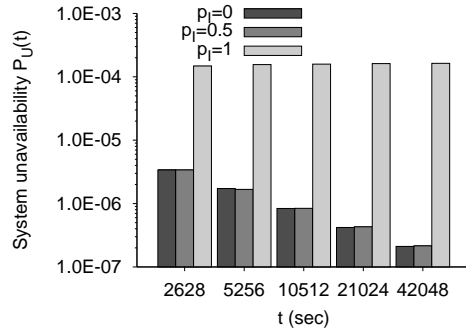


Figure 5.14: System unavailability $P_U(t)$ over mission time t for different values of p_I .

$P_U(t)$ is decreasing over time (at most is constant), so it is satisfactory in all the system configurations considered; the positive effect of proactive recoveries refreshing all the replicas is evident mostly for $p_I=0$, whereas the effect is smaller (but still effective) as p_I increases, since $P_U(t)$ is not increasing over time.

Another study was devoted to evaluate both system failure probability $P_F(t)$ and system unavailability $P_U(t)$ varying both the detection coverage c_M and the probability p_I of intrusions manifesting as invalid behavior. Varying the value of C_M we act on the mechanism of reactive recoveries in the following way: increasing the detection coverage, it increases also the number of invalid replicas detected to be invalid and hence reactively recovered. If $C_M=0$, reactive recoveries are triggered only on an omissive leader.

Figure 5.15 and 5.16 show respectively how $P_{FI}(t)$ and $P_{FO}(t)$ change over detection coverage C_M for some values of p_I . The two figures are plotted using the same scale for the y-axis in order to make easier their comparison. In general, $P_{FI}(t)$ decreases as C_M increases from 0 to 1. The extreme curves correspond to the two extreme system configurations: $p_I=1$, assuming the largest values, and $p_I=0$, assuming the lowest values: in this last case $P_{FI}(t)=0$ (the curve is out of the bounds of the figure). The curve showed in Figure 5.17 which assumes the smallest values corresponds to the system configuration where $p_I=0.2$. The curve corresponding to $p_I=1$ decreases quicker than the other curves (it decreases for about one order of magnitude), whilst the curve for $p_I=0.2$ is almost constant. $P_{FO}(t)$ shows an opposite behavior with respect to $P_{FI}(t)$: it increases as C_M increases from 0 to 1. The extreme curves correspond to the two extreme system configurations: $p_I=0$, assuming the largest values, and $p_I=1$, assuming the lowest values (the opposite behavior with respect to $P_{FI}(t)$). The curve corresponding to $p_I=1$ increases quicker than the other curves, whilst the curve for $p_I=0$ is almost constant.

Figure 5.15 and 5.16 show that increasing C_M there are two opposite effects with respect to $P_{FI}(t)$ and $P_{FO}(t)$: the former decreases, because invalid replicas reactively recovered are no longer weakening the system; the latter increases, because replicas, while recovering, do not contribute

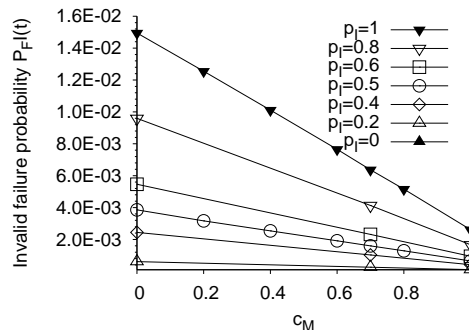


Figure 5.15: Impact of detection coverage c_M on failure probability $P_F(t)$ due to invalid behavior for different values of p_I .

