Architecture, Services and Protocols for CRUTIAL

Anas Abou El Kalan, Amine Baina, Hakem Beitollahi, Alysson Bessani, Andrea Bondavalli, Miguel Correia, Alessandro Daidone, Wagner Dantas, Geert Deconinck, Yves Deswarte, Henrique Moniz, Nuno Neves, Paulo Sousa, Paulo Verissimo

DI-FCUL TR–09–5

Março 2009

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

Technical reports are available athttp://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

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.:	**D18**
Title of the deliverable:	**Architecture, Services and Protocols for CRUTIAL**

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

Contractual Date of Delivery to the CEC:	13/03/2009
Actual Date of Delivery to the CEC:	13/03/2009
Organisation name of lead contractor for this deliverable	FCUL
Editor(s): Nuno Neves[2], Paulo Verissimo[2]	
Author(s): Anas Abou El Kalam[4]; Amine Baina[4]; Hakem Beitollahi[5]; Alysson Bessani[2]; Andrea Bondavalli[3]: Miguel Correia[2]; Alessandro Daidone[3]; Wagner Dantas[2]; Geert Deconinck[5]; Yves Deswarte[4]; Henrique Moniz[2]; Nuno Neves[2]; 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:	126

Abstract

This document describes the complete specification of the architecture, services and protocols of the project CRUTIAL. The CRUTIAL Architecture intends to reply to a grand challenge of computer science and control engineering: how to achieve resilience of critical information infrastructures (CII), in particular in the electrical sector.

In general lines, the document starts by presenting the main architectural options and components of the architecture, with a special emphasis on a protection device called the CRUTIAL Information Switch (CIS). Given the various criticality levels of the equipments that have to be protected, and the cost of using a replicated device, we define a hierarchy of CIS designs incrementally more resilient. The different CIS designs offer various trade offs in terms of capabilities to prevent and tolerate intrusions, both in the device itself and in the information infrastructure.

The Middleware Services, APIs and Protocols chapter describes our approach to intrusion-tolerant middleware. The CRUTIAL middleware comprises several building blocks that are organized on a set of layers. The Multipoint Network layer is the lowest layer of the 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 layer features three important building blocks: the Randomized Intrusion-Tolerant Services (RITAS), the CIS Communication service and the Fosel service for mitigating DoS attacks. The Activity Support layer comprises the CIS Protection service, and the Access Control and Authorization service. 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 definition of the services devoted to monitoring and failure detection activities.

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

Contents

List of Figures

1 Introduction

The largely computerized nature of critical infrastructures on the one hand, and the pervasive interconnection of systems all over the world, on the other hand, have generated one of the most fascinating current problems of computer science and control engineering: how to achieve resilience of critical information infrastructures. In project CRUTIAL, we are concerned with the susceptibility of the latter to computer-borne attacks and faults, i.e., with the protection of these infrastructures. We propose an architecture and a set of techniques and algorithms aiming at achieving resilience to faults and attacks in an automatic way. Although we focus on the computer systems behind electrical utility infrastructures as an example, the architecture we propose is generic and may come to be useful as a reference for modern critical information infrastructures.

It is worthwhile recapitulating some of the reasoning behind the blueprint of this architecture, recently published in [114, 115]. Although inspired by previous intrusion-tolerant system architectures, the CRUTIAL architecture was largely influenced by two facts. Firstly, the fact that Critical Information Infrastructures (CII) feature a lot of legacy subsystems (controllers, sensors, actuators, etc.). Secondly, the fact that conventional security and protection techniques can bring serious problems, when directly applied to CII controlling devices, by preventing their effective operation. Although they are very practical problems, we will show ahead that they yielded in fact very interesting research challenges.

Another relevant fact was that our belief that the crucial problems in critical information infrastructures lie with the forest, not the trees, has been confirmed everyday as new incidents have occurred. That is, the problem is mostly created by the generic and non-structured network interconnection of CIIs, which bring several facets of exposure impossible to address at individual level. Whilst it seems today non-controversial that such a status quo brings a considerable level of threat, to our knowledge there had been no previous attempt at addressing the problem through the definition of a reference model of a critical information infrastructure distributed systems architecture. One which, by construction, would lay the basic foundations for the necessary global resilience against abnormal situations. Our conjecture was that such a model would be highly constructive, for it would form a structured framework for (1) conceiving the right balance between prevention and removal of vulnerabilities and attacks; (2) achieving tolerance of remaining potential intrusions and designed-in faults; and (3) enabling adaptation and self-awareness mechanisms to overcome unforeseen situations.

Finally, and in a related manner, we conjectured that any solution, to be effective, has to involve automatic control of macroscopic command and information flows, occurring essentially between the several realms composing the critical information infrastructure architecture (both intra- and inter-organizations), with the purpose of securing appropriate system-level properties, at organizational level. This has to be addressed, in an automatic way, through innovative access control models that understand the organizational reality, and are thus capable of translating the related high-level security policies into the adequate technical mechanisms such as access control matrices and firewall filter rule-sets.

In this document, we present the specification of the architecture, services and protocols

of the project CRUTIAL. The definitions herein elaborate on the major architectural options and components established in the Preliminary Architecture Specification [83] and Preliminary Specification of Services and Protocols [84], with special relevance to the CRUTIAL middleware building blocks, and are based on the fault, synchrony and topological models defined in the previous documents. The document, in general lines, describes the generic aspects of the Architecture, the Middleware and the Runtime Support Services.

The Architecture chapter discusses the main options that were taken during the definition of the CRUTIAL Architecture. It also introduces several designs of a protection device called the *CRUTIAL Information Switch* (CIS). The CIS ensures that the incoming and outgoing traffic in/out of a protected LAN satisfies the security policy of the organization that manages the LAN. Given the various criticality levels of the CI equipment and the cost of using a replicated device, a hierarchy of CISs designs is proposed to offer incrementally more resilient solutions. The chapter ends with an overview of the CRUTIAL middleware, which is then next presented with more detail.

The Middleware chapter describes our approach to intrusion-tolerant middleware. The middleware organized in several layers, each one providing services to other layers or directly to the applications. 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 layer features three important building blocks: the Randomized Intrusion-Tolerant Services (RITAS), the CIS Communication service and the Fosel service for mitigating DoS attacks. The Randomized Intrusion-Tolerant Services (RITAS) are organized as a stack of randomized 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 [50] (also called the FLP result), but present a significant performance improvement over previous protocols of the same class. The CIS Communication service supports secure communication between CIS and, ultimately, between LANs. It provides secure channels, multicast primitives, and probabilistic gossip-based information diffusion between CIS. 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 the DoS problem with an overlay protection layer for DoS attacks on top of the normal infrastructure.

The Activity Support layer comprises the CIS Protection service and the Access Control and Authorization service. 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. 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. 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 layer is devoted to monitoring and failure detection activities. Diagnosis in Crutial 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 Runtime Support chapter features as a main component, the Proactive-Reactive Recovery Service, whose aim is to guarantee perpetual execution of any components it protects. 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.

2 Architecture

The CRUTIAL architecture encompasses four aspects:

- Architectural configurations featuring trusted components in key places, which a priori induce prevention of some faults, and of certain attack and vulnerability combinations.
- Middleware devices that achieve runtime automatic tolerance of remaining faults and intrusions, supplying trusted services out of non-trustworthy components.
- Trustworthiness monitoring mechanisms detecting situations not predicted and/or beyond assumptions made, and adaptation mechanisms to survive those situations.
- Organisation-level security policies and access control models capable of securing information flows with different criticality within/in/out of a CII.

Given the severity of threats expected, some key components are built using architectural hybridisation methods in order to achieve *trusted-trustworthy* operation [112]: an architectural paradigm whereby components prevent the occurrence of some failure modes *by construction*, so that their resistance to faults and hackers can justifiably be trusted. In other words, some special-purpose components are constructed in such a way that we can argue that they are always secure, so that they can provide a small set of services useful to support intrusion tolerance in the rest of the system.

Intrusion tolerance mechanisms are selectively used in the CRUTIAL architecture, to build layers of progressively more trusted components and middleware subsystems, from baseline untrusted components (nodes, networks) [112]. This leads to an automation of the process of building trust: for example, at lower layers, basic intrusion tolerance mechanisms are used to construct a trustworthy communication subsystem, which can then be trusted by upper layers to securely communicate amongst participants without bothering about network intrusion threats.

One of the innovative aspects of this work, further to intrusion tolerance, is the resilience aspect, approached through two paradigms: *proactive-resilience* to achieve exhaustion-safety [103], to ensure perpetual, non-stop operation despite the continuous production of faults and intrusions; and *trustworthiness monitoring* to perform surveillance of the coverage stability of the system, that is, of whether it is still performing inside the assumed fault envelope or beyond assumptions made [21]. In the latter case, dependable adaptation mechanisms are triggered.

Finally, the desired control of the information flows is partly performed through protection mechanisms using an adaptation of the *organisation-based access control model (OrBAC)* [1] for implementing global-level security policies. OrBAC allows the expression of security policy rules as high level abstractions, and the composition of the security policies of the organizations into one global policy.

The mechanisms and algorithms in place achieve system-level properties of the following classes: trustworthiness or resistance to faults and intrusions (i.e., security and dependability);

timeliness, in the sense of meeting timing constraints raised by real world control and supervision; coverage stability, to ensure that variation or degradation of assumptions remains within a bounded envelope; dependable adaptability, to achieve predictability in uncertain conditions; resilience, read as correctness and continuity of service even beyond assumptions made.

2.1 Fundamental Architectural Options

We view the system as a WAN-of-LANs, as introduced in [110]. There is a global interconnection network, the WAN, that switches packets through generic devices that we call *facility gateways*, which are the representative gateways of each LAN (the overall picture is shown in Figure 2.1). The WAN is a logical entity operated by the CII operator companies, which may or may not use parts of public network as physical support. A LAN is a logical unit that may or may not have physical reality (e.g., LAN segments vs. Virtual LANs (VLANs)). More than one LAN can be connected by the same facility gateway. All traffic originates from and goes to a LAN. As example LANs, the reader can envision: the administrative clients and the servers LANs; the operational (SCADA) clients and servers LANs; the engineering clients and servers LANs; the PSTN modem access LANs; the Internet and extranet access LANs, etc.

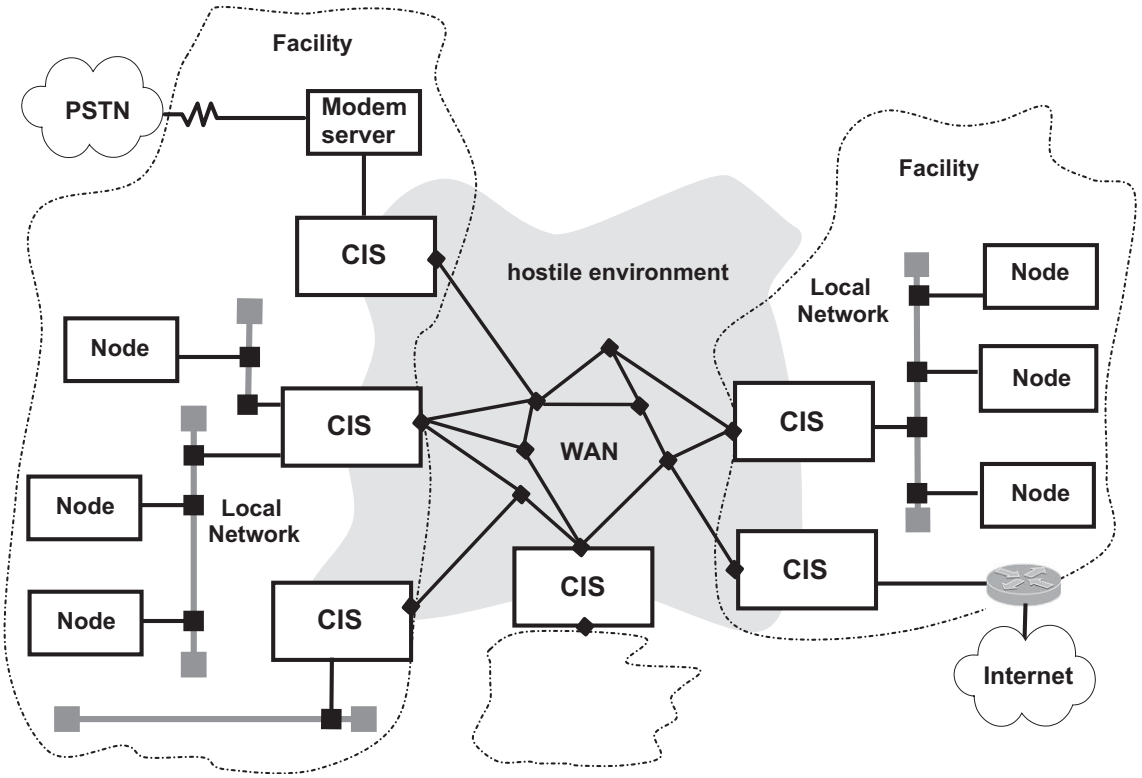


Figure 2.1: CRUTIAL overall architecture (WAN-of-LANs connected by CIS, *P* processes live in the several nodes)

The facility gateways of a CRUTIAL critical information infrastructure are more than mere TCP/IP routers. Collectively they act as a set of servers providing distributed services relevant to solving our problem: *achieving control of the command and information flow, and securing a set of*

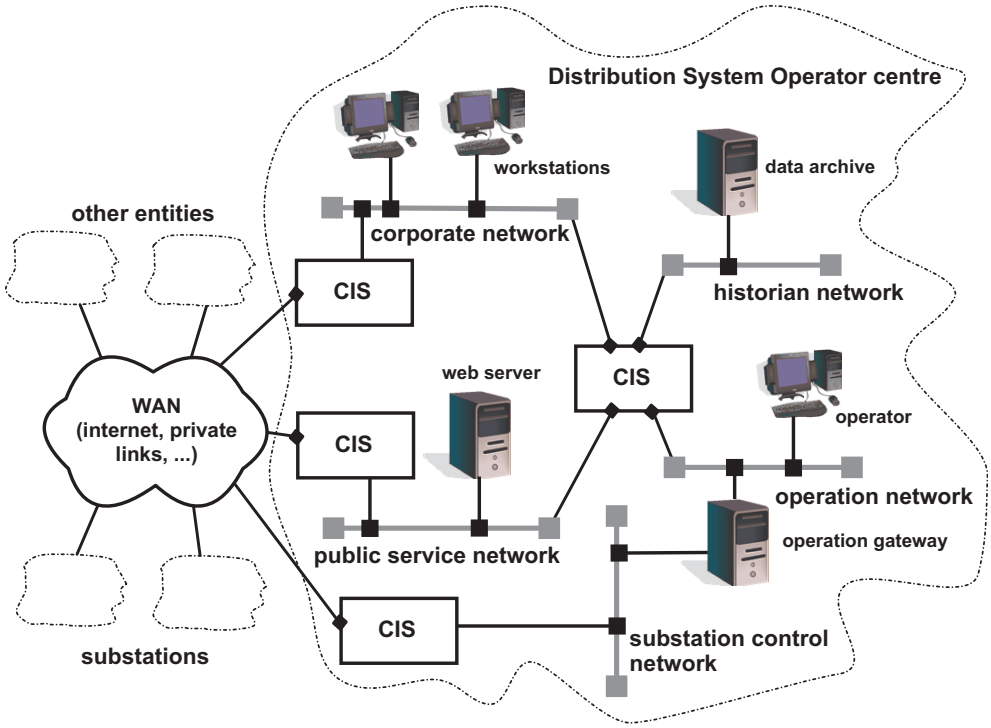


Figure 2.2: Example mapping of part of an infrastructure to the WAN-of-LANs architecture.

necessary system-level properties. CRUTIAL facility gateways are called *CRUTIAL Information Switches (CIS)*, which in a simplistic way could be seen as sophisticated circuit or application level firewalls combined with equally sophisticated intrusion detectors, connected by distributed protocols.

This set of servers must be intrusion-tolerant, prevent resource exhaustion providing perpetual operation (i.e., can not stop), and be resilient against assumption coverage uncertainty, providing survivability. The services implemented on the servers must also secure the desired properties of flow control, in the presence of malicious traffic and commands, and in consequence be themselves intrusion-tolerant.

An assumed number of components of a CIS can be corrupted. Therefore, a CIS is a logical entity that has to be implemented as a set of replicated physical units (CIS replicas) according to fault and intrusion tolerance needs. Likewise, CIS are interconnected with intrusion-tolerant protocols, in order to cooperate to implement the desired services. The CIS boxes in the figure represent these intrusion-tolerant, replicated, logical CIS.

An example WAN-of-LANs. The WAN-of-LANs model is very abstract so in this section we use it to represent a small part of a distribution power grid. This example is inspired in a testbed of the CRUTIAL project presented in [37][1], and the corresponding scenario in [45].

Figure 2.2 presents a Distribution System Operator (DSO) centre. This centre includes

[1] See Section 3.1.1 of that document.

several networks and is connected to the substations through the substation control network (bottom). This network is connected to the substations through the (logical) WAN, which can be the Internet, a set of private links, VLANs or other type of network. The DSO centre includes the corporate network (top), the public service network were services like web servers are placed (middle), the data historian network were historical information about the infrastructure is stored (top right), and the operation network were operators monitor and control the power generation infrastructure (right). All these networks are modeled as (logical) LANs and are connected by CIS, that protect them from one another and, especially, from the Internet/WAN.

2.2 Hierarchy of CRUTIAL Information Switches

The protection of a LAN is made by a CIS device that 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 organization that manages the LAN. CS supports secure communication between CIS and, ultimately, between LANs. The CS provides secure channels, reliable and atomic multicast primitives, and probabilistic gossip-based information diffusion between CIS.

Give the various criticality levels of the CI equipment and the cost of using a replicated device, it is worth defining a hierarchy of CISs designs incrementally more resilient. Next we present this hierarchy and, for each design, we point out what it offers, its cost and limitations.

Notice that more details about the CIS Protection and Communication Services are provided respectively in Section 3.3.1 and 3.2.2.

Non-Intrusion-Tolerant CIS Figure 2.3 depicts the design of a non-intrusion-tolerant CIS. The non-intrusion-tolerant CIS is the cheapest design because it only requires one machine just as any classical firewall. Nevertheless, it offers better protection than normal firewalls through the use of application-level policy enforcement and of a rich access control model. These characteristics reduce the probability of an attacker being able to construct a well-crafted message and deceive the CIS to let it go through. Although it is more difficult to mislead the CIS, an attacker may find a way of compromising the machine where the CIS is running (e.g., by exploiting a vulnerability of the operating system) and take control of CIS operation. Notice that the attacker may need days, weeks, or even months to find a vulnerability and deploy such an attack, but from what happened in the past we know that with a reasonable probability she will eventually have success in her quest.

Intrusion-Tolerant CIS 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 (or LAN)

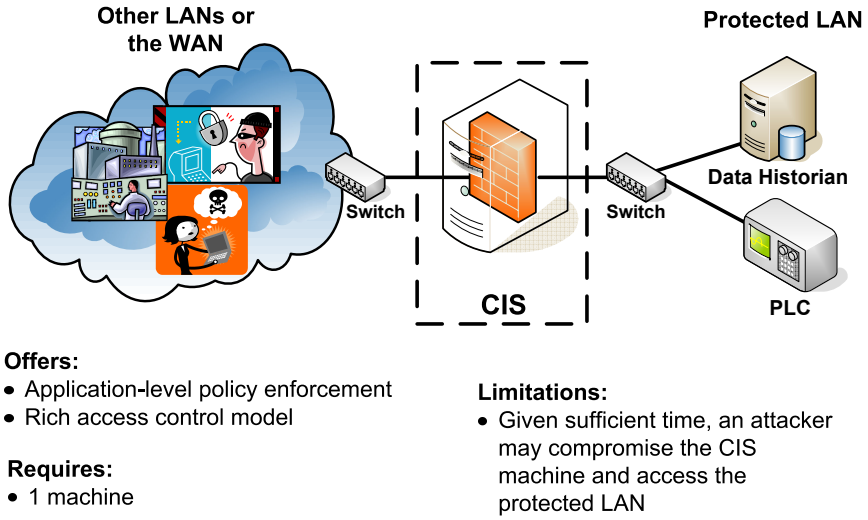


Figure 2.3: Non-intrusion-tolerant CIS design.

and the 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 non-trusted side, through the CIS, to the computers and/or devices in the protected LAN. A first difficulty to address is that traffic has to be received by all n replicas, instead of only 1 (as in a normal firewall), to allow every replica to participate in the decisions. A second problem is that up to f replicas can be faulty and behave malicious, both towards the other replicas as to the receiver computers (e.g., the SCADA controllers).