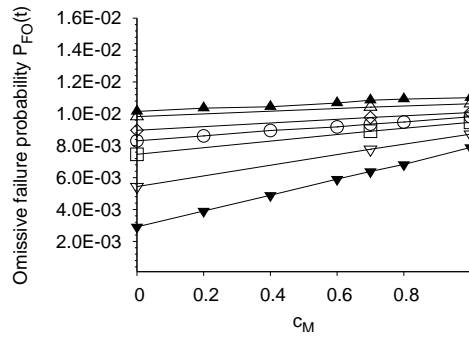


Figure 5.16: Impact of detection coverage c_M on failure probability $P_{FO}(t)$ due to omissive behavior for different values of p_I .

to system operation. The overall effect, shown in figure 5.17, is that system failure probability decreases as detection coverage C_M increases. The system was evaluated in this study for $t=2628$; the first study described above showed that system failure probability $P_F(t)$ increases over time (except for system configuration where $p_I=0$). We hence suppose that this study, if evaluated for a larger value of t , should show a larger difference between the extreme system configurations $p_I=0$ and $p_I=1$. This stresses that the value for C_M should be as higher as possible.

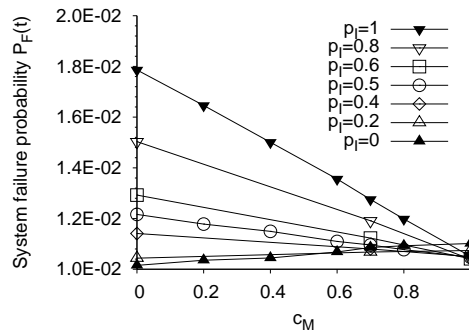


Figure 5.17: Impact of detection coverage c_M on system failure probability $P_F(t)$ for different values of p_I .

Figure 5.18 shows how system unavailability $P_U(t)$ changes over detection coverage C_M

for some values of p_I . In general, $P_U(t)$ increases as C_M increases from 0 to 1. The extreme curves correspond to the two extreme system configurations: $p_I=0$, assuming the largest values, and $p_I=1$, assuming the lowest values. If $p_I=0$, $P_U(t)$ is almost not influenced by changing the detection coverage, whilst increasing p_I the influence of C_M becomes more evident (almost an order of magnitude for $p_I=1$).

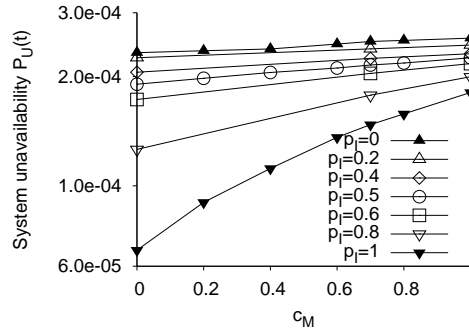


Figure 5.18: Impact of detection coverage c_M on system unavailability $P_U(t)$ for different values of p_I .

It turns out that there are two conflicting effects on system unavailability related to the increasing number of reactive recoveries. $P_U(t)$ is negatively affected by a larger value for the detection coverage c_M , because the larger is the detection coverage, the more reactive recoveries are triggered; the above trend is more evident as the probability p_I of intrusion manifesting as invalid behavior increases, because reactive recoveries are mainly triggered because of invalid behavior. On the other side, it has to be noticed that the smaller is p_I , the worse is system unavailability, because less reactive recoveries are triggered on faulty replicas.

The results of this study shows that increasing the detection coverage of invalid behavior has conflicting effects on system failure probability and system availability: the former improves as c_M increases, whilst the latter worsen as c_M increases.

The last study performed aimed to evaluate the impact of the number of replicas on both system failure probability and unavailability. When dealing with the number of replicas in the system, three parameters are relevant: n , the overall number of replicas in the system, f , the maximum number of corrupted replicas tolerated by the system and k , the maximum number of replicas simultaneously recovering without endangering the availability of the system. The values of the above parameters are disciplined by the following formula: $n=2f+1+k$. The following system configurations were evaluated:

1. $n=4, f=1, k=1$
2. $n=5, f=1, k=2$
3. $n=6, f=1, k=3$
4. $n=6, f=2, k=1$

Figure 5.19 and 5.20 show system failure probability $P_F(t)$ (decomposed in $P_{FI}(t)$ and $P_{FO}(t)$) and system unavailability $P_U(t)$ for the system configurations described above. In general $P_F(t)$ decreases as the number n of system replicas increases (this trend is due primarily to $P_{FO}(t)$); for the same value of n , the higher is k and the lower is $P_F(t)$. In general $P_U(t)$ decreases as the number n of system replicas increases; for the same value of n , the higher is k and the lower is $P_U(t)$.

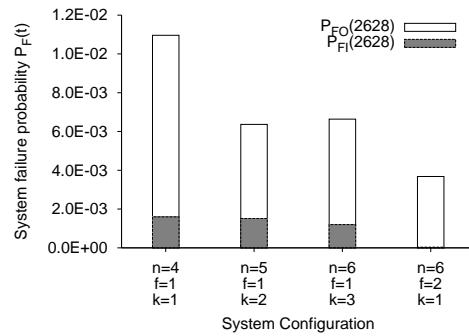


Figure 5.19: System failure probability $P_F(t)$ for different system configurations at mission time $t=2628$.

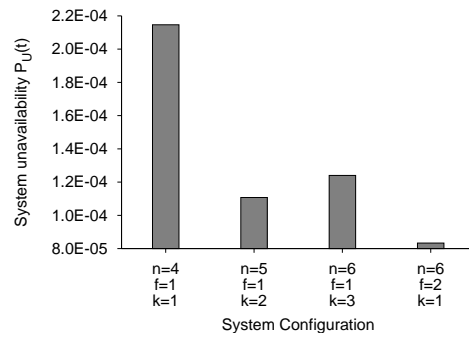


Figure 5.20: System unavailability $P_U(t)$ for different system configurations at mission time $t=2628$.

It turns out that for the setting used (as shown in table 5.4) the lower system failure probability and the lower system unavailability is measured in the system configuration 3; this configurations has the highest number n of replicas with regard to the other configurations evaluated. Configuration 4 has also the same n as configuration 3, but it has a lower value for k ($P_{FO}(t)$ is the main contributor to $P_F(t)$).

5.4.4 Advantages and Limits of the PRRW Strategy

The CIS intrusion tolerance is currently obtained through a recovery strategy (PRRW) based on a combination of proactive and reactive recoveries. The use of both proactive and reactive recoveries shows to be effective since the two techniques possess complementary characteristics.

Proactive recoveries periodically rejuvenate all the replicas, without any need of fault detection mechanisms (also latent/hidden faults are treated). The period of the proactive recoveries define a bounded temporal window (between two recoveries of the same replica) which represent a time limit for an attack attempt to be successful. In fact, this is the time an attacker has for conquering a majority of the replicas and thus for taking the control of the entire CIS. On the other side, being an “unconditional” recovery, the proactive recovery is applied also to correct replicas which become non available for the time necessary to perform the recovery. Moreover, if only proactive recovery is used in a system, a replica hit by a fault will be unavailable until the end of its next proactive recovery.

Reactive recovery is instead triggered only on detection of faults to have hit a replica, so its effectiveness depends on the assumed fault model and on the coverage of the detection mechanism used (latent/hidden faults are not treated). Reactive recoveries triggered on replicas detected of being faulty contribute to decrease system failure probability, as shown in Figure 5.17; reactive recoveries are in fact performed as soon as possible, however within the duration of $[f/k]$ recoveries, without waiting the next periodic recovery on the same replica. Being the replica reactively recovered as soon as possible, its rejuvenation is anticipated with respect to its next proactive recovery, so the (faulty) replica becomes active and correct earlier.

This behavior apparently suggests that the more reactive recoveries are performed, the worst is system availability as it appears evidently in Figure 5.13 and 5.14 for the curve corresponding to $p_I=1$: in this case all the intrusions manifest as invalid behavior and all the detected intrusions trigger a reactive recovery (detection coverage c_M is set to 0.7 as default). In reality what happens is that the system ability to survive gets increased, whereas for low values of the coverage (thus less reactive recoveries) the system fails as soon as replicas get affected by faults.