Our solution to the first problem is to use some device (e.g., an Ethernet hub) to broadcast the traffic to all replicas. These verify whether the messages comply with the security policy and do a vote, approving a message if and only if at least $f+1$ different replicas are in favor. This guarantees that at least one correct replica thinks that the message should go through. An approved message is then transmitted by the CIS to the destination by a distinguished replica, the *leader*, so there is no unnecessary traffic multiplication inside the LAN.

Traditionally, intrusion-tolerant mechanisms address the second problem with masking protocols of the Byzantine type, which extract the correct result from the n replicas, despite f maliciously: basically, only results (or messages) that are supported by $f+1$ replicas are accepted. Since a result must be sent to the computers in the protected LAN, this consolidation has to be performed either at the source or at the destination. The simplest and most usual approach implements a front-end at 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; or (3) the message has a signature generated by $f+1$ replicas using threshold cryptography[2]. This however would require changes to the end hosts and the traffic in the protected LAN would be multiplied by n in certain cases (every replica would send the message), which is undesirable.

[2]Threshold cryptography is a form of cryptographic that allows to split a key among a set of parties and to define the minimum number of parties that are required to make useful things with the key (see [99]). Unfortunately, it can only be used in combination with public key cryptography schemes where different keys are used to encrypt and to decrypt, whereas IPSEC/AH uses the same key to produce and verify MACs.

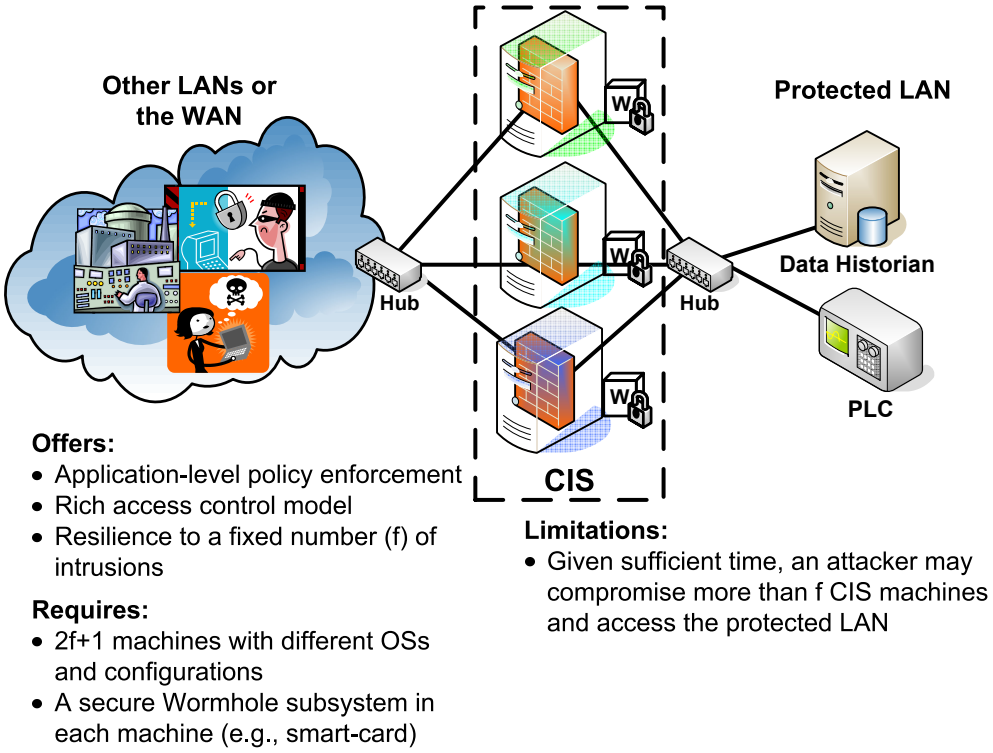


Figure 2.4: Intrusion-tolerant CIS design.

Therefore, we should turn ourselves to consolidating at the source and transmit *only one, but correct, message*. What is innovative here is that the source-consolidation mechanism should be transparent to the protected LAN. Moreover, this has to be done with the following difficulty in mind – since a faulty replica (leader or not) has direct access to the LAN, it can send incorrect traffic to the protected computers, which can not distinguish between good and bad messages. This makes consolidation at the source a hard problem.

According to best practice recommendations from expert organizations and governments, the standard protocols that are expected to be generalized in SCADA/PCS systems will utilize IPSEC to secure the communications. Consequently, we can assume that the IPSEC Authentication Header (AH) protocol [66] will be available, and take advantage of it to design our solutions. The basic idea is that the protected 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 generated using a shared key[3] K 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 highly resilient design – there must be some secure component that stores the shared key and produces MACs. This requirement calls for a secure component in an otherwise Byzantine-on-failure environment, which we will name as a *secure wormhole* [111]. The wormhole can be deployed as a set of local trustworthy components, one per replica, using technologies such as smart-cards or cryptographic boards. Figure 2.4 depicts the design of an intrusion-tolerant CIS.

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

This CIS design requires $2f + 1$ machines in order to tolerate f intrusions. Thus, the configuration presented in Figure 2.4 is able to tolerate one intrusion. Notice that such a design only makes sense if the different machines can not be attacked in the same way, i.e., they must not share the same set of vulnerabilities. In order to achieve this goal, each CIS replica is deployed in a different operating system (e.g., Linux, FreeBSD, Solaris), and the operating systems are configured to use different passwords and different services. In addition, each CIS replica is enhanced with a secure wormhole subsystem that stores a secret key and this key is used to produce a MAC for messages that are approved by at least $f + 1$ replicas. Given that the wormhole is secure, no malicious replica can force it to sign an unapproved message.

In practical terms, the intrusion-tolerant CIS is more difficult to compromise than the non-intrusion-tolerant version because an attacker will need to find vulnerabilities and deploy attacks against $f + 1$ diverse replicas and not just one. The task of compromising each replica may take days, weeks, or months, but attackers tend to be patient persons.

Intrusion-Tolerant and Self-Healing CIS The most resilient CIS design combines intrusion tolerance with self-healing mechanisms in order to address the limitations explained in the previous section. Self-healing is provided by a proactive-reactive recovery service that combines time-triggered periodic rejuvenations with event-triggered rejuvenations when something bad is detected or suspected [101].

Proactive recoveries are triggered periodically in every replica even if it is not compromised. The goal is to remove the effects of malicious attacks/faults even if the attacker remains dormant. Otherwise, if an attacker compromises a replica and makes an action that can be detected (e.g., sending a message not signed with the shared key K), the compromised replica is rejuvenated through a reactive recovery. Moreover, the rejuvenation process of a (proactive or reactive) recovery does not simply restore replicas to known states, since this would allow an attacker to exploit the same vulnerabilities as before. The rejuvenation process itself introduces some degree of diversity to restored replicas (change operating system, use memory obfuscation techniques, change passwords, etc.), so that attackers will have to find other vulnerabilities in order to compromise a replica. Figure 2.5 depicts the design of an intrusion-tolerant CIS with self-healing mechanisms.

This CIS design requires $2f + k + 1$ machines in order to tolerate f intrusions per recovering period (dictated by the periodicity of the proactive recoveries). The new parameter k represents the number of replicas that recover at the same time and its value is typically one. If this parameter was not included in the calculation of the total number of required machines, the CIS could become unavailable during recoveries. Thus, the configuration presented in Figure 2.5 composed of 4 replicas is able to tolerate one intrusion while another is being rejuvenated. The length of the recovering period corresponds to the sum of the recovery time of each replica. In the experiments performed in our laboratory we were able to recovery each replica in less than 2.5 minutes, which means that the recovering period was less than 10 minutes when using 4 replicas [101]. In this scenario, an attacker would need to find vulnerabilities and deploy attacks against $f + 1 = 2$ replicas within 10 minutes, which seems to be difficult given that each recovery changes the set of vulnerabilities and the attacker has to restart her work. Moreover, we believe that the recovery

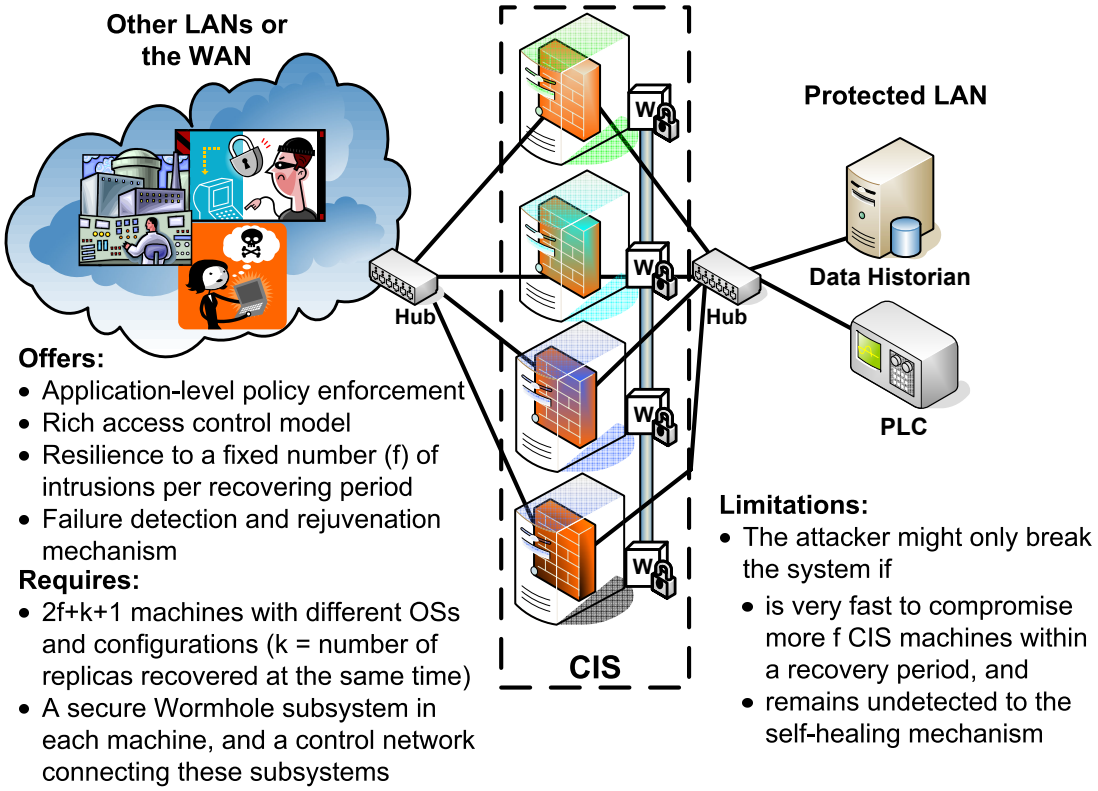


Figure 2.5: Intrusion-tolerant & self-healing CIS design.

time of a replica can be reduced to less than one minute using readily available technologies such as solid state disks.

2.3 Middleware

We now observe the part of the system made of the WAN and all the CIS that interconnect all the internal LANs of the critical information infrastructure to the WAN (recall Figure 2.1).

We model this setting as a distributed system with N nodes (CIS). We use the weakest fault and synchrony models that allow to carry out the application tasks. So, we use the asynchronous/arbitrary model, which does not make any assumptions about either time needed to make operations and faults/intrusions that can occur, as a starting point, and strengthen it as needed. For example, by resorting to hybrid models using wormholes [111], and assuming some form of partial synchrony.

We assume that the environment formed by the WAN and all the CIS is hostile (not trusted), and can thus be subjected to malicious (or arbitrary or Byzantine) faults. On the other hand, LANs trust the services provided by the CIS, but are not necessarily trusted by the latter. That is, as we will see below, LANs have different degrees of trustworthiness, which the CIS distributed protocols have to take into account. CIS securely switch information flows as a service to edge LANs as clients.

We assume that faults (accidental, attacks, intrusions) continuously occur during the life-time of the system, and that a maximum number of f malicious (or arbitrary) faults can occur within a given interval. We assume that services running in the nodes (CIS) cooperate through distributed protocols in such an environment. In consequence, these nodes have to be capable of tolerating fault/intrusions by employing replication techniques.

Some of the services running in CIS may require some degree of timeliness, given that SCADA implies synchrony, and this is a hard problem with malicious faults. We also take into account that these systems should operate non-stop, a hard problem with resource exhaustion (the continued production of faults during the life-time of a perpetual execution system leads to the inevitable exhaustion of the quorum of nodes needed for correct operation [103]).

LAN-level services A LAN is the top-level unit of the granularity of access control, regardless of possible finer controls. It is also and correspondingly, a unit of trust or mistrust thereof. In fact, we are not concerned with what happens inside a LAN, except that we may attribute it a different levels of trust. For instance, if the LAN is a SCADA network, the level of trust is high, but if it is the access to the Internet then the level of trust is low.

Traffic (packets) originating from a LAN receive a label that reflects this level of trust, and contains access control information, amongst other useful things.

The trustworthiness of a label (that is, the degree in which it can or not be tampered with) can vary, depending on the criticality of the service. In the context of this document, and without loss of generality, we assume it is an authenticated proof of a capacity.

WAN-level services The collection of CIS implements a set of core services, aiming at achieving the objectives we placed as desirable for a reference model of *critical information infrastructure distributed systems architecture*:

- Intrusion-tolerant information and command dissemination between CIS units, with authentication and cryptographic protection (broadcast, multicast, unicast).
- Pattern-sensitive information and command traffic analysis (behaviour and/ or knowledge-based intrusion detection) with intrusion-tolerant synchronisation and coordination between local Intrusion Detection Systems (IDSs).
- CIS egress/ingress access control based on LAN packet labels and/or additional mechanisms, implementing an instance of the global security policy.

The CIS middleware layers implement functionality at different levels of abstraction, as represented in Figure 2.6. As mentioned earlier, a middleware layer may overcome (through intrusion tolerance) the fault severity of lower layers and provide certain functions in a trustworthy way.

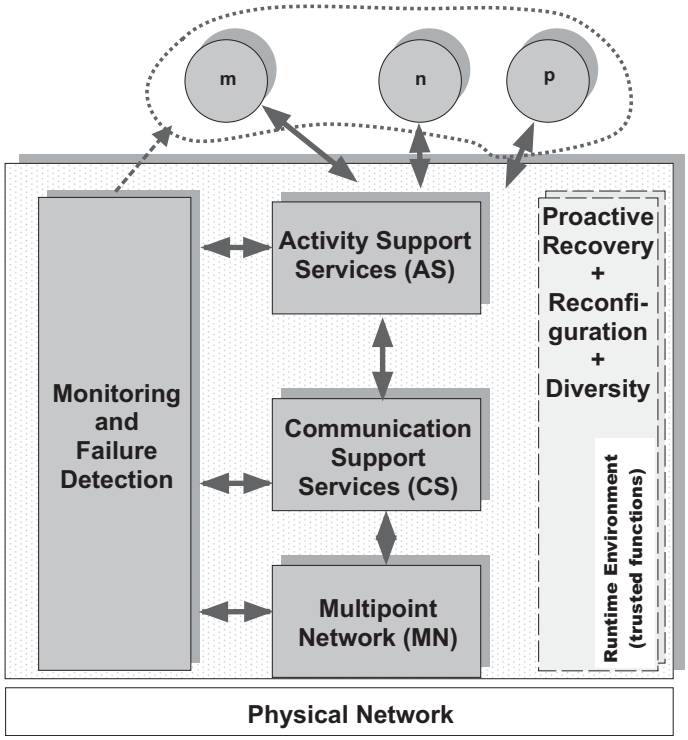


Figure 2.6: CRUTIAL middleware.

The lowest layer is the *Multipoint Network* module (MN), created over the physical infrastructure. Its main properties are the provision of multipoint addressing, secure channels (IPsec, SSL, etc.), and management communications, hiding the specificities of the underlying network. The *Communication Support Services* module (CS) implements basic cryptographic primitives, Byzantine agreement, consensus, group communication and other core services. The CS module depends on the MN module to access the network. The *Activity Support Services* module (AS) implements building blocks that assist participant activity, such as replication management (e.g., state machine, voting), IDS and firewall support functions, access control. It depends on the services provided by the CS module.

The block on the left of the figure generically implements Monitoring and Failure Detection. Failure detection assesses the connectivity and correctness of remote nodes, and the liveness of local processes. Trustworthiness monitoring and dependable adaptation mechanisms also reside in this module, and have interactions with all the modules on the right. Both the AS and CS modules depend on this information.

The block to the right represents the support services. These include the usual operating system's services, but also the trusted services supplied in support to the algorithms in the various modules: proactive recovery, reconfiguration, and diversity management.

In the following chapters we provide more details about all these components.

3 Middleware Services, APIs and Protocols

The Middleware Services, APIs and Protocols chapter describes the intrusion-tolerant middleware developed within the context of the project. The middleware comprises several layers, including the Multipoint Network, Communication and Activity Support, and Monitoring and Failure Detection.

3.1 Multipoint Network

The Multipoint Network (MN) is the bottom layer of CRUTIAL's middleware (see Figure 2.6). 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 IP – 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 IPSec – Internet Protocol Security

Internet Protocol Security (IPSec) is an extension of IP that provides some level of security [67]. 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. The Authentication Header (AH) protocol provides connectionless integrity, data origin authentication, and an optional anti-replay service. The Encapsulation Security Payload (ESP) protocol 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 UDP and TCP

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 SSL – Secure Socket Layer

The Secure Socket Layer (SSL) [59, 52], 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

This section presents the Communication Support layer, which features three main building blocks: the Randomized Intrusion-Tolerant Services, the CIS Communication service and the Fosel service for mitigating 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

RITAS, which stands for *Randomized Intrusion-Tolerant Asynchronous Services*, is stack of intrusion-tolerant protocols for distributed systems. At the heart of distributed applications there is usually a need to conduct some sort of coordinated activity. The protocols provided in RITAS can be used as primitives to carry out fundamental coordination activities in a dependable way. The stack provides several forms of broadcast and consensus protocols that operate correctly even if some of the processes that compose the system are attacked and compromised by a malicious adversary. In other words, the protocols are intrusion-tolerant.

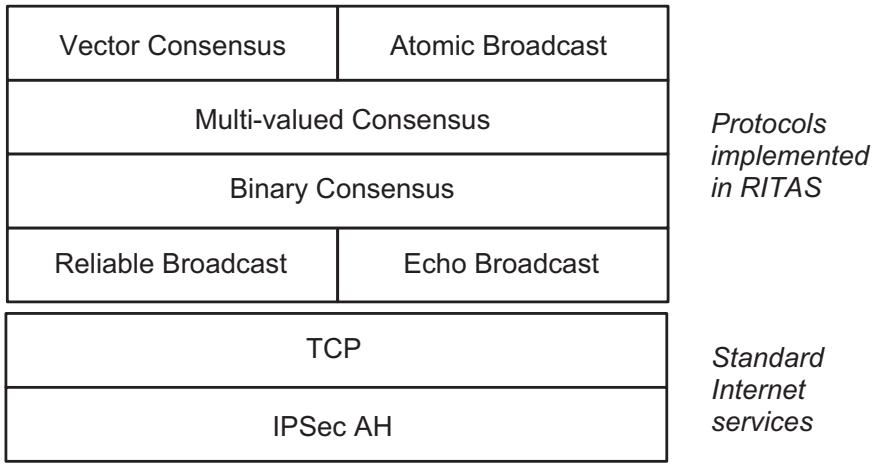


Figure 3.1: The RITAS protocol stack.