The PRRW strategy, as our analysis reveals, makes a significant difference in the way omission and invalid behaviors are treated. This is made evident by observing all the curves at varying values for p_I . Actually invalid behaviors are detected with a coverage c_M and trigger a reactive recovery whereas omissive behaviors currently are essentially not detected: only the omissive leader is detected and triggers some action, omissive followers are cured only with the proactive recovery. Increasing the capability to detect (and quickly react) to omissive behaviors is a way to improve the overall fault tolerance strategy

5.4.5 Direction for Improvements/Refinements

This section identifies the directions for refining and enhancing the CIS architecture. An extended fault model is introduced and some modifications to the recovery schemes are presented.

5.4.5.1 New Fault Model

The PRRW strategy is based on distinguishing and detecting a limited set of faults and does not consider some subtle faults that can occur in a replica (these faults are however treated thanks to the strategy of the proactive recoveries). All these faults are discussed hereafter, with the objective of showing that detecting such faults and treating them using reactive recoveries would improve both system dependability and availability.

In the PRRW strategy, the correct replicas detect the following faults:

- **Leader Benign Fault (LBF):** The faulty leader omits to send a signed message to the LAN. A correct replica will suspect the leader to be “silent” after O_t consecutive leader omissions on the same signed message.
- **Replica Malicious Fault (RMF):** The faulty replica (being it either the leader or a follower) sends a not signed message to the LAN; a correct replica will immediately detect the faulty replica to be a “malicious sender”.

It comes out that the PRRW schema takes into account both omissive and malicious faults in the leader replica, but only malicious faults in the follower replicas. The idea is that if a follower is going to have an omissive behavior, the problem will be eventually treated either by the proactive recovery or by the election of the replica as a leader (the replica will be extensively monitored in this case). In both cases, the negative effects of the faults will be eventually eliminated.

The subtle faults not considered by the current reactive recovery proposal are the following:

- **Malicious Approval (MA):** A faulty replica approves an illegal message; the faulty replica is intruded, because all correct replicas verify the same security policy.
- **Omitted Approval (OA):** A faulty replica omits to approve a legal message; the omission could be caused by communication problems (the replica never received the legal message), but it could be the effect of an intrusion.
- **Malicious Suspect (MS):** A faulty replica signals the wormhole an accusation about a correct replica; the faulty replica is intruded, because a correct replica does not show any incorrect behavior.
- **Omitted Suspect (OS):** A faulty replica does not signal the wormhole any accusation about a faulty replica; the omission could be caused by communication problems (the replica never received the legal message), but it could be still the effect of an intrusion.

In the MA and MS cases, the faulty replica is intruded, so it needs to be recovered as soon as possible; if the faulty replica is not detected as such, it is still considered correct! In the OA and OS cases, faults could be caused either by communication omissions (no recovery action is useful

to solve the problem) or as an effect of intrusions manifesting as omissive behavior (a recovery action could solve the problem). In all the cases, the proper detection and diagnosis of all the above faults improves both dependability and availability of the system.

5.4.6 Architectural Modifications for the Detection of the Extended Set of Fault

This Section describes the architecture modifications necessary to detect the faults described in 5.4.5.1 and trigger more reactive recoveries. In order to perform the detection of the above faults it is necessary to allow each payload replica to be informed about all the approval results and manifested suspects taken by all the other payload replicas.

A shared virtual memory (SVM) mechanism can be implemented as a reliable repository where each replica posts all its approval results and suspects; a majority of correct replicas is thus able to identify which replicas took the wrong approval decisions (if any) or manifested the wrong suspect (if any).

Approval results are stored for each incoming message as a data structure containing i) an identification for the incoming message m , ii) the approval decisions collected from all the replicas about m , iii) the final vote given by the wormhole about m . Suspects are stored as a data structure containing the suspecter(s), the suspected and the kind of suspect. Information is stored in the shared virtual memory, using it as a circular buffer in order to make room for newer information; therefore the SVM is used as a queue of dimension q . If the information to be broadcasted should be too heavy to be managed through the wormhole, some form of “compression” can be found.

Each message is identified using its MAC. Each approval decisions is stored in an array of n elements, where the i^{th} element represents approval result of replica i about message m :

- “ACCEPT”: replica i approves m ;
- “REJECT”: replica i does not approve m ;
- “null”: no approval information still received from replica i about m ;
- “recovering”: replica i is currently recovering.

The final vote can be one of the following: “LEGAL”, “ILLEGAL” and “VOTING”.

The follower payload behavior is monitored as follows. When message m comes from the WAN, each replica decides whether approving it or not, posting the final decision in the SVM. Not all the replicas will receive m in the same instant, and each replica will need some time in order to take the approval decision and post it in the repository, but a certain number of approval results about m will be available in the SVM at worst within T_{vote} time after the first post.

Replicas that did not take any approval result till that moment and that were not recovering (those corresponding to the “null” array elements) will be suspected of omission (they could not have received m because of communication faults or they could have omit maliciously).

Given the final vote about m , all the correct replicas (i.e. all the replicas which approval result is in agreement with the final vote) will be able to identify all the faulty ones (i.e. all the replicas which approval result is in disagreement with the final vote) and suspect them as malicious faulty replicas.

5.4.6.1 The Priority-Queue Recovery Schema

A recovery schema based on the mechanism of a priority queue (schema PQ) is proposed. The idea of managing recovery actions using priorities comes from the PRRW strategy, where reactive recoveries triggered by a quorum of “detections” have a higher priority than recoveries triggered by some a quorum of accusations (some “suspects” and some other “detections”).

The PQ schema manages the recovery actions (both proactive and reactive) on the basis of a severity level assigned to each recovery action. Reactive recoveries receive a priority level depending on the diagnosis of the affected replica: the more severe is the diagnosed fault affecting the replica, the higher is the priority of the recovery action.

Reactive recoveries have higher priority than proactive recoveries, because postponing a proactive recovery of a replica that is (seems) correct is less hazardous than postponing a reactive recovery triggered by some suspect (or certainty) on a misbehaving replica! In order to guarantee some “progress” to proactive actions, the longer a proactive recovery remains in the queue, the higher its priority has to become.

A new leader is elected either when the current leader is diagnosed non correct or when a proactive recovery is going to be performed on it. The new leader is the last replica recovered.

6 Conclusions

In this report, we described the preliminary specification of services and protocols for the Crutial Architecture. The Crutial Architecture definition, first addressed in Crutial Project Technical Report D4 (January 2007), intends to reply to a grand challenge of computer science and control engineering: how to achieve resilience of critical information infrastructures, in particular in the electrical sector. We believe that we have given important answers to the problem at architectural level. The Crutial architecture, services and protocols that we preliminarily define in these two reports are expected to make a contribution to a reference architecture for CII in general, not only electrical, but also gas or water, or telecommunication systems and computer networks like the Internet.

Given the complexity of current and future CII, ensuring acceptable levels of service and, in last resort, the integrity of systems themselves, when faced with threats of several kinds and possibly not completely defined, requires innovative approaches. Two key characteristics of any approach should in our opinion be: **automatic** and **adaptive**. Furthermore, any successful architecture will have to take into account the **hybrid** composition of modern critical information infrastructures, amongst SCADA, corporate and Internet access parts.

We hope to have shown in this report that we are indeed providing answers to meet these objectives, in the form of the preliminary specification of services and protocols for the Crutial Architecture: (i) the Runtime Support Services and APIs, and the Middleware Services and APIs; (ii) the Runtime Support Protocols, and Middleware Services Protocols.