The RITAS protocols stack is depicted in Figure 3.1. All protocols in the stack rely on two standard Internet services that are abstracted in 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. At the bottom there are the broadcast primitives: *echo broadcast* and *reliable broadcast*. On top of the broadcast primitives is the most basic flavor of consensus: *binary consensus*. This is the only randomized protocol in the stack. On top of binary consensus there is the *multi-valued consensus* protocol, which allows the proposal of arbitrary values. Finally, at the top of the stack there are two protocols: *vector consensus*, and *atomic broadcast*.

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

3.2.1.1 System Model

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

There are no constrains on the kind of faults that can occur in the system. This class of unconstrained faults is usually called *arbitrary* or *Byzantine*. Processes are said to be *correct* if they do not *fail*, i.e., if they follow their protocol until termination. Processes that fail are said to be *corrupt*. No assumptions are made about the behavior of corrupt processes – they can, for instance, stop executing, omit messages, send invalid messages 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 also access to a random bit generator that returns unbiased bits observable only by the process (if the process is correct).

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

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

as halting the communication between the processes altogether.

3.2.1.2 Protocols of RITAS

Reliable Channels. The two layers at the bottom of the stack implement a reliable channel (see Figure 3.1). This abstraction provides a point-to-point communication channel between a pair of correct processes with two properties: reliability and integrity. Reliability means that messages are eventually received, and integrity says that messages are not modified in the channel. More formally, these properties are defined as follows:

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

In practical terms, the properties can be enforced using standard Internet protocols: reliability is provided by TCP, and integrity by the IPSec Authentication Header (AH) protocol. TCP establishes a point-to-point two-way communication channel between a pair of processes and guarantees reliable and FIFO delivery of sender to receiver data. The IPSec AH protocol guarantees connectionless integrity and data origin authentication of IP datagrams. It protects all fields of an IP datagram except those that are mutable during the transmission of an IP packet on the network (e.g., the TTL field). The IPSec AH protocol requires that every pair of processes p_i and p_j share a secret symmetric key k_{ij}.

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.

Reliable Broadcast. The *reliable broadcast* protocol ensures that all correct processes eventually receive the same set of messages. No constrains are placed on the relative delivery order of messages. It is defined formally as follows:

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

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

Algorithm 1 Reliable Broadcast protocol (for process p_i).

Function *R_Broadcast* (v_i, rbid)

INITIALIZATION:

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

TASK T0 (SENDER ONLY):

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

TASK T1:

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

The implemented reliable broadcast protocol was originally proposed in [23], and it is presented in Algorithm 1. An instance of the protocol, identified by *rbid*, starts with the sender broadcasting a message (INITIAL, *m*, rbid) to all processes. Upon receiving this message a process sends a (ECHO, *m*, rbid) message to all processes. It then waits for at least $\lfloor \frac{n+f}{2} \rfloor + 1$ (ECHO, *m*, rbid) messages or $f+1$ (READY, *m*, rbid) messages, and then it transmits a (READY, *m*, rbid) message to all processes. Finally, a process waits for $2f+1$ (READY, *m*, rbid) messages to deliver *m*. The broadcasts inside the protocol are made via the reliable channels.

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

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

The implemented *echo broadcast* primitive was originally proposed in [108], and is a variant of the previously described *reliable broadcast* protocol. It is presented in Algorithm 2.

Algorithm 2 Echo Broadcast protocol (for process p_i).

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

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

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

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.

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

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

Algorithm 3 Binary Consensus protocol (for process p_i).

Function *B_Consensus* (v_i, bcid)

1: **repeat**
2: *R_Broadcast* ($\langle$S1, v_i, bcid, i$\rangle$);
3: **wait until** ($(n-f)$ *valid* S1 messages have been delivered);
4: $\forall_j$: **if** ($\langle$S1, v_j, 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, bcid, i$\rangle$);
11: **wait until** ($(n-f)$ *valid* S2 messages have been delivered);
12: $\forall_j$: **if** ($\langle$S2, v_j, bcid, j$\rangle$ has been delivered) **then** V_i[j] $\leftarrow v_j$; **else** V_i[j] $\leftarrow \perp$;
13: **if** ($\exists_v : v \neq \perp$ 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, bcid, i$\rangle$);
19: **wait until** ($(n-f)$ *valid* S3 messages have been delivered);
20: $\forall_j$: **if** ($\langle$S3, v_j, 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**

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 [23]. The protocol has an expected number of communication steps for a decision of 2^{n-f}, and uses the underlying *reliable broadcast* as the basic communication primitive. The main advantage of this algorithm is that is does not use any cryptography whatsoever (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 proceeds in 3-step rounds, running as many rounds as necessary for a decision to be reached (see Algorithm 3). The first step (lines 2-9) of an execution of the protocol identified by *bcid* starts when each process p_i (reliably) broadcasts its proposal v_i. Then waits for $n-f$ *valid* messages and changes v_i to reflect the majority of the received values. In the second step (lines 10-17), p_i broadcasts v_i, waits for the arrival of $n-f$ *valid* messages, and if more than half of the received values are equal, v_i is set to that value; otherwise v_i is set to the undefined value $\perp$. Finally, in the third step (lines 18-27), p_i broadcasts v_i, waits for $n-f$ *valid* messages, and decides if at least $2f+1$ messages have the same value $v \neq \perp$. Otherwise, if at least $f+1$ messages have the same value $v \neq \perp$, then v_i is set to v and a new round is initiated. If none of the above conditions apply, then v_i is set to a random bit with value 1 or 0, with probability $\frac{1}{2}$, and a new round is initiated.

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

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 $\mathscr{V}$. Depending on the proposals, the decision is either one of the proposed values or a default value $\perp \notin \mathscr{V}$. 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 [32]. It uses the services of the underlying *reliable broadcast*, *echo broadcast*, and *binary consensus* layers. The main difference from the original protocol is the use of echo broadcast instead of reliable broadcast at a specific point, and a simplification of the validation of the vectors used to justify the proposed values. These changes grant greater efficiency to the protocol without compromising its correctness. The protocol is presented in Algorithm 4.

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

Function *M_V_Consensus* (v_i, cid)

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

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 $V_i[k] \in \mathscr{V}$ 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

Algorithm 5 Vector Consensus protocol (for process p_i).

Function *Vector_Consensus* (v_i, vcid)

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

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).

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 [91]. 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 [32], which uses *reliable broadcast* and *multi-valued consensus* as underlying primitives. The protocol starts by reliably broadcasting a message containing the proposed value by the process and setting the round number r_i to 0 (see Algorithm 5). 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.

Atomic Consensus. 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 [32]. The main difference from the original protocol is that it has been adapted to use multi-valued consensus instead of vector consensus and to utilize message identifiers for the agreement task instead of cryptographic hashes. These changes were made for efficiency and have been proved not to compromise the correctness of the protocol. The protocol uses *reliable broadcast* and *multi-valued consensus* as primitives.

The atomic broadcast protocol, presented in Algorithm 6, is conceptually divided in two tasks: (1) the broadcasting of messages, and (2) the agreement over which messages should be delivered.

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

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

Algorithm 6 Atomic Broadcast protocol (for process p_i).

INITIALIZATION:

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

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

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

TASK T1:

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

TASK T2:

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

$\perp$, 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 $10 \leq num < 50$.

3.2.1.3 API exported by RITAS

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

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

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

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

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

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

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

int ritas_proc_add_ipv4(*ritas_t* $*$*ctx*, *u_short pid*, *u_char* $*$ *ip*, *u_short port*, *u_char* $*$*key*);

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

Service Request Functions. The service request functions give the application programmer access to the actual protocols provided by the stack. These functions can be divided in two groups,

one for the broadcast primitives and another for the various consensus protocols. The service request functions can only be called after the relevant session context has been properly initialized and the individual processes added to the group. When a session context is destroyed, no service requests functions for that particular session can be called afterwards.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The communication between SCADA/PCS system and the remote control and monitoring systems often has to occur in a timely manner, i.e., with real-time requirements. That is, the messages transmitted have to be delivered within specific application-defined deadlines which reflect known standard routines in the CII operation network (e.g., for power grid operation see [36]). Connection to the large-scale network often relies on *multihoming*, i.e., each local network is connected to two or more service providers (or ISP, from Internet Service Provider) [6]. End-to-end control application communication happens over well-provisioned IP channels that provide high-bandwidth and a good level of redundancy. Nevertheless, some of these channels share ISP network resources with other kinds of traffic, including external traffic from the Internet.

Just like in conventional applications running over IP, communication between SCADA systems and remote control entities may experience network failures which can result in communication problems. Firstly, failures may affect network components (e.g., routers and links between them) and disrupt the IP-level routing service. These failures can cause from network service performance degradation to connectivity outages, hindering endpoint communication. In fact, network failures are not uncommon, even in IP backbones [60, 76]. Secondly, failures may be caused by *denial-of-service (DoS) attacks* [92], which represent a more challenging threat. CII are attractive targets to cyber-terrorism acts [125], therefore, depending on how powerful the attack is, the severity of failures can be alarmingly high. In this case, failures affect a broader amount of network equipment in more locations than accidental failures, being more frequent and/or longer in time. Consequently, these failures pose an increased security risk to the CII communication, impacting even more its timeliness.

Overall, from the standpoint of control applications, network failures, either accidental or intentional, negatively affect the network quality of service, which otherwise would allow deadline-compliant exchanging of data between communicating endpoints. Nevertheless, there is a substantial difference to traditional systems: communication failures in these control applications are more critical since they may result in severe social and economic impacts.

Overlay networks (or *overlay routing*) is a paradigm that can be used by distributed applications to overcome communication problems encountered at network level. The idea is that a given set of network-connected endpoints (overlay nodes) is engaged in collaboratively relaying application data through virtual paths between them. Each virtual path connecting two overlay nodes is composed by one or more of the underlying network-level paths and is selected according to flexible application criteria. Therefore, overlay networks provide the overall system improved end-to-end performance and redundancy to be more resilient to network failures. This fits quite well with applications with specific requirements that in another way would be at least difficult to be satisfied by a legacy network.

There is much literature about building application semantic-aware overlay routing, with different virtual path selection strategies. However, most solutions do not have timeliness as objective (e.g., [8, 55]), while others do stem from a context where latency is a concern but not achieving pre-defined maximum delays (e.g., [100]). In this latter case, timeliness is a design

requirement, but it is not clear whether it would be achieved in very stringent scenarios characterised, for example, by persistent packet losses caused by long-term congestions [10].

Here we intend to provide *a communication service with timeliness constraints* over a large-scale network as needed nowadays by control applications in CIIs. It refers to a network environment where connectivity to the large-scale network portion is provided by *multihoming* and end-to-end communication is managed via *well-provisioned channels*.

This work is based on the assumption that the network does not provide latency and bandwidth guarantees. Theoretically it is possible to have these guarantees, e.g., by using technologies like ATM and DiffServ, but network service providers typically do not provide them, especially in large-scale networks. Therefore, we propose a solution that does not require such guarantees from the network, only plain Internet-like IP communication service. We propose a new solution based on multihoming and overlay routing to provide timely reliable communication. The solution unifies the best of existing ones but with a novel path selection scheme that results in more guarantees than other strategies to communicate control data in a timely way. The design of such novel overlay-based solution is described throughout the rest of this section. In particular, we explain the reasoning of the solution, which is specially designed taking into account the specificities of the CII scenario, and we present how such solution is currently implemented, describing its main functional parts.

3.2.2.1 CII Networks and their Properties

The CII process control system consists on a set of distributed applications spread over different locations within some territory (like a country for example) that are interconnected by a large-scale data network. Such applications control and monitor in a cooperative and electronic way an associated set of one or more physical processes running in the CII. The applications are SCADA/PCS systems and remote controller computers, located in different power grid utility local networks that together automate the necessary control and monitoring activities performed over an electricity supply operation chain (e.g., generation and transmission processes).

The CII employed in such context has a collection of specific properties that are described in more detail next.

- *Static territory-limited large-scale network*: The substrate large-scale network covers a geographically delimited zone and its organisation is tendentiously static, that is, the arrangement of local networks and the interconnecting links between pairs of these local networks are not changed often.
- *Connectivity to large-scale network via ISP-provided multihoming*: The large-scale network service is provided by one or more third parties (like commercial ISPs). This service, whose requirements are commonly expressed by service level agreement (SLA) clauses, is characterised by offering well-provisioned channels, i.e., ones exploiting network paths with some level of redundancy, good availability and performance. It permits both primary

connectivity of the applications to the large-scale network and end-to-end communication between CII application endpoints. Moreover, it is supposed that CII control applications manage large-scale connectivity through multihoming in order to get better performance and fault-tolerance in their end-to-end communication [6]. This means that, in practice, CII network infrastructures already provide a number of redundant large-scale access links to applications by using one or more different ISPs.

- *Large-scale network provides timely communication to control applications in fault-free cases*: CII operations can be mission-critical and have to be accomplished within some known time window, so communication steps that are expected to occur during the execution of some CII operation have to be done in a timely manner. In the normal case, when there are no failures, the existing network service already has the ability to provide timely communication.

- *Large-scale network failures affect the communication timeliness*: Failures on the communication affect the expected behaviour of distinct network components such as routers and links between them, which together provide timely communication channels in a failure-free scenario. Accidental timing failures may be caused by a variety of events, from those motivated by planned human intervention (e.g., scheduled maintenance operations in the ISPs' backbones) to unexpected crash of some network components (e.g., due to router hardware problems or software bugs). In addition to accidental reasons, failures may also be originated by DoS attacks launched by outsiders in the Internet, representing a more serious threat. The CII network is usually implemented as a VPN on the ISP(s) infrastructure(s), but network resources are usually used also to provide Internet service, so DoS attacks in the Internet may cause delays on the CII network, leading to deadline violations.

3.2.2.2 CIS-CS Design Rationale

Let us now describe our solution to leverage timeliness for end-to-end communication over large-scale networks as demanded by the applications that participate on the process control and monitoring operations in nowadays CIIs. This solution constitutes the *CIS Communication Service (CIS-CS)*. Firstly, we explain the CIS-CS design as a service offered by a building block of the reference architecture conceived to enhance dependability properties in CIIs, the CRUTIAL Architecture [114, 20]. Next, the CIS-CS design is presented in more details focusing on its internal structure, showing its main components and their features.

CIS-CS in the CRUTIAL Architecture. The CRUTIAL Architecture models a CII as a WAN-of-LANs. In this model, depicted in the Figure 2.1, local networks that integrate the CII (LANs), like control subnetworks, are interconnected by a large-scale network (WAN). This model is interesting because it presents at a conceptual level common CII features as legacy control networks and interconnections between critical (e.g, control network generated traffic) and non-critical traffic (e.g., traffic generated by conventional applications of corporate offices).

Essentially, at the gate of each LAN in the CII, a robust special protection device is positioned, the CIS. A CIS is designed to be intrusion-tolerant by redundancy and incorporating a number of particular mechanisms. It provides two basic services. The protection service (CIS-PS) is intended to protect the LANs from invalid traffic by enforcing security policies on incomming/outgoing traffic, delivering various levels of trustworthiness to the corresponding protected LAN. *The communication service (CIS-CS)* provides timely and reliable communication between pairs of LANs through the WAN, being our focus here. In the text we will call each CIS a *CIS-CS node* to emphasize that here we are only interested in the communication service. We will also abstract from the fact that the CIS is replicated. The operation of a CIS sending a message to another CIS (or to a host inside a LAN) involves avoiding that several replicas send the same message, which can be done using the mechanisms explained in Section 3.3.1.4.

Normally, two CIS can exchange data across the WAN directly, using the paths provided by the IP routing infrastructure (*direct paths* for short). However failures in the network might cause a direct path to be slow or unavailable. In this case, the CIS-CS would have the alternative to relay the message over indirect paths that would rather be compliant with CII application timely constraints by using other cooperating CIS-CS entities.

CIS-CS has to satisfy three properties: *reliability* (messages have to get their destinations), *security* (messages have also to be transmitted from legal nodes and their contents undisclosed during transmission) and *timeliness* (messages have to arrive in time). The two first properties can be assured basically by resorting to retransmissions and cryptography. The third is the most complex to deal with: as already said, CIIs are clients of network services contracted from commercial service providers that are immersed in an unpredictable (in terms of timeliness) environment as the Internet.

In this work, we focus our attention on the problem of enforcing the timeliness property. However, we do not forget the other two (reliability and security). Actually, both properties are evenly met in the CIS-CS design and implementation as long as message retransmission (like in TCP) and HMAC mechanisms (like in IPSec) are aggregated to respectively manage reliable transmission and message authentication.

CIS-CS Design. Besides the CRUTIAL Architecture, there are other key aspects that influence the internal design and later architecture of the CIS-CS.

- *Overlay Network*: Our solution is based on the overlay network routing paradigm. Following this paradigm, CIS-CS nodes are the nodes of the overlay, which create a virtual (or overlay) network on top of the underlying IP-level routing infrastructure. Messages can be relayed through CIS-CS nodes, offering a set of alternative channels on top of the existing direct paths, augmenting the chances of obtaining timeliness and fault-tolerance, preserving both the semantics of control applications and the safety properties of control operations.
- *Multihoming*: CIS-CS nodes are connected to the WAN via a multihoming scheme [6], by using distinct access links provided by different ISPs. As a result, the CIS-CS benefits from improved availability and performance. A multihoming configuration can be adjusted

according to two basic parameters: the number of ISPs used in the multihoming scheme and the correlation of resources among those ISPs. The first is obvious. The second is related to the possibility of two ISPs sharing equipment (e.g., routers) or physical links. Our objective is to provide certain properties even if failures occur, therefore it is important to have as low as possible correlation between ISP resources.

- *End-to-end solution via one-hop source routing*: For simplicity without lacking the power of overlay networks, CIS-CS considers one one-hop source routing (like RON [8]). In this approach, an overlay route is entirely defined at the sender of a message and composed of, at most, one simple intermediary CIS-CS node, that relays the message directly to the final destination. Routes are computed and first selected according to a metric of *latency* (round-trip time), which indicates the two-way time to communicate between a certain CIS-CS source node and another CIS-CS destination. This approach incurs on the cost of collecting network performance information, whereas not using randomness to select overlay paths, which is known to perform worse with respect to overlay path selection strategies that are latency-driven [10].

- *Probing-based for non-painful deployment*: Besides selecting overlay channels according to network performance information, a probing-based solution is necessary for two more reasons. First, we do not have environment flexibility, control of the utility network nor previous knowledge of network topological information to strategically deploy overlay nodes, such that a best-fit performance and fault-tolerance overlay channels may be achieved. It excludes the application of integrated services at network level. Also, this prevents the possibility of the application of overlay strategies aware of network topology data as in [58]. Second, our environment of concern may suddenly change as long as it is subject to a variety of DoS attacks. This aspect violates the basic assumption in which topology-aware routing schemes rely upon.

- *Judicious spatial redundancy*: Depending only on one overlay channel to push control messages may be an unfortunate strategy, since it is likely that some critical message, mainly one with a very short deadline, may fail to get to its destination because of some irreversible failure to successfully communicate. Therefore, it is crucial to account an incremental amount of *backup virtual channels* rather than using only one overlay channel, thereby affording spatial redundancy. Up to here, exploring all possible overlay paths seems appealing because it may lead with high probability to timely communication. However, it works at the expense of a high cost, namely going toward flooding the network. A pure multi-path approach [100] is not enough as long as there is such a lack of a metric-based selection, leading the solution to be good for short-term congestion losses but problematic for long-term ones [10], a possible symptom of DoS attacks.

Using latency as the main metric to compose spatially redundant channels does not mean always exploring the overlay path with the best performance benefit as in RON [8]. That is because we are just seeking to get messages from one side to another in time. At the same time, just using the best overlay channel has a consequence of disregarding the potential diversity and redundancy provided by overlays across distinct deadlines. So, it is fairly possible to expand our design to incorporate an additional subset of overlay channels that do not offer the best latency but that

permit messages to arrive in time, while even allowing for the best overlay channel being explored in some limit case. This feature is primarily interesting to enhance availability of overlay paths. Besides, it serves to further improve the fault-tolerance aspect of the whole communication, being necessary some supplementary information on the degree of failure uncorrelation across all candidate overlay channels. The more failure uncorrelated to the primary channel the backups are, the more likely to be utterly successful for timely transporting the message the communication service is.