Bibliography

- [1] Napster. <http://www.napster.com>. visited in August 2007.
- [2] J. Abela. Universal format for logger messages. <http://www.hsc.fr/ressources/normalisation/ulm/draft-abela-ulm-05.txt>.
- [3] K. Aberer and M. Hauswirth. An overview on peer-to-peer information systems. In *Proceedings of the Workshop on Distributed Data and Structures (WDAS'02)*, 2002.
- [4] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *Proceedings of 4th IEEE International Workshop on Policies for Distributed Systems and Networks – Policy'03*, June 2003.
- [5] A. Adya and et al. Frasite: Federated, availability, and reliable storage for an incompletely trusted environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, pages 1–14. USENIX Association, 2002.
- [6] M. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, 2000.
- [7] R. Baldoni, J.-M. Hélary, M. Raynal, and L. Tangui. Consensus in Byzantine asynchronous systems. *J. Discrete Algorithms*, 1(2):185–210, April 2003.
- [8] S. Bellovin. Distributed firewalls. ;login: *The USENIX Magazine*, November 1999.
- [9] A. Bessani, P. Sousa, M. Correia, N. Neves, and P. Verssimo. Intrusion-tolerant protection for critical infrastructures. DI/FCUL TR 07–8, Department of Informatics, University of Lisbon, April 2007.
- [10] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49(3):230–245, 2000.
- [11] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$ -resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, August 1984.
- [12] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$ -resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, August 1984.
- [13] A. Burns and A. Wellings. *Real-time systems and programming languages*. Addison Wesley, Third Edition, 2001.
- [14] E. Byres and J. Lowe. The myths and facts behind cyber security risks for industrial control systems. In *Proceedings of the VDE Kongress*, 2004.

- [15] C. Cachin and J. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. of the Int. Conf. on Dependable Systems and Networks - DSN 2002*, pages 167–176, June 2002.
- [16] C. Cachin and A. Samar. Secure distributed DNS. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 423–432, 2004.
- [17] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.
- [18] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), March 1996.
- [19] Y. Chawathe, S. Ratnasamy, and L. Breslau. Making gnutella like p2p systems scalable. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*, pages 25–29, 2003.
- [20] S. Chen and R. Chow. A new perspective in defending against ddos. In *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*, 2004.
- [21] M. Chow and Y. Tipsuwan. Network-based control systems: a tutorial. In *Proceeding of the IEEE Industrial Electronics S0c., Denver, CO*, pages 1593–1602, 2001.
- [22] M. Chrc, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active replication in Delta-4. In *Proceedings of the 22th Symposium on Fault-Tolerant Computing*, pages 28–37, July 1992.
- [23] M. Correia, N. Neves, and P. Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 174–183, October 2004.
- [24] M. Correia, N. F. Neves, and P. Verissimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 41(1):82–96, January 2006.
- [25] A. Daidone, F. Di Giandomenico, A. Bondavalli, and S. Chiaradonna. Hidden Markov models as a support for diagnosis: Formalization of the problem and synthesis of the solution. In *25th IEEE Symp. on Reliable Distributed Systems (SRDS 2006)*, pages 245–256, Leeds, UK, October 2006.
- [26] S.W. Dan. A survey of peer-to-peer security issues. In *Proceedings of the International Symposium on Software Security, Theories and Systems (ISSS'02)*, pages 42–47, 2002.
- [27] D. Denning. An intrusion-detection model. *IEEE TSE*, 13(2):222–232, 1987.
- [28] F. Depaoli and L. mariani. Dependability in peer-to-peer systems. *IEEE Transaction on Internet Computing*, 8(4):54–61, July and August 2004.