3.2.2.3 The Calm-Paranoid Algorithm

The Calm-Paranoid algorithm (CP) is the overlay path selection algorithm and the core building block of the CIS-CS. The CP algorithm is mainly based on round-trip time (RTT) measurements. The RTT is the elapsed time to communicate a single message from a sender CIS-CS node to another CIS-CS destination and back. The *RTT integrated estimate* used by the CP algorithm is composed of both RTT values and variation estimates taking into account network performance changes. We describe in detail how this estimate is derived in the Section 3.2.2.4.

Definition of the Channel Table. Let us define a finite set of CIS-CS nodes as forming an *overlay network* $O = \{o_1, o_2, \cdots, o_n\}$, with $|O| = n$. A node $o \in O$ has a unique identification in the overlay *oid* and gets access to the large-scale network through a multihoming scheme composed of k redundant links, each one from a distinct ISP x ($x \in X = \{1, \cdots, k\}$). A *channel* can be: (1) either a direct path between the sender and the destination node using an ISP; (2) or a combination of two of these paths, one from the sender to an intermediate node through an ISP, and another one from the latter to the destination through the same or another ISP. We use the notation o^x to denote that, given a certain channel, node o sends data data via ISP $x \in X$. An example of a channel is from p to q passing through intermediate node r using ISP 1 from p to r and ISP 2 from r to q is: $p^1 \rightarrow r^2 \rightarrow q$.

An overlay node o stores RTT measurements and failure information of overlay channels in a *channel table CT*, which is periodically refreshed. We define the channel table CT as a data structure with two levels. The first is the set D_o of destinations for which o can send messages, i.e., $D_o = \{o_1, o_2, \cdots, o_n\} \setminus \{o\}$. For each o' in this set there is a second-level structure: a channel table instance $T_{o'}$. A *channel table instance* $T_{o'}$ stores a channel table entries corresponding to the overlay channels that connect the table owner o to another overlay channel $o' \in D_o$. A *channel table entry* is composed of four fields: an index i (a positive integer starting from 0), a virtual channel field c with at most one intermediary overlay node, a flag $faulty$ indicating whether the overlay channel is faulty or not, and the associated channel RTT integrated estimate rtt. Table entries are sorted by the rtt field in ascendent order. The *size* of the channel table instance $T_{o'}$, $|T_{o'}|$, is defined as the number of entries it contains.

Figure 3.2 gives an example of a channel table T for a node $p \in O = \{p, q, r, \cdots, z\}$ ($|O| = n$). Any CIS-CS node in the example overlay O uses 2 redundant access links provided by 2 different ISPs, identified as ISP 1 and ISP 2 (i.e., $k = 2$ and $k \in X = \{1, 2\}$). The set

$D_p = \{q, r, \cdots, z\}$ is the first-level structure of CT (top). The channel table instance T_q ($o' = q$) is the second-level structure (at the bottom). It represents the RTT-ordered set of overlay channels connecting p to q with at most one intermediary overlay node. Three overlay channels are explicitly exhibited in the figure. For example, channel #0 (identified as faulty) connects p to q through ISP 1 with RTT estimate of 100ms. Another channel #n (labeled as correct) with RTT estimate of 220ms connects p to q using node r as intermediary, through both ISPs 1 and 2.

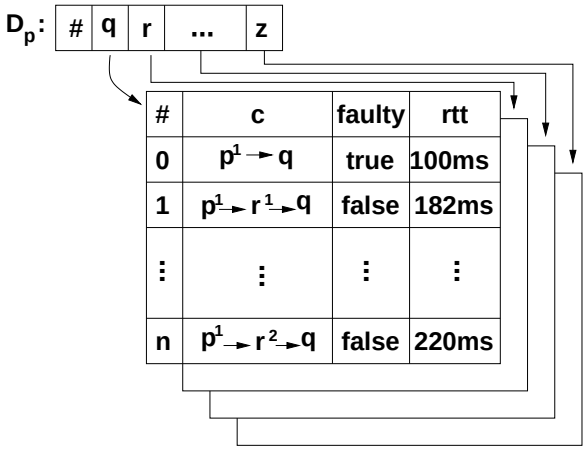


Figure 3.2: Generic channel table T of node p (channel instance T_q in detail).

The algorithm considers that a message m with deadline $m.d$ has to be sent over the network within a number of tries n. This parameter n is an estimate of the maximum number of retransmissions that can be done before the deadline expires. The algorithm essentially operates in two modes:

- *Basic or "calm" mode*: send m though a channel c such that it can still be possible to send m though faster channels if c fails and m be delivered before $m.d$;
- *Advanced or "k-paranoid" mode*: in addition, use a set of k auxiliary channels as less correlated to c as possible that permit m to be delivered before $m.d$.

In the first case, the motto is that *"messages don't need to reach their destination the soonest possible, they need to arrive in time"*. This means that we can initially use channels that do not provide the lowest RTT but that allow delivering the message before its deadline. These "calm" channels are used while no confirmation of delivery is received, until eventually as a final resource, the best-latency overlay channel is used. In the second case, the motto is that *"IP networks provide no guarantees so we have to take some preventive measures"*, what forces us to additionally have an alternative plan: to send messages through k uncorrelated channels.

This basic algorithm leaves open a set of design options. Two important ones are the following. The first is the possibility of *using a greater number of paranoid channels* than only one as specified before. Clearly, there is a trade-off at this point, since it is possible to have more spatial redundancy by pushing messages to the network using many virtual channels whereas

incurring on more communication costs. An immediate principle is to limit the number of useful paranoid paths by the amount of different WAN access links provided by multihoming. A result from [6] helps to address this issue pointing to 3- or 4-redundancy at maximum to manage good performance and reliability, what leads us to apply a threshold of 3 auxiliary channels.

The second point is *to define a base strategy to select alternate paranoid channels as most as possible uncorrelated to the main calm channel*. Another trade-off happens here: the more granular the analysis of correlation is, the more costly to deploy. Given our environment features, using network or physical topology information is out of the question. On the other hand, it is quite possible to use overlay-level information as a second criterium, since it is cheaper and reasonable to get, even though it may lead to selection mistakes in terms of expected correlation information. For example, it is very likely that 2 apparently uncorrelated overlay channels over distinct infrastructures of 2 ISPs share the same network-level resources. An additional option is to explore a second application metric (e.g., loss rate) along with latency. However, it is not clear whether this strategy would bring more benefits than another based on overlay-level data, specially in cases where the choice would be restricted to channels over one ISP infrastructure.

Algorithm Description. The Calm-Paranoid Algorithm executed by node o is presented in Algorithm 7. The node has a channel table instance T with size $|T|$. T stores the table entries relative to all overlay channels with at most one intermediary node that connects o to another CIS-CS overlay node $o' \in O$ ($o' \neq o$). Recall that the contents of T is ordered by the RTT field. We designate table entry j by $T[j]$ and refer the contents of its fields using the dot operator ('.'): $T[j].c$, $T[j].faulty$ and $T[j].rtt$ are respectively the channel, faulty flag and RTT estimate fields.

Let $send(\langle \texttt{TYPE}, m \rangle, c)$ be the communication primitive to send a message m of TYPE through a single overlay channel c whose destination is the node o'. Message m contains its deadline information in the field $m.d$. At the algorithm level, the send primitive is mapped into two operations that can be called at distinct moments: LAN_*send*$(\langle \texttt{TYPE}, m \rangle)$ and WAN_*send*$(\langle \texttt{TYPE}, m \rangle)$. They refer to the service interfaces provided by the CIS-CS to enable control application messages to be pushed out from the CIS to the LAN or WAN.

The main part of the algorithm is executed by a source node o upon the WAN_*send* operation being invoked with the objective of transmitting a message m with deadline $m.d$ to the destination o'. This can happen at two moments. Initially, upon o receiving a message m from the LAN control application to be sent through the large-scale network for the first time. Secondly, upon o detecting that some previously activated timeout is expired and that m has to be retransmitted. On the other communication side, upon m is received from the WAN, a correspondent ACK message is echoed by using the reverse way of the forward overlay channel and the data message m is passed along to the proper receiving control application as well.

When o invokes WAN_*send* for a data message m, it first calculates the expected number of tries n to send m through the large-scale network according to the current information known by o about the network stored in its table T (lines 1 to 9). The algorithm seeks to predict the maximum possible number n of future retransmissions of m according to the reference deadline value $m.d$ by progressively taking advantage of the distinct non-faulty overlay channels available in the entries

Algorithm 7 Calm-Paranoid Algorithm for a CIS-CS overlay node o

upon invoke WAN_*send*($\langle$DATA,$m\rangle$):

1: **if** ($\frac{T[k].rtt}{2} \leq m.d$ AND $\neg T[k].faulty$) **then**
2: $elapsed \leftarrow \frac{T[k].rtt}{2}$; $n \leftarrow 1$
3: **for** $k \leftarrow 1$ to $|T|$ **do**
4: **if** $((elapsed + T[k].rtt) \leq m.d$ AND $\neg T[k].faulty)$ **then**
5: $elapsed \leftarrow elapsed + T[k].rtt$; $n \leftarrow n+1$
6: **else**
7: **break**
8: **end if**
9: **end for**
10: **else**
11: LAN_*send*($\langle$NOK,$m\rangle$)
12: **end if**
13: $c \leftarrow T[n-1].c$; $RTT_c \leftarrow T[n-1].rtt$
14: *send*($\langle$DATA,$m\rangle, c$)
15: $P \leftarrow \{T' |\ T' \subseteq T, |T'| \leq k\}$, such that for each $0 \leq l \leq |P|$:
16: (i) $\frac{T'[l].rtt}{2} \leq m.d$
17: (ii) $T'[l].c$ has minimal correlation with c
18: $\forall p \in P$, *send*($\langle$DATA,$m\rangle, p$)
19: $m.d \leftarrow m.d - RTT_c$
20: *activate_timeout*(m, RTT_c)

upon timeout for m:

21: WAN_*send*($\langle$DATA,$m\rangle$)

upon receive $\langle$DATA,$m\rangle$ from LAN:

22: WAN_*send*($\langle$DATA,$m\rangle$)

upon receive $\langle$DATA,$m\rangle$ from WAN:

23: $\{c'$: reverse of the forward channel$\}$
24: *send*($\langle$ACK,$m\rangle, c'$)
25: LAN_*send*($\langle$DATA,$m\rangle$)

upon receive $\langle$ACK,$m\rangle$ from WAN:

26: **if** *cancel_timeout*$(m) = true$ **then**
27: LAN_*send*($\langle$OK,$m\rangle$)
28: **else**
29: LAN_*send*($\langle$NOK,$m\rangle$)
30: **end if**

of T, each of which with latency cost $T[k].rtt$. If not possible to perform any transmission of m with deadline $m.d$ (e.g., deadline is too short or there is no available non-faulty channels), i.e., $n = 0$ (line 11), o invokes the LAN_*send* operation with message error of type NOK to notify the LAN control application.

At least, if it is possible to transmit m just once, the algorithm defines the calm channel c as being the one that was later considered during calculation of the number of tries n and the

timeout RTT_c for potential retransmission (line 13). Through c, the message m is primarily sent (line 14). In addition, a set P of k auxiliary (paranoid) channels is defined where each channel holds two basic conditions (lines 15 to 17): the paranoid channel p must expectedly support timely communication of m, which can happen on average if half of its RTT estimate is equal to $m.d$ at least; further, as a weaker but desirable condition, the channel p presents minimal failure correlation with channel c in order to augment fault-tolerance of the whole timely communication.

The algorithm updates the message deadline information with the RTT provided by the selected calm channel (line 19). And, finally, it triggers a timeout for future retransmission of m (line 20). Such timeout for m is cancelled upon o receives a respective `ACK` message from the CIS-CS node o' if that timeout is not expired (*cancel_timeout* function returns *true* status). In this case, o sends to the appropriate LAN control application a notification `OK` message informing about the successful timely transmission. Otherwise, o notifies the control application with message error of type `NOK`.

3.2.2.4 CIS-CS Architecture

The Figure 3.3 displays the CIS-CS architecture, which is composed of 3 inline communicating modules that rely on a common network support. The latter is presented here as an abstraction of the standard TCP/IP network protocol stack that provides to all upper layer components the essential network connectivity and end-to-end communication services.

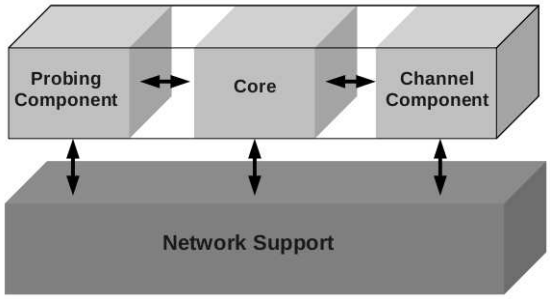


Figure 3.3: CIS-CS architecture.

In the figure, we identify the three CIS-CS components. They are the *Probing Component* (leftmost), the *Channel Component* (rightmost) and the *Core*. Basically, each component provides respective features for (i) collecting network performance information and detecting network failures; (ii) announcing probing local information and building overlay channels; (iii) enabling LAN and WAN interfaces for CIS-CS, performing both data forwarding and relaying operations, and managing the operations of Probing and Channel components. Such features are described in more detail as follows.

Probing Component

Network Probing. Each CIS-CS overlay node obtains RTT information primarily by running a simple ping protocol periodically with a *predefined probing time interval* T_p. In a normal execution, a CIS-CS node sends a special request message to some of its overlay probing peers containing the prober local time, which corresponds to the protocol start time. Upon receiving the request message, the CIS-CS peer just sends it back to the prober. Upon the prober receiving the response, it takes the difference between its current local time, which corresponds to the protocol end time, and the prior time stored in the message recently received.

The information collected by running the probing protocol is stored in a local *probing table*. Each *probing table entry* has the information fields *id*, RTT_e, RTT_{var} and *skew*, respectively for storing the peering node identification, the currently estimated round-trip time between the table owner prober and its peer, the RTT variation estimate, and the probing deviation time. The deviation time is a random seed taken according to the number of overlay nodes that is applied to T_p in order to make difficult for two different probers to initiate their ping protocol instances at the same time, preventing potential side effects on the probing activity results.

One possible optimization of this simple ping protocol is exploring an optimistic approach inspired by RON [8]. In this approach, a probing peer takes advantage of receiving a request message to start its own probing session rather than waiting until a next probing period happens. As a result, we can reduce the complexity of communicating steps from four to only three messages exchanged.

The RTT estimate is obtained using the following exponential weighted moving average formula:

$$RTT_e = \alpha.RTT_e + (1-\alpha).RTT_m$$

In this formula, the current estimate of the round-trip RTT_e (left side) is calculated from consolidating the previously obtained round-trip estimates (right side RTT_e) and the recent network RTT measure RTT_m. A common practice is, as a prudent measure, to give more weight to the previous RTT measurements than to the last single measure done. This makes possible to keep the current RTT estimate closer to the perception of a recent enough RTT performance history and farther from instantaneous network performance uncertainties. TCP's RTT estimation considers a value of $\alpha = 0.875$ [90], and RON considers for RTT estimation $\alpha = 0.9$. We have a less conservative and more sensitive approach at this point taking a weighted average factor with a value of $\alpha = 0.75$.

Besides simply estimating overlay channel RTT performance, we also derive the correlated estimate RTT variation. The idea, inspired by TCP's retransmission timeout calculation [90] and also employed by MONET's overlay architecture [9], is to use both measures in order to estimate a more realistic RTT value RTT', which will later serve as the retransmission timeout of the CIS-CS overlay channel selection strategy. Such value of RTT variation RTT_{var} is calculated just as the one of RTT estimate RTT_m previously shown (note: $\alpha = 0.75$):

$$\begin{aligned} RTT_{var} &= \alpha.RTT_{var} + (1-\alpha).|RTT_m - RTT_e| \\ RTT' &= RTT_e + 4.RTT_{var} \end{aligned}$$

Failure Detection. The probing ping protocol explained before also works as the CIS-CS network failure detection mechanism. Every time a node p sends a probing message to a node q, it waits for the response until a timeout t_f. If the response is not received in time, the overlay channel that connects p to q is labeled as faulty; otherwise, the channel is correct. If p considers the channel faulty, then p uses the double of the current RTT estimate RTT_e as the next RTT measure RTT_m to calculate the next RTT_e for that channel. Following related works (e.g., [8, 55]), we set the default value of t_f to 3 seconds.

The CIS-CS' interpretation on the failure of network channel is quite categoric, but the philosophy behind such eagerness can be justified. In order to maximize the chance of improving timeliness for any single critical message communication, the CIS-CS needs to *promptly* know about the network changes that can affect the availability of candidate timely channels. We note that such CIS-CS principle for failure detection is a shortening of what is done in RON, which purely concerns with the availability of overlay channels used, but not with timeliness. Thus, while the latter engages a detection by probing the same overlay channel four consecutive times until concluding whether such is available or not, the former abbreviates this process by performing just one probing.

Channel Component

Probing Information Propagation. The probing component provides information about direct paths only, i.e., about channels connecting pairs of overlay probing peer nodes over direct paths. Therefore, to compose end-to-end virtual channels with one intermediary overlay node (i.e., indirect overlay channel), it is necessary to additionally execute an overlay routing protocol, where overlay nodes advertise their probing local information to the rest of the peering nodes. This information is related to existing direct virtual channels connecting the advertising node and its probing peers.

Similarly to the probing protocol, a CIS-CS node periodically runs an overlay route announcing protocol, but to simply inform peers about the existence of up-to-date direct virtual channel information in its probing table. Just as the ping protocol, the periodicity of such announcement protocol is fixed at some predefined interval T_c.

Alternatively, such notification may work in a different way to reduce the overhead associated with it. In this case, the announce is conditioned to the outcome of a prior probing protocol instance. It may be launched by the overlay node that runs the probing session depending on the return status of the update operation on its probing table. Therefore, upon such prober receives a probing response message from this peer, it verifies whether the corresponding probing table entry is either updated or not. If so, then the prober turns into an advertiser that informs all CIS-CS

overlay nodes – but its probing peer – about the just updated direct overlay channel information. Some update relaxation criterium may also be applied here to reduce overhead.

Overlay Route Composition. Each overlay node p keeps locally one generic channel table (see earlier explanation in Section 3.2.2.3). A channel table entry holds information about all overlay channels (direct or not) that are known by p. Entries related to direct overlay channels from p to each distinct overlay node q (i.e., $p \rightarrow q$) are automatically composed by incorporating information from the corresponding probing table entries.

Indirect channels for p to q using r as overlay transit node (i.e., $p \rightarrow r \rightarrow q$) are composed by merging information of two different direct channels: the one immediately provided by p's own probing table ($p \rightarrow r$) and the other previously provided by r's channel component service to p corresponding to the direct channel from r to q (i.e., $r \rightarrow q$). RTT related information (both RTT performance and RTT variation) is trivially merged by summing the RTTs of $p \rightarrow r$ and $r \rightarrow q$. Failure information of an indirect channel $p \rightarrow r \rightarrow q$ is provided by performing an `OR` operation on the failure information of $p \rightarrow r$ and $r \rightarrow q$. This means that it is enough one of the composing direct overlay channels to be faulty to the composed indirect channel to be faulty too.

Core Component

Internal Dispatcher. The Core is responsible for managing some CIS-CS internal tasks that requires orchestration between the other two CIS-CS existing components. The Core acts as the interface between the Probing and Channel components whenever needed. For example, for the Channel component to compose or to update indirect overlay channels, it is required the presence of probing local information served by the Probing component.

Network Zone Interfacing and Data Forwarding. On the CIS-CS side, the Core still plays another key architectural role. It is directly involved in the materialization of the CIS concept as described by the CRUTIAL Architecture. Accordingly, the CIS captures a central features for each distinguished CII utility subnetwork, defining two distinct network zones, namely the CII LAN and WAN portions.

The Core supports the implementation of such CIS concept by providing two special interfaces, the LAN and WAN dispatchers. *The LAN Dispatcher* is the interface used to push out and pull in control messages from and to LAN. It is referred to the inner zone protected by the CIS. *The WAN Dispatcher* is the interface used to push out and pull in messages from and to the WAN. It is referred to the outer zone with which the CIS interfaces (conceptually, the large-scale network). The fundamental part of the CIS-CS is situated in the WAN interface, which is responsible for the data forwarding and relaying feature, implemented by the Calm-Paranoid Algorithm.

3.2.3 Fosel for Mitigating DoS Attacks

Denial-of-service (DoS) attacks are one of the key threats against availability in the Internet. The increasing numbers and variety of DoS attacks and distributed-DoS (DDoS) attacks have become a growing annoyance to the network administrators and ISPs [81, 94]. Today the Internet has also become an emerging technology for remote control of industrial applications (e.g. power plants controllers), for which availability is an essential parameter for correct execution [30]. DoS attacks can cause significant disruptions and damages on such control networks.

Filtering techniques are one of the main approaches that reactively protect application sites from DoS attacks [47, 72, 89]. Filtering techniques use the characterization provided by detection mechanisms to filter out attack streams. In some cases malicious packets can be identified by clearly-defined metrics, e.g. obviously wrong source addresses or other obvious errors in the packet header. Such packet flows can be filtered at routers. Examples include dynamically deployed firewalls [35], as well as some commercial systems [11, 78]. While filtering techniques may be appropriate for several types of DoS attacks, they are not always applicable or practical due to the following limitations:

- The accuracy with which legitimate traffic can be distinguished from the DoS traffic is low.
- 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.
- If attackers obtain an account of one legitimate user, they can use a legitimate source IP address to attack the system.
- Filtering techniques need to process the address of all packets in order to drop or accept the packets. Unfortunately such processing is time-consuming. This solution itself can reduce availability and performance of the system.

An efficient and well-suited filter must meet three main goals: (1) accurate attack detection, (2) effective response (dropping or rerouting) to reduce the flood, and (3) precise identification of legitimate traffic and its safe delivery to the destination. A Fosel filter (*Filter with helping an Overlay Security Layer*) is an efficient and well-suited filter that meets all three important conditions.

The Fosel architecture uses overlay networks as many other approaches. Overlay networks have been proposed to protect application sites against DoS attacks [68, 105, 124]. These overlay networks are also known as proxy networks [122, 124]. The key idea is to hide the location of vital nodes of the architecture (the node which delivers the final traffic to the target) behind a proxy network, using the proxy network to mediate all communication among application sites (or between users and the application), thereby preventing direct attacks on the application.

The Fosel architecture is independent from DoS attack patterns, so we do not worry about changing attack patterns or accuracy detection of malicious traffic. In fact the Fosel architecture

is not based on detection of malicious packets, since it only accepts traffic from some special overlay nodes (green nodes) and discards the traffic of all other nodes (solving the first and second problems). In the Fosel architecture an overlay network within the secret green nodes makes a secure way to deliver data to the target. The final filter only accepts data from secret green nodes which ensures that attackers cannot use spoofed IPs to attack the target (solving the third problem). In case of a DoS attack, the final filter rather than processing all arriving messages, processes only a subset of the received messages. To guarantee that legitimate messages are not lost, multiple copies of a legitimate message are sent. This technique reduces processing time noticeably (solving the fourth problem). Firewalls with access control techniques are used to allow only legitimate users to use the overlay network.

In this section, we discuss how an efficient and well-suited filter can be designed with the of help of an overlay layer to mitigate DoS attacks. In fact Fosel is a dependable and secure architecture to protect application sites from DoS attacks. Fosel architecture is adaptable and compatible with the rest of CRUTIAL architecture.

The section also provides an evaluation of the Fosel architecture by simulation. Two approaches can be employed to attack the system: a) an attack against an application site and b) an attack against the overlay network. Through simulation we show that by employing the Fosel architecture, attackers have a negligible chance to saturate the application sites with malicious packets. For attacking the overlay, we evaluate the likelihood that an attacker is able to prevent communication among application sites. Results show that even if attackers are able to launch massive attacks against the overlay, they are unlikely to prevent successful communication. For example, in a static attack case (focused attack on a fixed set of nodes), DoS attacks are countered completely. In a dynamic attack case, if attackers can launch attacks upon 50% of nodes in the overlay, still 90% of communications among application sites are successful.

3.2.3.1 A Background on DoS Attack Defense Techniques

Many research projects, papers and commercial products attempt to tackle DoS problems. But due to the framework of this work, only those that are based on filtering and those that use overlay networks are reviewed here. See [43, 79] for a more complete survey of all DoS defense approaches.

Filtering based techniques take in a packet and decide whether to accept or to reject (drop) it. So the corner stone of filtering techniques is DoS traffic detection. There are number of statistical approaches for detection of DoS attacks. In [46] entropy and chi-square statistics are used to differentiate between attack and normal packets, while [69] computes the conditional legitimate probability of a packet (the likelihood that a packet is legitimate given a baseline nominal traffic pattern). In D-WARD approach [80], attacks are detected by the constant monitoring of two-way traffic flows between the network and the rest of the Internet to detect discrepancies from normal flow models and rate-limit mismatching flows in proportion to their aggressiveness. Some other statistical detection techniques of DoS attacks include the use of MIB traffic variables [25], IP addresses [93] (which assumes that attack traffic uses randomly spoofed source addresses), IP ad-

dresses and TTL (Time To Live) values [64] and TCP SYN/FIN packets for detecting SYN flood attacks [121].

The filtering policy architecture and location of filters can be different. For example the D-WARD defense system [80] is deployed at source-end networks while DPF [89] can be distributed in Internet core routers to proactively stop packet flows with obviously wrong source addresses, and meanwhile reactively tracing back the attacking source. Mahajan et al. [74] provide a scheme in which routers learn a congestion signature to tell good traffic from bad traffic. The router then filters the bad traffic according to this signature. Furthermore, a pushback scheme [63] is given to let the router ask its adjacent routers to filter the bad traffic at an earlier stage. Ingress filtering [47] and SAVE filtering [72] also are router based filters.

Some DoS defense techniques are based on the location hiding of the application. Normally overlay networks are used to hide the location of the application site. SOS (Secure Overlay Services) [68] is an example of such approaches. In the SOS architecture, access requests will be authenticated by SOAP nodes and then routed via the Chord overlay network [106] to one of the beacon nodes and then to one of the servlets, which then forwards the requests to the target site that is protected via filters. Although SOS has simplified the filter rules to reduce processing time, still processing time for large DoS attacks (DDoS) is a challenge where the Fosel architecture tries to avoid this problem. OPL (Overlay Protection Layer) [16] is another approach that uses overlay networks. In the OPL architecture, communications are done via normal network in case of no attack. However, when an attack is started against an application site, the application site is disconnected from the public network and connects to its secret nodes via private networks, then inform all other application sites to connect to it via the overlay network. Now a request access is routed via the overlay network to the secret nodes of the target and then the secret nodes deliver it via the private network to the target. However, the high cost of private networks is a main challenge of the OPL architecture. Some other works related to DoS defense techniques via overlay networks can be found in [105, 122, 123, 124].

3.2.3.2 Fosel Architecture Description

The Fosel architecture is a proactive approach to safeguard application sites from DoS attacks. The goal of the Fosel architecture is to provide a well-suited filter that can be installed on destination routers to protect application sites from DoS attacks. Fosel allows communication only among confirmed application sites, i.e., sites that have given each other a prior permission. Typically, this means that any packet must be authenticated through the Fosel architecture before the packet is allowed to be forwarded to the destination. The Fosel architecture can be used for a set of distinct nodes that communicate together via the Internet. For example the offices of a company that is distributed in the large area (e.g., various countries of Europe), or a CII composed of different facilities.

The Fosel architecture employs an overlay network that is composed of nodes that communicate atop the underlying network. The IP addresses of all overlay nodes and application sites are known to the public and also to the attackers. However, a certain set of nodes is kept secret

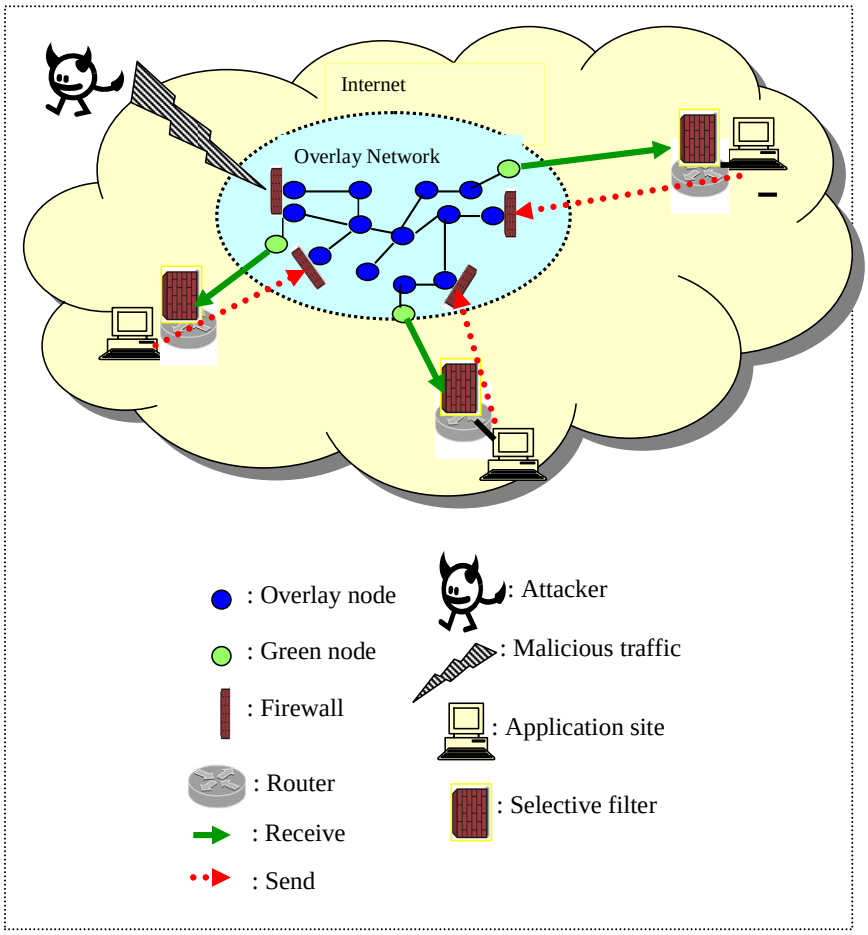


Figure 3.4: The Fosel architecture.

from public and attackers. We call these secret nodes, green nodes.

Attackers can launch DoS attacks against application sites and overlay nodes from different points of the Internet. As any application site needs to authenticate before connecting to the overlay, 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 Fosel architecture that protects application sites from DoS attacks. The following elements play role in the Fosel architecture:

1. Application site: the legitimate sender or receiver (the target).
2. Firewall: commercial firewalls that are installed on the overlay nodes.
3. Overlay network: a logical network that communicates atop of the underlying network.
4. Secret green nodes: the hidden nodes in the overlay that deliver the final traffic to the target.
5. Selective filter (receiver's filter): a well-suited filter that protects the target from DoS attacks.

The elements from the first index until fourth index help the fifth element to make a suitable filter that meet all three necessary mentioned conditions. In the following subsections the architecture is explained step by step.

Architecture and Operation Design Rationale

The goal of the Fosel architecture is to provide a immune sub-network from DoS attacks on the Internet among the application sites of a company. Any application site to send a message or a request for any other application sites connects to a random overlay node. The overlay node analyzes and verifies the message by authentication techniques. If the message is authorized successfully it is allowed to enter inside the overlay, otherwise it is dropped by the overlay node. Hence, at a very basic level, we need the functionality of firewalls in the overlay 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.

After a legitimate message is verified by the overlay node it is forwarded and routed through the overlay network to the secret green node of the application site. Then the green node delivers the message to the application site (final destination) through the selective filter.

In case of a DoS attack against an application site, the green node of the application site delivers multiple copies of the message to the application site, otherwise (no DoS attack) it delivers only one copy of the message to the application site.

The Selective Filter

The IP address of an application site is sufficient information for adversaries to attack the application site by a DoS attack. To prevent such attacks, a selective filter can be constructed and installed on destination's routers. Hence, malicious and illegitimate packets are dropped by the selective filter. We assume attackers do not have access inside the destination's routers and filter region. Past history indicates that it is significantly more difficult for an attacker to access a router and poison the filter (compared to, e.g., a web server or a desktop computer) [68]. Hence, we assume that attackers cannot penetrate inside the router.

Selective filter accepts only those packets that come from the green nodes of the application site. Hence selective filter drops all other packets whose source address does not match the address of its green nodes. A selective filter does not need to verify a signature and decrypt data. It just needs to see the IP address of a message. If the IP address belongs to green nodes, the message is accepted, otherwise it is dropped. This simple and straightforward approach has two main advantages: (i) when an application site changes its location and moves to another location (change IP) the filter does not need to be notified and then modified, because it is independent from users location (the application sites IP); (ii) attackers cannot use spoofed IPs to attack the

target. As the IP of the application sites is clear to the public, attackers can use their IPs to attack the target, but as the selective filter only accepts data from secret green nodes, attackers cannot attack the target with spoofed and bogus IPs.

Because of the small number of such filter rules and simple nature (filtering only on source IP address), the time for processing a packet is decreased to at least 2/3 of the time required by other filters (filtering on signature and data decryption). Suppose any read, decrypt and compare operation takes one T. Other filters need 3T (read, decrypt and compare) to process any packet while selective filter needs 2T (read and compare).

The second task of the selective filter is that in case of DoS attacks rather than processing all arriving messages, it processes only a subset of the received messages. To ensure that legitimate messages are not lost, the green nodes of the application site transmits multiple copies of each message. Upon first observation, this technique appears to worsen the DoS attack by increasing the amount of traffic received by the application site (multiple copies of each message). However, due to asymmetry in the cost of sending versus processing messages, the large reduction in processing cost outweighs the increase in network traffic during an attack. For the green nodes, sending multiple copies of message is relatively cheap. To alleviate the potential high cost (increase traffic by multiple copies), the selective filter randomly drops any message with probability (1-P), where P is the probability of message acceptance. For example when $P = 0.6$, it means that any message has chance of 60% to be processed. In other words 40% of messages are dropped without any processing. With this approach most bogus messages are discarded and dropped without any processing, while due to sending multiple copies of a message, the legitimate message has a high chance to be processed.

Let us mathematically calculate the effectiveness of the second task of the selective filter. Let the time required for any action be a unit of time, T. For example any action like read, decryption or comparison is done in a T. A straightforward filter (direct application access) needs 3T for processing any packet (read, decryption, and comparison). The Fosel architecture needs 2T (read and comparison) due to checking only the source address of a message.

Let α be the attack rate (i.e., the number of attack messages arriving at the receiver per T) and s be the number of arriving legitimate messages per T. When a straightforward filter is used (direct application access), the time required for processing all arriving messages is $T_{straightforward} = 3(\alpha + s)T$. When the Fosel architecture is used this time is $T_{Fosel} = 2P(\alpha + c.s)T$, where c is the number of message copies. Thus, when the attack rate α is much larger than s and c, then the attack rate is diminished by a factor of P. For instance when $P = 0.5$ and $c = 10$ (miss rate less than 1%) we would expect to process only $1/3$ of the DoS packets.

Green Nodes

In short, green nodes are secret nodes of the overlay network. The location of green nodes is kept secret. Only application sites know the location of green nodes, and other nodes as well as attackers do not know which nodes are green nodes. The overlay network, along with the

green nodes, helps the application site to have an effective and fast filter (selective filter) to drop malicious packets.

To activate a green node, the application site *randomly* chooses an overlay node and then sends a message to the overlay node that has been selected as a secret green node. The application site informs the secret green node of its task. Each application site may have a few green nodes. The selective filter in the destination is set to only pass packets whose source address matches the address of the green nodes of the destination. In fact green nodes are the final nodes in the architecture that deliver legitimate traffic to the destination through the selective filter. As green nodes are chosen randomly, the location of green nodes is kept hidden from attackers. Secret green nodes give randomness and anonymity to the architecture that makes it difficult for an attacker to find final delivery nodes and attack them specifically.

A green node, upon receiving a message from the overlay, forwards it to the destination through the selective filter. In case of DoS attack it delivers multiple copies of the message.

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

Attacking the Overlay

Attackers can attack the overlay from a variety of points of the Internet simultaneously. However, these attacks have no influence 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.

Although green nodes are the secret nodes of the architecture and attackers do not know their location, as green nodes are nodes from the overlay network and attackers can attack any overlay node, it is possible that green nodes are attacked accidentally. However, if a green node identity is disclosed and the green node is targeted as an attack point, that green node is simply removed from the overlay and the target (the application site) chooses another node randomly as a new green node.

Some Additional Points

Below are some additional information about the current Fosel architecture:

- The Fosel architecture utilizes the Chord network as an overlay network. 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. The Chord network has some valuable properties that match the needs of the Fosel architecture:
  - In a Chord network, to find a key X from any node, O(log N) steps are required. 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 adversaries 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.
  - The Chord network is more dependable than other overlay networks [14, 15, 39].
- An application may have several green nodes (redundancy in green nodes) due to the fault tolerance policy.
- Anomalies (DoS traffic) are detected when the current system state differs from the model by a certain threshold. If the arriving rate at a destination is more than β (threshold), the application site (destination) can understand that it is under attack and sends a message to its green nodes that it is under attack.
- The receiver may process multiple copies of a legitimate message. A nonce can be used to quickly discard duplicates. Nonces can be analyzed at the destination after the acceptance of the message by selective filter (the message has passed the filter).

Summary of a Packet Transmission

To summarize, the sequence of operations in Fosel for the send operation is explained in both cases with or without DoS attack.

- The sending operation in case of no DoS attack (attack-free):
  1. An application site sends its message to a random overlay node.
  2. The overlay node verifies the message through firewalls (authentication techniques). If it is ok, the message is accepted; otherwise it is dropped.
  3. The overlay node sends the message to the green nodes of the destination.

4. The green nodes deliver message to the destination through a selective filter.
5. The selective filter *checks source address of the message*: if it is from green nodes, it is accepted; otherwise it is dropped.

- The sending operation in case of a DoS attack against application site ”A”:

1. The application site ”A” (receiver) sends a message to its green nodes that it is under DoS attack.
2. An application site (sender) sends its message to a random overlay node.
3. The overlay node verifies message by authentication techniques (firewalls). If it is ok, the message is accepted; otherwise it is dropped.
4. The overlay node sends the message to the green nodes of the application site ”A”.
5. The green nodes *deliver C copies of the message* to the application site through the selective filter.
6. The selective filter *randomly drops messages (malicious or legitimate) with a given probability without any processing* and processes the remaining messages.
7. The selective filter checks the source address of the remaining messages: if the message is from the green nodes, it is accepted; otherwise it is dropped.

Applying Fosel to the CRUTIAL architecture

The selective filter of Fosel resides in the CIS, or in other words, the selective filter is a service offered by the CIS. The CIS provides the selective filter to protect the LAN from denial of service attacks, and furthermore to other services such as the Protection and Communication Services.

In the CRUTIAL architecture the CIS communicate through the WAN according to the WAN-of-LANs model. By applying Fosel to the CRUTIAL architecture, the overlay network of the Fosel architecture resides in the WAN. In other words the CIS communicate through the overlay on the top of the WAN.

Figure 3.5 illustrates how the Fosel architecture is integrated into the CRUTIAL architecture. The CIS captures packets and checks them according to the Fosel policies (in addition to other checks for other security policies). Then, it discards those packets disapproved by Fosel and forwards the remaining ones to the LAN.

The Fosel approach can be used by the Monitoring and Failure Detection service to adapt the CIS behavior to current network conditions. The detection of the DoS traffic is done inside the CIS. For example, the CIS monitors the network traffic and calculates the rate of arriving packets per specific time period (for instance for 3 minutes) and compares it with a certain threshold. If the arriving rate is more than what is allowed by the threshold, the CIS understands that the LAN is under a DoS attack and Fosel takes action to handle it.