- [29] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures (extended abstract). In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, pages 457–469, August 1992.
- [30] D. Dittrich. *The DoS project's trino distributed denial of service attack toll*. University of Washington, 1999.
- [31] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash to byzantine failures. In *da-Europe '02: Proc. of the 7th Ada-Europe International Conference on Reliable Software Technologies*, pages 24–50, London, UK, 2002. Springer-Verlag.
- [32] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proc. of the 3rd European Dependable Computing Conference*, pages 71–87, September 1999.
- [33] D. Dzung, M. Naedele, T. Von Hoff, and M. Crevatin. Security for industrial communication systems. *Proceedings of the IEEE*, 93(6):1152–1177, 2005.
- [34] F. Garrone (editor). Analysis of new control applications. Deliverable D2, EC project IST-2004-27513 (CRUTIAL), January 2007.
- [35] S. Donatelli (editor). Model-based evaluation of the middleware services and protocols & architectural patterns. Deliverable D25, EC project IST-2004-27513 (CRUTIAL), January 2008.
- [36] P. Ferguson and D. Senie. Network ingress filtering: defeating denial of service attacks which employ ip source address spoofing. In *Proceedings of the IETF, RFC2267*, January 1998.
- [37] L. Fink and K. Carlsen. Operating under stress and strain. *IEEE Spectrum*, March 1978.
- [38] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
- [39] J. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218, 1985.
- [40] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 protocol. Netscape Communications Corp., November 1996.
- [41] A. Garg and A.L. Narasimha Reddy. Mitigation of dos attacks through qos regulation. In *Proceedings of the 10th IEEE International Workshop on Quality of Service*, 2002.
- [42] L. Gordon and et al. *CSI/FBI Computer Crime and Security Survey*. Computer Security Inst., http://i.cmpnet.com/gocsi/db_area/pdfs/fbi/FBI2004.pdf, 2004.
- [43] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. 2006 CSI/FBI computer crime and security survey. Computer Security Institute, 2006.

- [44] V. Hadzilacos and S. Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR 94-1425, Dep. of Computer Science, Cornell Univ., New York - USA, May 1994.
- [45] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proc. of the 2nd Workshop on Hot Topics in System Dependability*, 2006.
- [46] K. Hickman. The SSL protocol. Netscape Communications Corp., February 1995.
- [47] International Electrotechnical Commission. Inter control center protocol (ICCP). IEC 60870-6 TASE.2, 1997.
- [48] International Electrotechnical Commission. Communication networks and systems in substations. IEC 61850, 2003.
- [49] A. Abou El Kalam, S. Benferhat, A. Miège, R. El Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin. Organization based access contro. In *POLICY*, 2003.
- [50] A. Abou El Kalam, Y. Deswarte, A. Baïna, and M. Kaâniche. Access control for collaborative systems: A web services based approach. In *ICWS*, 2007.
- [51] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen. Analysis of vulnerabilities in Internet firewalls. *Computers and Security*, 22(3):214–232, April 2003.
- [52] S. Kent and R. Atkinson. Security architecture for the internet protocol. IETF Request for Comments: RFC 2093, November 1998.
- [53] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.
- [54] A.D. Keromytis, V. Misra, and D. Rubenstein. Sos: Secure overlay services. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'02)*, August 2002.
- [55] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. IETF Request for Comments: RFC 2104, February 1997.
- [56] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication authors. RFC 2104, February 1997.
- [57] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang. Save: Source address validity enforcement protocol. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'02)*, 2002.
- [58] B. Littlewood and L. Strigini. Redundancy and diversity in security. In *Proceedings of the 9th European Symposium on Research Computer Security - ESORICS 2004*, pages 423–438. September 2004.
- [59] N. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.

- [60] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.
- [61] M. Marsh and F. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January 2004.
- [62] D. Moore, G.M. Voelker, and S. Savage. Inferring internet denial-of-service activity. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 2001.
- [63] B. Mukherjee, L. Heberlein, and K. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, 1994.
- [64] P. S. Munindar. Peering at peer-to-peer computing. *IEEE Transaction on Internet Computing*, 5(1):4–5, 2001.
- [65] S.G. Nathaniel and A. Krekelberg. Usability and privacy: a study of kazaa p2p file-sharing. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'03)*, pages 137–144, 2003.
- [66] N. Neves and P. Verissimo (editors). Preliminary architecture specification. Deliverable D4, EC project IST-2004-27513 (CRUTIAL), January 2007.
- [67] OASIS. Universal description, discovery and integration, v3.0.2. In *UDDI Specifications TC*, 2005.
- [68] R. Obelheiro, A. Bessani, L. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, Univ. of Lisbon, 2006.
- [69] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, pages 51–59, 1991.
- [70] K. Park and H. Lee. on the effectiveness of route-based packet filtering for distributed dos attack prevention in power-law internet. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 15–26, 2001.
- [71] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
- [72] S. Ratnasamy, P. Francis, M. Haudley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 27–31, 2001.
- [73] M. Reiter, M. Franklin, J. Lacy, and R. Wright. The omega key management service. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 38–47, 1996.
- [74] RFC. *RCF 3164*. <http://www.ietf.org/rfc/rfc3164.txt>.