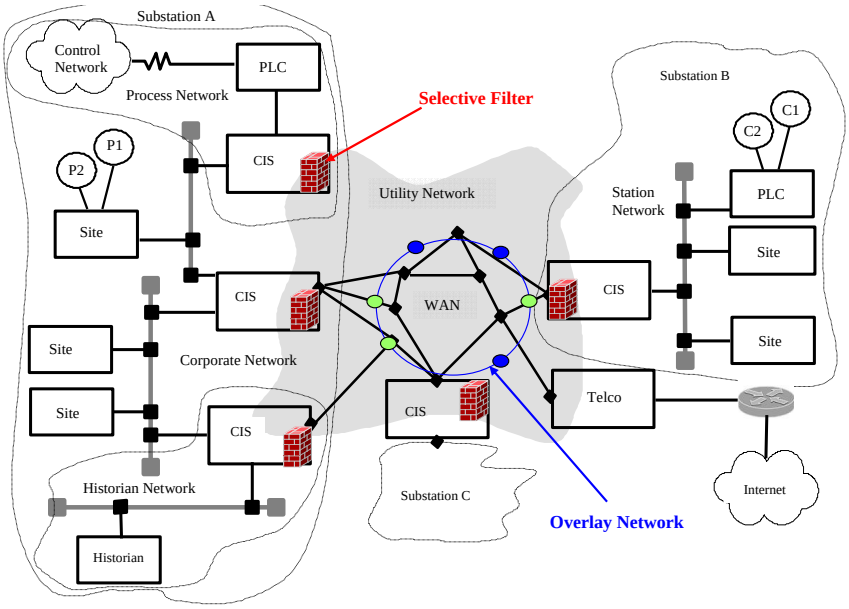


Figure 3.5: Applying Fosel into the CRUTIAL architecture.

3.2.3.3 *Performance Evaluation of Fosel*

In this section we analyze the performance of the Fosel architecture by simulation. First some points related to the setup of the simulation:

- Attackers know the IP of a set of overlay nodes and also application sites. So attackers can start DoS attacks against them.
- Attackers have a bounded and fixed amount of bandwidth to attack the architecture. For instance, attackers can attack a 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 application site has five green nodes.
- Twenty senders simultaneously send messages to an application site.

Implementation

We implemented a simulator to evaluate the Fosel architecture. To do this we implement the Chord protocol (main backbone of Fosel architecture), an attack toolkit, and the selective filter (destinations' filter).

- *Implementing the Chord network*: We have implemented the Chord network in an iterative style [106]. 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, e.g. node m, cannot contact another node, e.g. 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 both). 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 the basic structure of Trinoo [40] to generate both DoS and DDoS attacks in C++. In fact we implement two basic procedures for the attack toolkit: daemon and master procedures. We have several daemons that are controlled by a master procedure. Daemons 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.

- *Implementing the selective filter (receiver's filter)*: The selective filter utilizes q queues (total queues). Incoming messages (legitimate or malicious) are enqueued into queues at random. Then the filter selects k queues (K=p.q) at random for processing and drops (q-k) queues without any processing. For processing a message, the filter only checks the source address of the message. If the source address is an address of one of its green nodes, the message is accepted otherwise it is dropped.

Simulation Results

There are two possibilities to attack the system: attacking the application sites and attacking the overlay network.

Attacking the application sites

A sender's message can potentially not be processed due to the random selection of k queues from q queues (total number of queues). It is hence possible that due to the miss probability, no copies of a particular message are present in any of the selected queues. We call this a miss and the fraction of unique sender requests that end up missing the miss rate. So the miss rate is the number of unprocessed packets from legitimate senders per the total number of packets of legitimate senders.

$$miss\_rate = \frac{unprocessed\_senders}{total\_number\_of\_senders}. \tag{3.1}$$

Here, we show that by checking only a small number of queues, the miss rate becomes quite low. We can assume two states for an application site: a) unlimited length queues and b) limited length queues. We suppose that receiver has 20 queues (q=20) in total.

Queues with Unlimited Length. Figure 3.6 shows the miss rate for different values of selected queues (in the other words, different values of P, $P = k/q$) when the number of message's copies increases along the x-axis. K is the number of selected queues.

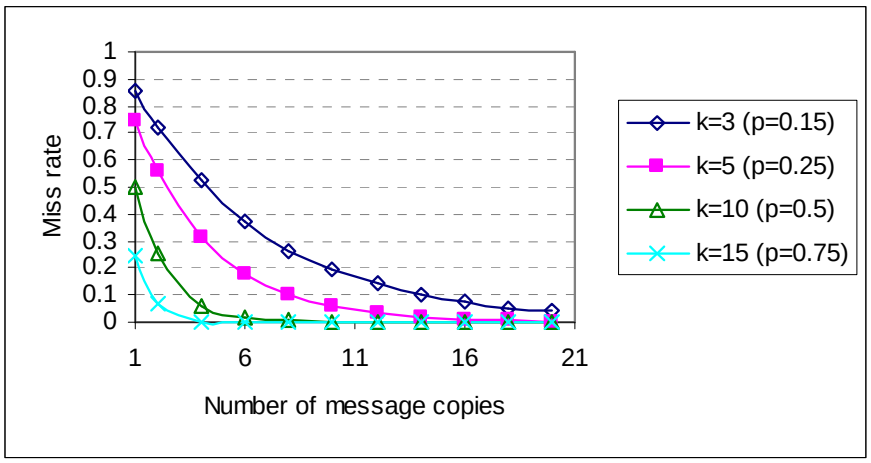


Figure 3.6: Miss rate vs. number of message copies (20 senders simultaneously).

As a first observation, simulation results show that by increasing the number of message copies the miss rate is decreased. Simulation results show that by selecting 10 queues from a total of 20 queues (P=0.5), a green node needs to send just 5 copies of a message (in attack time) to have miss rate less than 1%.

Figure 3.7 shows the miss rate for different values of message copies when the number of selected queues varies along x-axis. C is the number of message copies.

Both Figures 3.6 and 3.7 show that the miss rate decreases when the number of message copies or the number of selected queues is increased. Figure 3.7 also shows that a green node needs to send just 5 copies of a message when the receiver selects 10 queues for processing (P=0.5) to have a miss rate less than 1%.

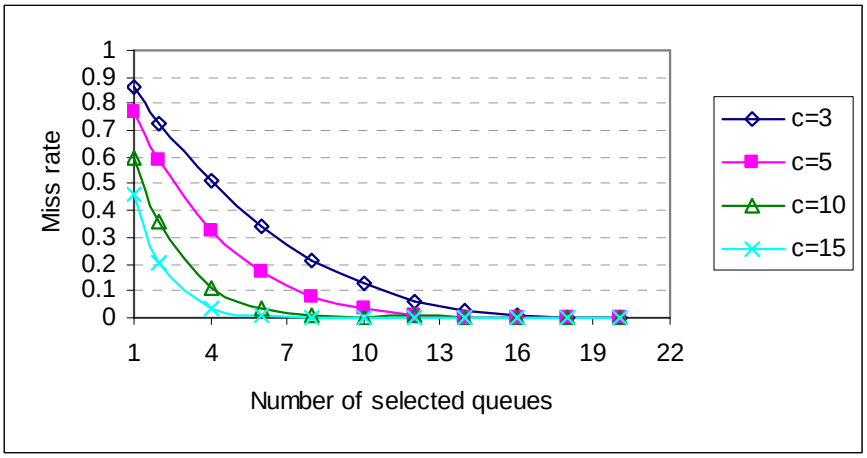


Figure 3.7: Miss rate vs. number of selected queues (20 senders simultaneously).

From these figures we can extract useful information for the system. Table 3.2.3.3 shows the number of message copies that a green node needs to send during the attack time when a receiver selects different numbers of queues for processing (miss rate less than 1%). From this table, the freedom to choose for the designer is obvious. If sending more copies of messages is cheaper than processing more selected queues, we can send more copies of messages (e.g., c=20) but instead we can select less queues for processing (e.g., k=3) and vice versa.

Number of selected queues	Number of message copies
K=3 (P=0.15)	C=20
K=5 (P=0.25)	C=15
K=10 (P=0.5)	C=5
K=15 (P=0.75)	C=3

Table 3.1: Number of selected queues vs. number of message copies (miss rate less than 1%)

Queues with Limited Length. Here we assume that queues have limited length (reality) and we show the efficacy of our approach against DoS attacks. When queues have unlimited length, the attack rate does not have effect on performance, as the message copies can be enqueued and will be processed eventually. When queues have limited length, some message copies may not be enqueued due to the large amount of attack traffic and they may not be processed. Hence, the amount of attack traffic has definitely effect on performance.

Figure 3.8 displays the miss rate for different values of selected queues when the amount of attack rate varies along the x-axis. In this experiment we keep C at 10.

The simulation results show that when the attack rate is increased, the miss rate also grows. However, by increasing the number of selected queues for processing we can reduce the miss rate. For instance by selecting 10 queues for processing, the miss rate is less than 10% when the attack rate is at the maximum amount (2000 packet/sec).

Figure 3.9 depicts the miss rate for different values of message copies when the attack rate

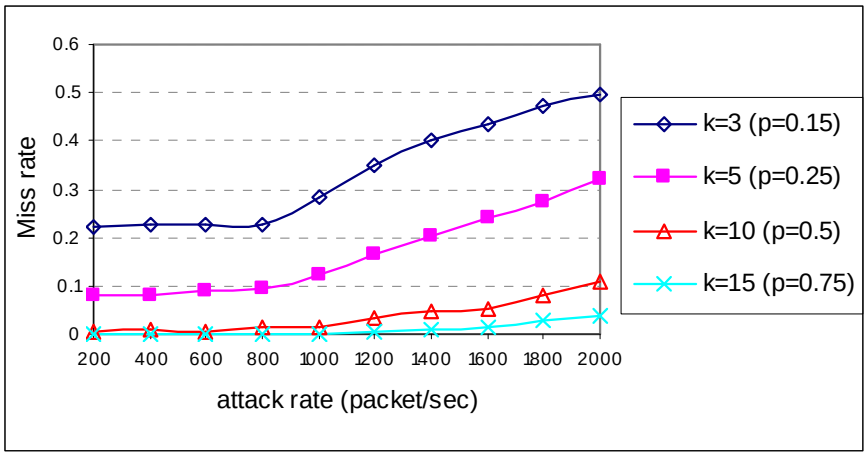


Figure 3.8: Miss rate vs. attack rate (for different values of selected queues).

varies along the x-axis. We hold k at 10 in this experiment. Results show that by increasing the number of copies of a message, we can thwart the effect of large attacks.

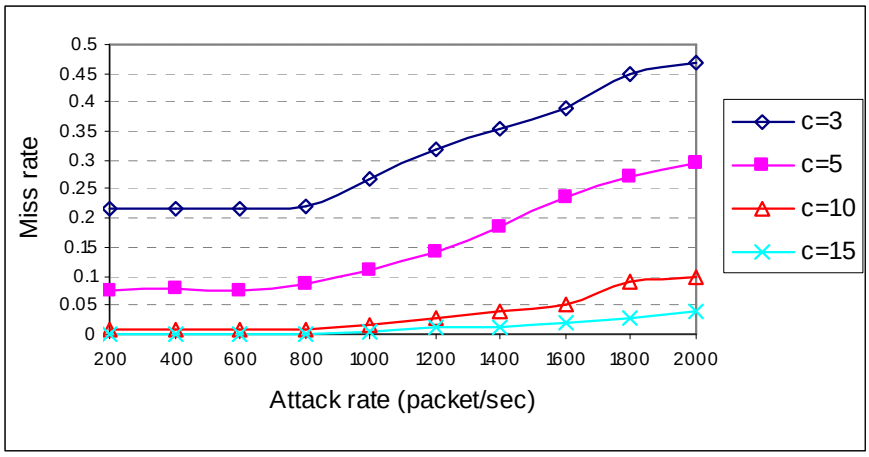


Figure 3.9: Miss rate vs. attack rate (for different values of message copies).

Another interesting view of these figures is that the miss rate does not change or increase at low rates (low dip) when the attack rate starts to increase. For instance Figure 3.9 shows that for C=10 or 15 (number of message copies) the miss rate is less than 1% when the attack rate is less than 1000 packet/sec. By reducing the number of message copies or the number of selected queues, this amount (attack rate) should be less to have no change in the miss rate. For example when C=3, the attack rate should be less than 800 packet/sec to have small change in the miss rate. As an outcome of these experiments, we can summarize that for any value of the attack rate, the best values can be found for any pair of selected queues and message copies by simulation or analytical modeling of the real system.

Messages with a Deadline. Here we compare Fosel, SOS and a straightforward filter, when a message should be processed before its deadline. We assign deadlines to the messages based on a uniform distribution between $100T \leq deadline \leq 500T$. We suppose queues have a limited length

(maximum 50 messages). Figures 3.10 and 3.11 show the deadline miss rate for Fosel, SOS and a straightforward filter when the number of selected queues and the number of message copies varies along the x-axis respectively. Here a deadline miss message corresponds to a message that has not been processed or that was processed after the deadline. Results show that Fosel is far better than SOS and a straightforward filter.

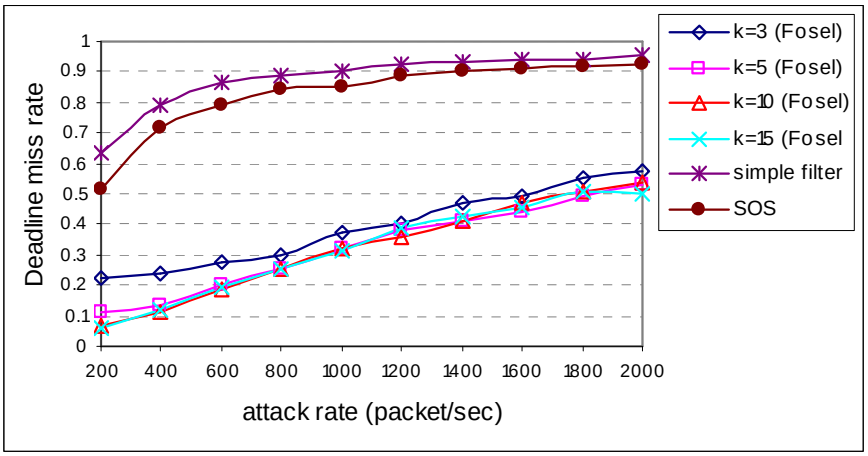


Figure 3.10: Compare Fosel, SOS and a straightforward filter (deadline miss rate vs. attack rate for different values of selected queues).

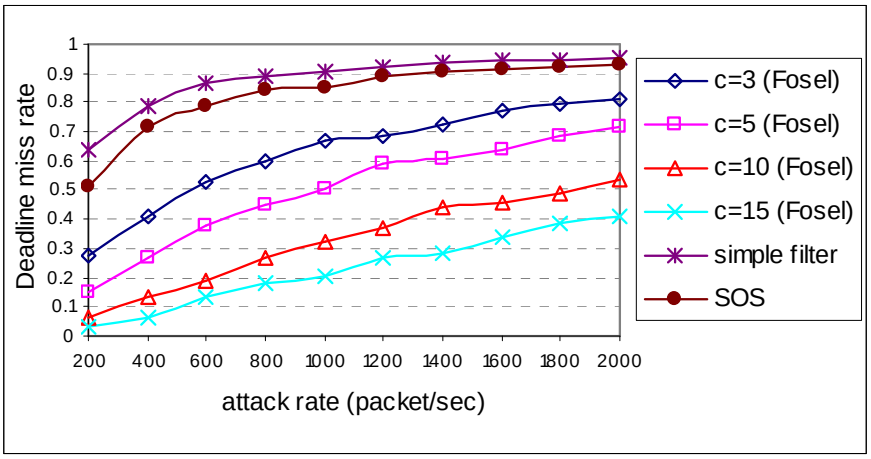
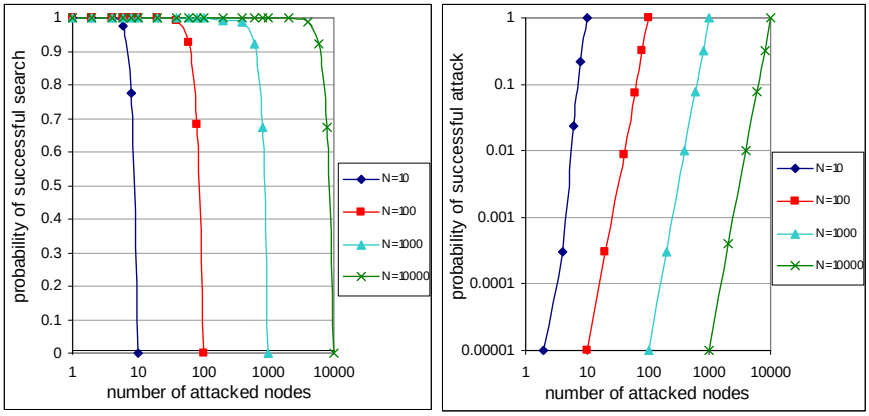


Figure 3.11: Compare Fosel, SOS and a straightforward filter (deadline miss rate vs. attack rate for different values of message copies).

Attacking the overlay network

In this set of simulations we assume that adversaries attack the overlay nodes randomly and maybe some of these attacked nodes are green nodes. Our evaluation determines the probability of successful searches and the probability of successful attacks. A search is successful if a path between any two arbitrary application sites can be found.

The first group of experiments analyzes the likelihood of successful searches in the static case (an attacker would select a fixed set of nodes to attack and Fosel takes no action towards repairing the attack). Figure 3.12 (a and b) shows the probability of successful searches and the probability of a successful attack when the number of attacked nodes varies along the x-axis, respectively.



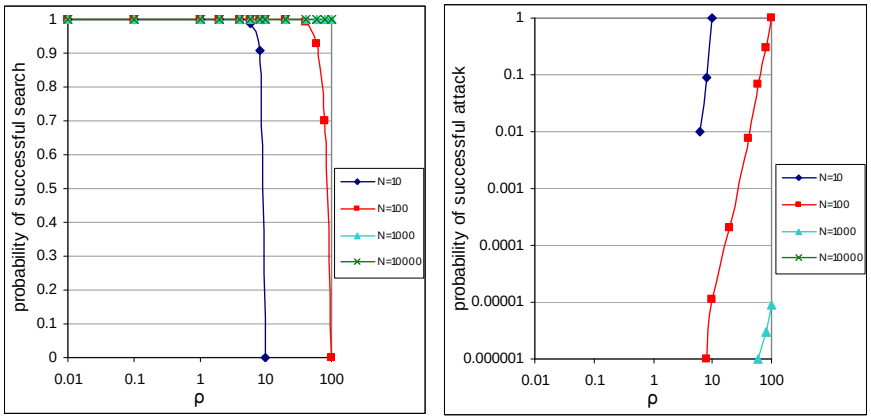
a) Probability of successful search b) Probability of successful attack

Figure 3.12: DoS attack in the static case.

From this figure, we see that the likelihood of an attack successfully terminating the 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 99.999% successful searches. If attackers bring down between 20% and 40% of overlay nodes, more than 98% successful searches exist. Even if attackers bring down half of total nodes (50%), there are still 90% successful searches.

The second set of experiments in this section evaluates the Fosel architecture in a dynamic case. Previous experiments assumed that an attacker would select a fixed set of nodes to attack, and that Fosel takes no action towards repairing the attack. The scenario of this set of experiments is that when Fosel identifies an attacked node, that node is removed from the overlay. When an adversary identifies that a node it is attacking no longer resides in the overlay, it redirects the 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 and 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 Fosel architecture in the dynamic case for both DoS and DDoS attacks.

Figure 3.13 (a and b) plots the probability of a successful search and a successful attack respectively for a 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 service. Also for large value of N (e.g.

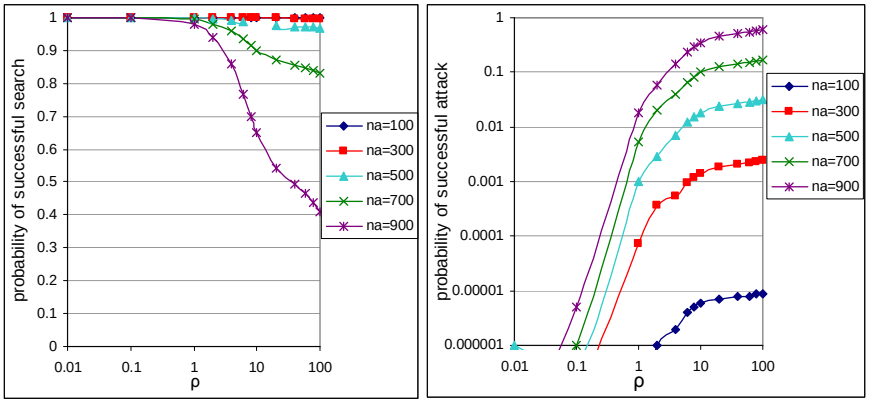


a) Probability of successful search b) Probability of successful attack

Figure 3.13: DoS attack in the dynamic case.

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 searches. We think in practice that ρ is always less than 10. As a result, a DoS attack in the Fosel architecture with repair is solved nearly completely.

Figure 3.14 (a and b) plots the probability of successful search and successful attack respectively for a distributed attack (DoS attack on several overlay nodes simultaneously) in the dynamic case. In this figure n_a is the maximum number of overlay nodes that can be attacked simultaneously and the simulation is done for N = 1000.



a) Probability of successful search b) Probability of successful attack

Figure 3.14: DDoS attack in the dynamic case.

This figure shows that when an attack is distributed, the fraction of time for which the attack is successful can be significant when a large fraction of nodes in the overlay is attacked. From this figure we can understand although a distributed DoS attack is harder to tolerate than a centralized DoS attack, for $n_a \leq N/2$ it can be tolerated. For instance, when $n_a \leq N/2$ for any value of ρ the probability of successful search is more than 90%. An interestingly note of Figure 3.14 is that when ρ is increased above 10, the curves reach a steady state and remain constant. It means that if ρ is increased to even more than 100, the probability of successful attack will not change or change unnoticeably.

3.3 Activity Support

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

3.3.1 CIS Protection Service

The CRUTIAL reference architecture to protect critical infrastructures [45, 114] 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 the definition of 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 creating 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 the *Protection Service (PS)* of the CIS device. PS ensures that the incoming and outgoing traffic in/out of the LAN satisfies the security policy of the infrastructure. A CIS, however, 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 [54, 65]. Instead, a CIS is a distributed protection device based on a sophisticated access control model and developed with intrusion-tolerant capabilities. In this section we focus on the design of the intrusion-tolerant protection service of the CIS, starting with an overview of the architecture and then we define the algorithms it executes. In the rest of this section, "the CIS" means "the CIS *Protection Service*".

3.3.1.1 Overview 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, which do not even have know that 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 [17], and is specially useful for CII 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) [24].

- *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 [44]. 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 [49]. 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, based on OrBAC (Organization Based Access Control) [1].

In addition to these characteristics, and as explained in Section 2.2, several designs of the CIS can ensure high levels of resilience through intrusion tolerance, which is achieved with the (physical or virtual) replication of its components. Over the years, several intrusion-tolerant services have been proposed in the literature (e.g., storage [28, 51, 77], certification authorities [95, 127], and DNS [27]), either based on Byzantine quorum systems (BQS) [75, 127] or state machine replication (SMR) [28, 98]. 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 a method has to be devised to protect the recipients from messages not satisfying the security policy that are forwarded by faulty replicas.

Figure 3.15 represents an 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. As described in Section 2.2, this MAC allows internal applications to detect and discard malicious messages produced by faulty replicas. 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.

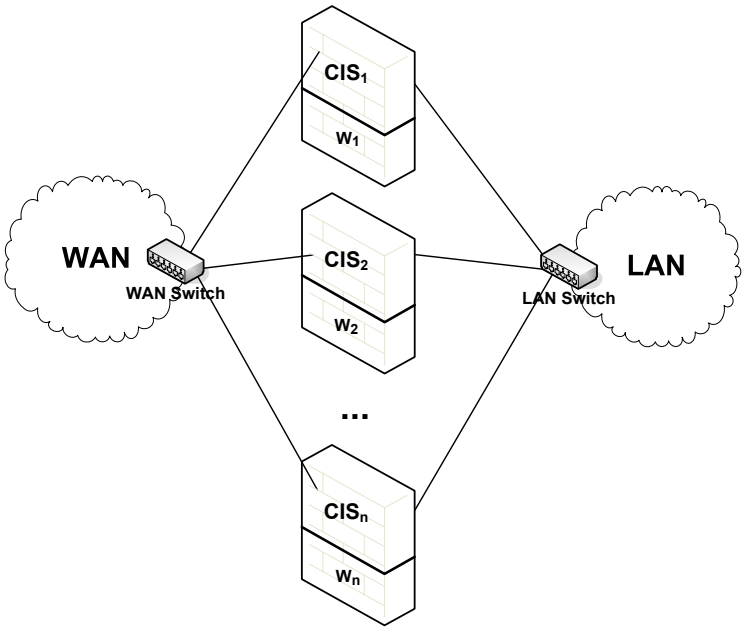


Figure 3.15: An intrusion-tolerant CIS architecture.

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 [61] and IEC 61850 [62], are designed on top of TCP, which ensures reliable communication even if the network (in this case the CIS) loses messages.

3.3.1.2 Model of the System

The system is composed by n CIS replicas $CIS = \{CIS_1, ..., CIS_n\}$. 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 [111]: 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 [87]. Finally, we assume also that the station computers can not be compromised[2].

Wormhole. *Asynchronous secure tamper proof subsystem* $W = \{W_1, ..., W_n\}$ 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 [5, 73] 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-multicast*(G,m), to multicast a message m to the group G, and *U-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 3.15).

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 [70], 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., $m_\sigma = m.MAC(m,K)$.

3.3.1.3 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:

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

- 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.

3.3.1.4 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 *PolEng_verify* function. 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 3.2.

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 *true* if σ is a MAC of m produced with K_{LAN}, and *false* otherwise

Table 3.2: Wormhole services specification.

The Algorithm. The CIS replicas execute Algorithm 8 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

Algorithm 8 CIS payload (replica CIS_i).

```
{Parameters}
integer T_vote {Expected time to vote a message}

{Variables}
set Voting = ∅ {Messages being voted}
set Pending = ∅ {Not yet forwarded messages}
set TooEarly = ∅ {Messages forwarded before their arrival}

{Code for WAN message reception and processing}
upon U-receive(WAN, m)
 1: if m_σ ∈ TooEarly then
 2:     TooEarly ← TooEarly \ {m_σ}
 3: else
 4:     if PolEng_verify(m) then
 5:         Voting ← Voting ∪ {m}
 6:         m_σ ← approve(m)
 7:         Voting ← Voting \ {m}
 8:         Pending ← Pending ∪ {m_σ}
 9:         waitRandom()
10:         if m_σ ∈ Pending then
11:             U-multicast(LAN, m_σ)
12:         end if
13:     end if
14: end if

{Code for LAN message reception and processing}
upon U-receive(LAN, m_σ)
15: if m_σ ∈ Pending then
16:     Pending ← Pending \ {m_σ}
17: else if W_verify(m_σ) then
18:     TooEarly ← TooEarly ∪ {m_σ}
19: end if

function approve(m)
20: vote_i ← W_create_vote(m)
21: ∀CIS_j ∈ CIS, send(j, ⟨VOTE, H(m), vote_i⟩)
22: C_m ← ∅
23: repeat
24:     wait until receive(j, ⟨VOTE, H(m), vote_j⟩)
25:     C_m ← C_m ∪ {vote_j}
26:     σ ← W_sign(m, C_m)
27: until σ ≠ ⊥
28: return m_σ

{Periodic task for message retransmission}
for each T_vote that Voting ≠ ∅
29: ∀m ∈ Voting : U-multicast(WAN, m)
```

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.

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)[3]. 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 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

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

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).

3.3.1.5 Wormhole Protocol

The implementation of the three services provided by the wormhole is presented in Algorithm 9. 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 9 Wormhole services (local wormhole W_i).

{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** $\mathsf{MAC}(\langle i,m\rangle, K_W)$

service $W\_sign(m, C_m)$
2: **if** $|\{v \in C_m : \exists j \text{ s.t. } v = \mathsf{MAC}(\langle j,m\rangle, K_W)\}| \geq f+1$ **then**
3: **return** $\mathsf{MAC}(m, K_{LAN})$
4: **else**
5: **return** $\perp$
6: **end if**

service $W\_verify(m_\sigma)$
7: **if** $\mathsf{MAC}(m, K_{LAN}) = \sigma$ **then return** *true* **else return** *false*

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).

3.3.1.6 Taking Advantage of Distributed Wormholes

In Section 4.1 of this report it is described a proactive-reactive recovery mechanism to be integrated on the CIS in order to make it self-healing, i.e., CIS replicas are recovered to a clean and safe state both periodically and reactively when problems are detected.

To use this solution, the system model must include not a local asynchronous wormhole (like the one described in Section 3.3.1.2), but a distributed and timely one as described in Section 4.1.2. This solution, depicted in Figure 3.16, is what we call *Self-healing CIS* (SH-CIS).

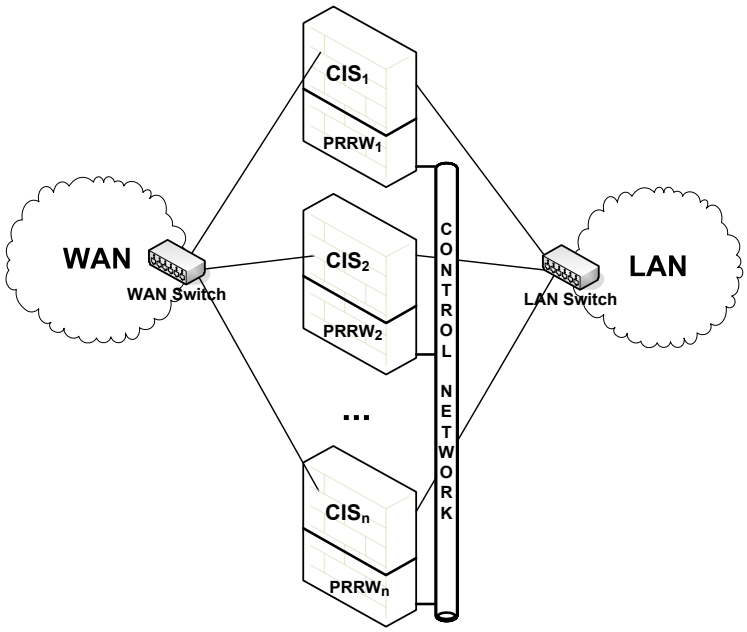


Figure 3.16: Self-healing CIS architecture.

Comparing this figure with 3.15 it can be seen that here we have a separate network connecting all local wormholes. We can take advantage of this feature to improve the CIS design through two modifications:

- *DoS-resilient message voting*: In the design presented in the previous section, 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, since it can degrade substantially the message processing protocol[4] (Algorithm 8). Considering the CIS architecture augmented with the distributed secure wormhole, in

[4]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., [28, 126, 127]).

which all replicas' local wormholes are connected by a secure network, the whole voting protocol can be securely executed by the wormholes (in a crash-failure model), preventing malicious replicas from disturbing it.

- *Leader Election*: To avoid making all n CIS replicas send the same approved message to the station computer on the LAN, the algorithm of previous section implements a random waiting for each replica in such a way that a replica would send an approved message to its destination if and only if this message was not received in the LAN (which only happens if some other replica already sent it - see Section 3.3.1.4). This would not be need if we have one distinguished CIS replica, that is responsible for sending the message to the station computer. The problem in having this leader replica is that it requires a *perfect failure detection mechanism* (to detect failures in the leader) and a *leader election protocol* (to define a new leader in case of faults). It is trivial to implement these components inside the timed distributed wormhole:

  - *Perfect failure detection*: Since both the processing and communication delays have defined bounds (δ and Δ, respectively), it is trivial to implement a perfect failure detector using timeouts [29]: each process wormhole w_i send a message to another process wormhole w_j and wait $2\Delta + \delta$ for a reply. If it is received, w_j is accounted as correct, otherwise it is considered as faulty.
  - *Leader election*: due to the existence of time bounds in the wormhole, any classical leader election protocol (e.g., [53]) can be directly implemented.

  Even with these services inside the wormhole it still remains the problem of a compromised leader replica that omits to send some approved messages or that send invalid messages. In this case, the leader will be suspected or detected faulty through the use of some services offered by the proactive reactive recovery mechanism, as described in Section 4.1.5.

3.3.1.7 Supporting Statefull Firewalls

The CIS design presented in previous 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 messages 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 the same sequence of messages are delivered and evaluated in all CIS replicas (see Section 3.2.1 for the definition and an implementation of an Atomic Broadcast primitive, which ensures this kind of guarantee). This protocol could require f more replicas and either a random oracle (see Section-3.2.1), some timing assumptions (e.g., [28]) or a distributed secure and timely wormhole [31].

3.3.2 Access Control and Authorization

As indicated in Section 2.1, the CII architecture can be seen as a WAN of LANs interconnected by dedicated devices, called CIS. Each LAN is composed of logical/physical systems, has its own applications and access control policy, and proposes its services to other systems. Each LAN belongs to a facility (e.g., power plant, substation, control center, etc.), and the WAN interconnects all the facilities belonging to the CII. The CII is managed and accessed by different actors and stakeholders (power generation, transmission and distribution companies, regulation authorities, communication and computing system providers, brokers, subcontractors, etc.).

In this section, we mainly focus on security challenges related to access control, collaboration, and interoperability between the organizations and the systems composing the CII, and we show that this problem can be solved by using PolyOrBAC [4, 12, 3], the security framework we developed for CRUTIAL. This framework is based on the OrBAC access control model and on web services (WS).

3.3.2.1 CII Security Requirements

Globally, a CII can be seen as a set of interconnected organizations involving different actors and stakeholders (e.g., power generation companies, substations, energy authorities, maintenance service providers, transmission and distribution system operators, etc.) using heterogeneous logical and physical information and communication systems and networks, exhibiting different levels of security threats and protection mechanisms. In order to provide a satisfactory level of protection for the global interconnected critical infrastructure, the following security constraints and requirements need to be carefully addressed.

- Secure cooperation between deferent organizations, possibly mutually suspicious, with different features, functioning rules and policies.
- Autonomy and loose coupling: each organization controls its own security policy, users, resources, applications, etc., while respecting the global functioning of the whole system.

To satisfy the CII security requirements cited above, two approaches are possible. The first solution consists in considering a centralized approach with a global CII organization. In terms of access control, this solution implies defining a global security policy for the CII, and all participating organizations must adapt their own security policies to conform with this global policy. This is generally unacceptable for large companies that may belong to several independent CIIs, which could potentially define conflicting security policies. A better solution has been proposed in some previous works such as MultiOr-BAC [2] and O2O [33], where the various organizations accept to cooperate so that roles in one organization are given privileges in another organization. For that, each participating organization must trust the others, at least for the definition of their roles and for the assignment of the corresponding roles to trustworthy users. This approach is also

intrusive with respect to the confidentiality of each organizations internal structure, user identity, and security policy. This is equally unacceptable for CIIs where participating companies are in competition, and thus are mutually suspicious. Clearly, each organization wants to keep its autonomy on the choice of its internal security policy, and would not accept to open its information and communication system to unknown external users working for its competitors. Ideally, an organization should know nothing about the other organizations users or assets, but only the information needed to cooperate fairly. Enabling a secure collaboration between organizations while preserving each organizations autonomy and self-determination is the challenge addressed by the second approach, based on decentralized security policies. Basically, the PolyOrBAC framework [12] adopts this second approach. Actually, PolyOrBAC uses (1) OrBAC (Organization-based Access Control) model [1] for specifying local security policies and (2) web services to manage the collaboration aspects. The following subsections describe our proposal.

3.3.2.2 Specifying Local Security Policies in OrBAC

In the context of CIIs, each organization specifies its own security policy, which defines who has access to what, when, and in which conditions. In this subsection, we show that OrBAC is the most suitable access control model for achieving this task. Actually, the OrBAC model is an extension of the traditional RBAC (Role-Based Access Control) model [97, 48]. In RBAC, roles are assigned to users, permissions are assigned to roles and users acquire permissions by playing roles. By abstracting users into roles, RBAC facilitates the security policy management. Indeed, if users are added to or withdrawn from the system, only instances of the relationship between users and roles need to be updated.

OrBAC goes further by abstracting objects into views and actions into activities. In this way, security rules are specified by abstract entities only. Consequently, the representation of the security policy is completely separated from the implementation. More precisely, in OrBAC, an activity is a group of one or more actions; a view is a group of one or more objects; and a context is a specific situation that conditions the validity of a rule. Actually, two security levels can be distinguished in OrBAC:

- *Abstract level*: the security administrator defines security rules through abstract entities (roles, activities, views) without worrying about how each organization implements these entities.
- *Concrete level*: when a user requests to perform an action on an object, permissions are granted to him according to the concerned rules, the organization, the role currently played by the user, the requested action (that instantiates an activity defined in the rule) on the object (that instantiates a view defined in the rule), and the current context. The derivation of permissions (i.e., runtime evaluation of security rules) can be formally expressed as follows:

$\forall$ org $\in$ Organizations, $\forall$s $\in$ Subjects, $\forall$ $\alpha \in$ Actions, $\forall$ o $\in$ Objects, $\forall$ r $\in$ Roles, $\forall$a $\in$ Activities, $\forall$ v $\in$ Views, $\forall$ c $\in$ Contexts
Permission **(org, r, v, a, c)** $\wedge$ *Empower* (org, s, r) $\wedge$ *Consider* (org, α, a) $\wedge$ *Use* (org, o, v) $\wedge$ *Hold* (org, s, a, o, c) $\rightarrow$ ***Is permitted*****(s, α, o)**

This rule means: if in a certain organization, a security rule specifies that role r can carry out the activity a on the view v when the context c is true, and if r is assigned to subject s, if action α is a part of a, and if object o is part of v, and if c is true, then s is allowed to perform α (e.g., WRITE) on o (e.g., f1.txt). Prohibitions, and obligations can be defined in the same way.

As rules are expressed only through abstract entities, OrBAC is able to specify the security policies of several collaborating and heterogeneous sub-organizations (e.g., departments) of a global organization. In fact, the same role, e.g., operator can be played by several users belonging to different sub-organizations; the same view e.g., TechnicalFile can designate a table TF-Table in one sub-organization or a XML object TF1.xml in another one; and the same activity read can correspond in a particular sub-organization to a SELECT action while in another sub-organization it may specify an OpenXMLfile() action.

In our context, OrBAC presents several benefits and satisfies several security requirements of organizations participating in a CII: rules expressiveness, abstraction of the security policy, scalability, heterogeneity and evolvability. OrBAC is thus more suitable than RBAC (and its variants); in particular, for specifying local security policies of the CIIs organizations. These security policies can subsequently be locally enforced by (local) security mechanisms such as Access Control Lists (ACL), firewall rules, security credential (e.g., XML capabilities), OASIS WS security mechanisms, etc.

3.3.2.3 Managing Collaboration into PolyOrBAC

If OrBAC is suitable for specifying local security policies, it has an important limitation in our context. Actually, OrBAC is centralized and does not handle collaborations between non-hierarchical organizations, i.e., secure handling of accesses to external resources. In fact, OrBAC does not allow specifying rules that involve several autonomous organizations. Moreover, it is impossible to associate permissions to users belonging to other organizations. As a result, OrBAC is unfortunately only adapted to centralized infrastructures and does not cover the distribution and collaboration needs of current CIIs. To fulfill this requirement, we have proposed the MultiOrBAC model in [2]. Basically, MultiOrBAC abstract rules specify that roles in a certain organization are permitted (or prohibited or obliged) to carry out activities on views belonging to other organizations. Therefore, contrarily to OrBAC, a MultiOrBAC rule may involve two different organizations that do not belong to the same hierarchy: the organization where the role is played, and the organization to which belong the view and the activity. However, in the context of CIIs, MultiOrBAC presents several weaknesses. In fact, MultiOrBAC offers the possibility to define local rules/accesses for external roles (i.e., belonging to another organization), without having any information about who plays these roles and how the (user, role) association is managed in the remote organization. This causes a serious problem of responsibility and liability: who is

responsible in case of remote abuse of privileges? How can the organization to which belongs the object have total confidence in the organization to which belongs the user?