- [75] L. Romano, A. Bondavalli, S. Chiaradonna, and D. Cotroneo. Implementation of threshold-based diagnostic mechanisms for COTS-based applications. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 296–303, Osaka University, Suita, Japan, October 13–16 2002.
- [76] A. Rowstron and P. Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceeding of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, November 2001.
- [77] S. Savage, D. Wetherall, A. karlin, and T. Anderson. Network support for ip traceback. *ACM/IEEE Transactions on Networking*, 9(3):226–237, June 2001.
- [78] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [79] A.C. Snoeren. Hash-based ip traceback. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, pages 3–14, 2001.
- [80] P. Sousa, N. Neves, A. Lopes, and P. Verissimo. On the resilience of intrusion-tolerant distributed systems. DI/FCUL TR 06–14, Dep. of Informatics, Univ. of Lisbon, Sept 2006.
- [81] P. Sousa, N. Neves, and P. Verissimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proc. of Int. Conf. on Dependable Systems and Networks*, pages 98–107, June 2005.
- [82] P. Sousa, N. F. Neves, and P. Verissimo. Hidden problems of asynchronous proactive recovery. In *Proc. of the Workshop on Hot Topics in System Dependability (HotDep'07)*, June 2007.
- [83] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1), 1989.
- [84] A. Stavrou and et al. Websos: An overlay-based system for protecting web servers from denial of service attacks. *the International Journal of Computer and Telecommunications Networking*, 48(5):781–807, August 2005.
- [85] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'02)*, 2002.
- [86] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer communications (SIGCOMM'01)*, pages 149–160, 2001.
- [87] K. Stouffer, J. Falco, and K. Kent. Guide to Supervisory Control And Data Acquisition (SCADA) and industrial control systems security. Recommendations of the National Institute of Standards and Technology (NIST). Special Publication 800-82 (INITIAL PUBLIC DRAFT), September 2006.

- [88] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1984.
- [89] P. Verissimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1), 2006.
- [90] P. Verissimo, N. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 2003.
- [91] P. Verissimo, N. Neves, and M. Correia. CRUTIAL: The blueprint of a reference critical information infrastructure architecture. In *Proceedings of CRITIS'06 1st International Workshop on Critical Information Infrastructures Security*, August 2006.
- [92] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.
- [93] W3C. Soap, version 1.2. In *W3C Recommendation*, 2003.
- [94] W3C. Extensible markup language (xml). In *W3C Recommendation*, 2004.
- [95] W3C. Web services description language (wsdl), version 2.0. In *W3C Recommendation*, 2006.
- [96] C. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *Software Engineering*, 23(11):684–721, 1997.
- [97] J. Wang, L. Lu, and A. Chien. Tolerating denial-of-service attacks using overlay networks-impact of topology. In *Proceedings of the ACM Workshop on Survivable and Self-Regenerative Systems*, 2003.
- [98] Y. Wang, I. Avramopoulos, and J. Rexford. Morpheus: Making routing programmable. In *Proceedings of the ACM Workshop on Internet Network Management (INM'07)*, pages 27–31, 2007.
- [99] K.Y. Yau, C.S. Lui, and F. Liang. Defending against distributed denial of service attacks with max-min fair server-centric router throttles. In *Proceedings of the IEEE International Workshop on Quality of Service (IWQoS'02)*, 2002.
- [100] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, 2003.
- [101] L. Zhou, F. Schneider, and R. Van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.