The MultiOrBAC logic is thus not adapted to CIIs where in-competition organizations can naturally be mutually suspicious. Moreover, in MultiOrBAC the access control decision and enforcement are done by each organization, which means that the global security policy is in fact defined by the set of the organizations security policies. In that case, it is difficult to enforce and maintain the consistency of the global security policy, in particular if each organizations security policy evolves independently. In our PolyOrBAC framework, collaboration and interactions between organizations are made through the use of the WS technology, which provides platform-independent protocols and standards for exchanging heterogeneous interoperable data services. Software applications written in various programming languages and running on various platforms can use WS to exchange data over computer networks in a manner similar to inter-process communication on a single computer. WS also provide a common infrastructure and services for data access, integration, provisioning, cataloging and security. These functionalities are made possible through the use of open standards, such as: XML for exchanging heterogeneous data in a common information format [118]; SOAP, a data transport mechanism to send data between applications in one or several operating systems [117]; WSDL, used to describe the services that a business offers and to provide a way for individuals and other businesses to access those services [119] and UDDI, an XML-based registry which enables businesses to list themselves and their services on the Internet and discover each other [85]. The question that arises is how to integrate WS and OrBAC. To answer this question, we introduce two new notions, the virtual users and the WS images:

- for the organization offering a WS (i.e., that allows external accesses to its local resources through WS interfaces), the other organization is seen as a virtual user which plays a role authorized to use the WS,
- for the organization requesting the WS, the WS is seen as an external object, locally represented by its image.

To illustrate these notions, we explain the two main phases of PolyOrBAC: publication and negotiation of collaboration rules as well as the corresponding access control rules and runtime access to remote services. In the first phase, each organization determines which resources it will offer to external partners. Web services are then developed on application servers, and referenced on the Web Interface to be accessible to external users. When an organization publishes its WS at the UDDI registry, the other organizations can contact it to express their wish to use the WS. Let us take a simple example where organization B offers WS1, and organization A is interested in using WS1. A and B should negotiate and come to an agreement concerning the use of WS1. Then, A and B establish a contract3 and jointly define security rules concerning the access to WS1. These rules are registered (according to an OrBAC format) in databases located at both A and B. For instance, if the agreement between A and B is users from A have the permission to consult Bs measurements in the emergency context, B should, in its OrBAC security policy: have (or create) a rule that grants the permission to a certain (local) role (e.g., Operator) to consult its measurements: Permission(B, Operator, Measurements, Consult, Emergency); create a virtual

user noted PartnerA that represents A for its use of WS1; add the Empower (B, PartnerA, Operator) association to its rule base. This rule grants PartnerA the right to play the Operator role. In parallel, A creates locally a WS1 image which (locally in A) represents WS1 (i.e., the WS proposed by B), and adds a rule in its OrBAC base to define which of As roles can invoke WS1 image to use WS1.

Considering the second phase of PolyOrBAC dedicated to the control of runtime access to remote services, we use an AAA (Authentication, Authorization and Accounting) architecture, which separates authentication from authorization; we distinguish access control decision from access control enforcement; and we keep access logs in each organization. Basically, if a user from A (let us note it Alice) wants to carry out an activity, she is first authenticated by A. Then, protection mechanisms of A check if the OrBAC security policy (of A) allows this activity. We suppose that this activity contains local as well as external accesses (e.g., invocation of Bs WS1). Local accesses should be controlled according to As policy, while the WS1 invocation is both controlled by As policy (Alice must play a role that is permitted to invoke WS1 image), and by Bs policy (the invocation is transmitted to virtual user PartnerA, which must play a role authorized to execute the web service), according to the contract established between A and B. If both policies grant the invocation, WS1 is executed (under the access control enforcement mechanisms implemented by A and by B).

3.3.2.4 Checking WS Interactions in PolyOrBAC

In the last subsection, we have shown that PolyOrBAC offers several useful concepts and mechanisms for access control in CIIs: it permits a better specification and control of local security policies through OrBAC; each organization authenticates its users and manages its resources autonomously; and interactions are handled by WS. Consequently, the service-requesting organization is liable for its users while the service-providing organization is liable for its services. However, some aspects need to be addressed:

- Enforcement and real time checking of contracts established between different organizations; in fact, the system must be able to check the satisfaction as well as the correct enforcement of the signed contracts.
- Audit logging and assessment of the different actions; in particular, every deviation from the signed contracts should trigger an alarm and notify the concerned parties.
- Handling of mutual suspicion between organizations; no information is disclosed about local security policies; the organization providing the web service does not know which user of the other organization requests the web service, and the organization requesting the web service does not know which role performs which activity in the service providing organization. But it is also necessary to detect any abuse of the contract by a malicious organization.

To deal with these issues, we state that for each WS use, an e-contract should be negotiated between the partner organizations (WS provider and WS client). This contract must specify

precisely the web service functions and parameters (including the expected quality of service, the liability of each party, payment, penalties in case of abuse, etc.), but also the security rules related to the invocation and the execution of the web service. And these security rules must be checked and enforced at runtime to prevent, or at least detect any abuse.

The question that arises now is how to specify e-contracts. Actually, the most relevant security requirements for contracts are workflows, actions, permissions, prohibitions, obligations, time constraints, disputes and sanctions. To express these requirements, we propose using timed automata [7]. First, permissions (actions that are authorized by the contract clauses) are simply specified through transitions in our timed automata. For instance in Figure 3.17, the system can (i.e., has the permission to) execute the action a at any time and then, behaves like the automaton A.

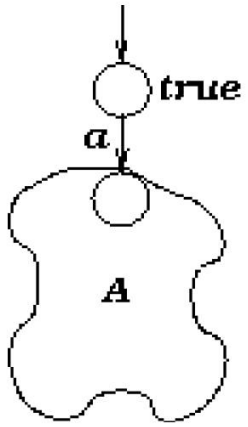


Figure 3.17: Modeling Permissions.

Second, we distinguish two kinds of prohibitions in e-contracts:

- Implicit prohibitions: the idea is to only specify permissions in the automata. The states, actions and transitions not represented in the automata are by essence prohibited because the runtime model checker will not recognize them.
- Explicit prohibitions: explicit prohibitions can be particularly useful in the management of decentralized policies / contracts where each administrator does not have details about the other organizations participating in the CII. Moreover, explicit prohibitions can also specify exceptions or limit the propagation of permissions in case of hierarchies.

In our model, we specify explicit prohibitions by adding a failure state where the system will be automatically led if a malicious action is detected. In Figure 3.18, as the a action is forbidden, its execution automatically leads to the failure state described by an unhappy face.

Let us now deal with obligations. Actually, obligations are actions that must be carried out; otherwise the concerned entity will be subject to sanctions. Besides that, as every obligation

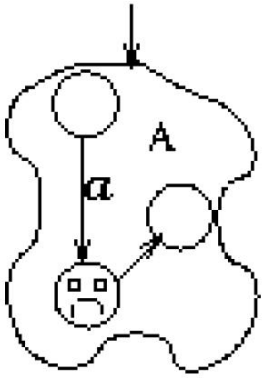


Figure 3.18: Modeling Prohibitions.

is also a permission, obligations will be specified by particular transitions (in the same way as permissions). However, as obligations are stronger than permissions, we should add another symbols to capture this semantics and to distinguish between what is mandatory and what is permitted but not mandatory. Actually, to model obligations, we use transition time-outs and invariants. In this respect, an obligation is considered as a simple transition, and if a maximum delay is assigned to the obligation, a time-out (noted by d in Figure 3.19) is set for the delay. When the obligation is fulfilled, this event resets the time-out and the system behaves like A1. On the contrary, if the timeout expires, an exception is raised and the system behaves like A2 (which can be considered as an exception).

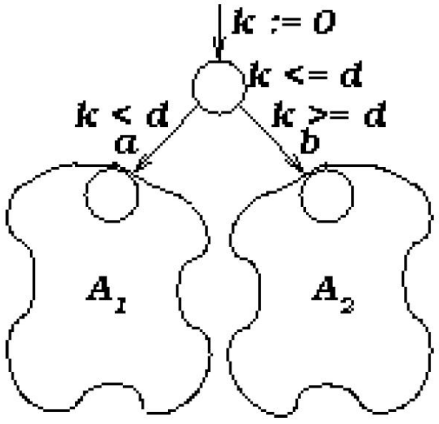


Figure 3.19: Modeling Obligations.

Basically, when an explicit prohibition is carried out or when an obligation is not fulfilled, a conflicting situation (e.g., one of the parties does not comply with the contract clauses) arises, and the automaton automatically makes a transition to a dispute situation (i.e., to the unhappy state) or triggers an exception processing (A2 in Figure 3.20). Actually, modeling disputes will allow to not only identify anomalies and violations, but go further by identifying activities (succession of actions, interactions) that led to these situations, and finally can automatically lead to the cancelation of the contract. Moreover, as disputes have different severities and as they are not all subject to the same sanctions, we use variables (i.e., labels on the unhappy state) to distinguish the different kinds of disputes as well as the corresponding sanctions (in Figure 3.20). Note that

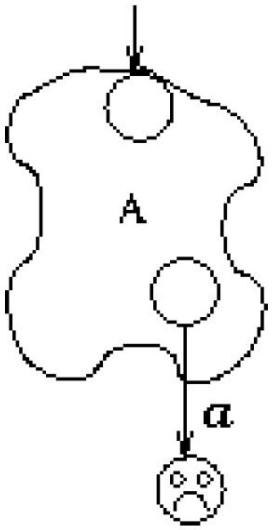


Figure 3.20: Modeling Conflicts.

once the expected behaviors of the contracting parties are modeled by timed automata, we can verify some security properties and enforce them at run-time by model checking [3]. In particular, we can (1) verify statically if the system can reach a dispute state, (2) maintain an audit log and perform model-checking during runtime, and (3) notify the concerned parties in case of contract violation. All these issues will be detailed on an example in the next section.

Considering the CRUTIAL architecture, since all organizations of the CII are interconnected by CISs, and in order to provide a controlled cooperation adapted to the CII, each CIS must contain mechanisms to enforce the local security policy of its organization with respect to external accesses. These policies and mechanisms must allow the authorized accesses to the resources and prevent the unauthorized accesses (accidental or malicious ones). In the next subsection we describe one of the scenarios we considered in the CRUTIAL project. Then, we specify its WS and we identify the virtual users. Finally, we present the e-contracts enforcement and runtime checking.

3.3.2.5 Electrical Scenario Interpretation with PolyOrBAC

This section describes the application of PolyOrBAC, our collaborative access control framework based on OrBAC access control model and on the Web services technology, to the critical information infrastructure in the particular case of an electrical power grid. First of all, we will briefly recall the architecture of an electric power grid, and present a load shedding scenario in emergency conditions. This scenario is similar to scenario 2 described in [45], but modified to increase the role of human operators, in order to highlight the risks of malicious actions made by an attacker impersonating a human operator, or by a malicious partner participating in the CI, which justifies the enforcement of access control. In an electric power grid, one or more electricity generation companies (each in charge of one or several power plants) is connected to one or more transmission grids. Each transmission grid (managed by Transmission System Operators, TSO) is

composed of transmission substations (monitored by one National Control Center NCC per country and several Regional Control Centers RCC), and is connected to one or more distribution grids. Finally each distribution grid (managed by Distribution System Operators, DSO) is composed of distribution substations (monitored by Area Control Centers ACC), and distributes electricity to subscribers (industrial, commercial and residential users) over distribution lines [45].

The following scenario illustrates the application of PolyOrBAC on the electrical power grid. This scenario considers the possible cascading effects due to ICT threats to the communication channel among TSO/DSO Control centers and their substations in emergency conditions (e.g., line overloads). It is assumed that in emergency conditions, the TSO is authorized by the DSOs to activate defense plan actions for performing load shedding activities on the Distribution Grid.

In studying this scenario, we distinguish four important classes of organizations: Transmission System Control Centers (TS CC) that are managed by TSOs, Transmission System Substations (TS SS), Distribution System Control Centers (DS CC) that are managed by DSOs, and Distribution System Substations (DS SS). Figure 3.21 details the most important commands and signals exchanged between these organizations in normal and emergency situations.

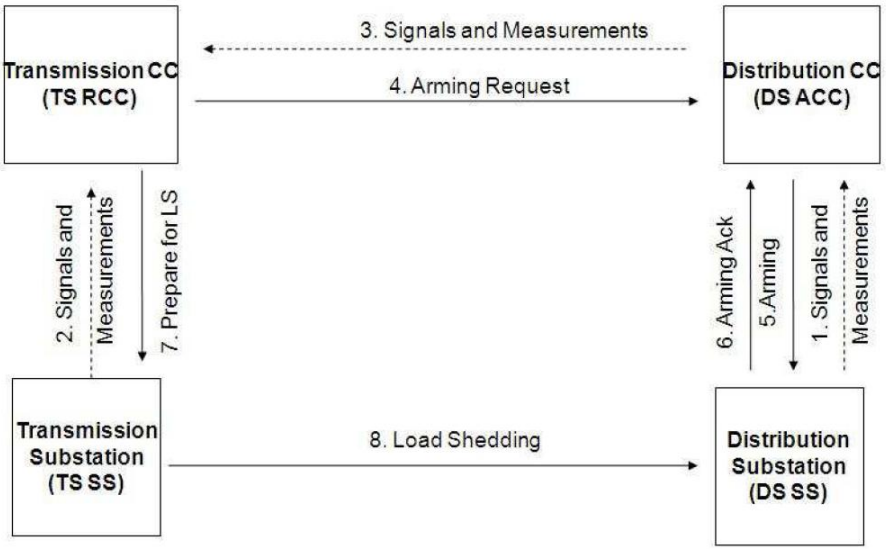


Figure 3.21: Exchanges Signals in Load Shedding Scenario.

In normal operation, all DS SSs send various signals and measurements (power, voltage, frequency) to the TS CC via their DS CC (1) and (3). On the other hand, all TS SSs send various signals and measurements to their TS CC (2). The TS CC monitors the Electric Power System and, if a critical situation is detected, elaborates some potentially emergency conditions that could be remedied with opportune load shedding commands applied to particular areas of the distribution grid. In order to actuate the defense action, the TSO asks a DSO to prepare for a possible load shedding up to a certain power (4). The DSO selects which distribution substations (DS SSs) can shed the needed load with the less severe disturbance for the users and arms these substations for a possible upcoming load shedding (5). When a DS SS receives an arming order from a DSO, it arms the corresponding electrical component (Monitoring Control and Defence Terminal Unit

or MCDTU) and sends an acknowledgement to the DSO (6). When the DS CC has received all the acknowledgements from the DS SS, it sends an acknowledgement to the TS CC. In parallel, the TS CC sends an order to the TS SS controlling the distribution area to prepare itself for a possible load shedding in a near future (7). In case of detection of a real emergency situation, the TS SS sends a load shedding command to all DS SSs participating to the emergency plan, and only the previously armed DS SSs will perform load shedding over their MCDTUs (8). During all the duration of the critical situation, to reduce user disturbance, the DSO can choose to disarm certain DS SSs and arm others. When the critical situation is solved (i.e., no more load shedding is expected at short term), the DSO reintegrates the disconnected DS SS. In our scenario, we distinguish 4 organizations (TS CC, DS CC, TS SS, DS SS), and 3 web services in Figure 3.22.

Service	Provider	Client
WS1-arming-request	DS CC	TSO
WS2-arming-order	DS SS	DSO
WS3-prepare-for-LS	a Sentinel process in the TS SS	TSO
WS4-Load-Shedding	DS SS	a Sentinel process in the TS SS
WS5-disarming/reintegration	DS SS	DSO

Figure 3.22: Created Web Services in Load Shedding Scenario.

Figure 3.23 summarizes the different web services, virtual users (representing remote organizations that can request web services), and ws-images (local images of remote web services that can be invoked), and resources involved in the execution of this scenario.

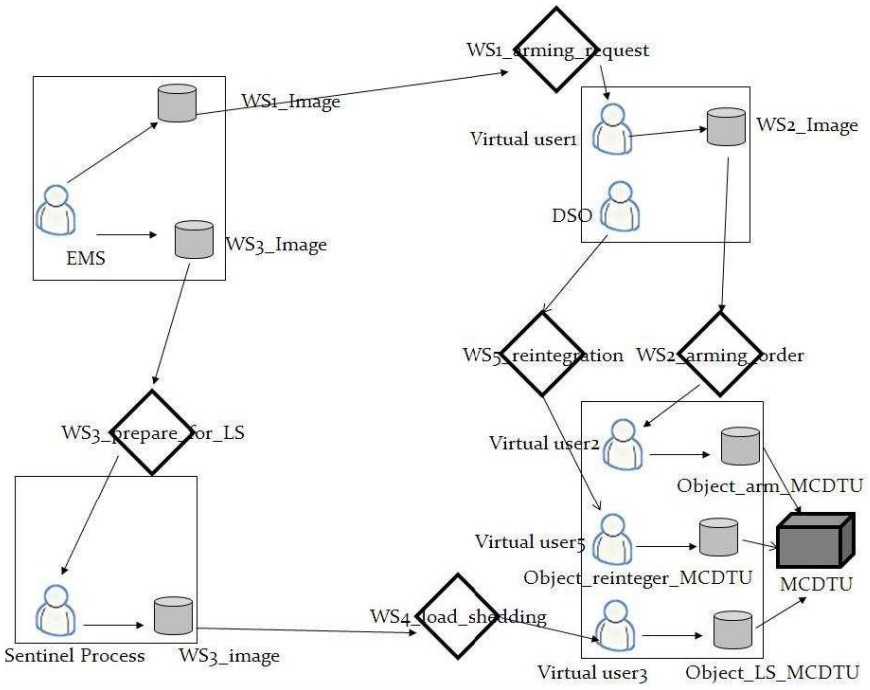


Figure 3.23: Using PolyOrBAC in Load Shedding Scenario.

We assume that the EMS requests the DS CC to arm its DS SS MCDTUs. When the EMS activates WS1-Image, the execution of WS1-arming-request is automatically activated. This access (EMS to WS1-image) is checked according to TS CC policy and is granted according to the

OrBAC rule described in Figure 3.24, that manages the Access Control for the Arming Request Web service at the level of the organization that invokes the service (i.e. TS CC).

Rules
Permission(TS CC, TSO, DS CC arming request, send, critical situation) ∧ Empower(TS CC, Martin, TSO) ∧ Consider(TS CC, invoke_WS1, send) ∧ Use(TS CC, WS1-Image, DS CC arming request) ∧ Hold(TS CC, Martin, invoke_WS1, WS1-image, critical situation) ∧ ⇒ is-permitted(Martin, invoke_WS1, WS1-image)

Figure 3.24: OrBAC rule for the arming request at TS CC side.

On the DS CC side, WS1-arming-request asks DSO to arm some DS SS, i.e., to access object WS2-Image. These accesses (virtual-user1 to DSO display, DSO to WS2-Image) are checked according to DS CC policy and is granted according to the OrBAC rule described in Figure 3.25, managing the Access Control for both Arming Request and Arming Order Web services at the level of the organization that provides the service (i.e., DS CC).

Rules
Permission(DS CC, DSO, DS CC arming order, Send, critical situation) ∧ Empower(DS CC, virtual-user1, DSO) ∧ Consider(DS CC, invoke_WS2, Send) ∧ Use(DS CC, WS2-Image, DS CC arming order) ∧ Hold(DS CC, virtual-user1, invoke_WS2, WS2-Image, critical situation) ∧ ⇒ is-permitted(virtual-user1, invoke_WS2, WS2-Image)

Figure 3.25: OrBAC rule for the arming request/order at DSCC side.

When DSO accesses object WS-Image2, WS2-arming-order is automatically activated, then virtual-user2 activates Object-arm-MCDTU in DS SS, and finally the physical arming command is executed over the physical component MCDTU. This access (virtualuser2 to Object-arm-MCDTU) is checked according to DS SS policy and is granted according to the OrBAC rule described in Figure 3.26, managing Access Control for Arming Order web service in DS SS.WS3-prepare-for-LS, WS4-Load-shedding, andWS5-disarming/reintegration are negotiated and activated in the same way.

3.3.2.6 E-contracts Enforcement and Runtime Checking

For each negotiated WS, we specify two automata representing the established contract on the WS client and WS provider sides respectively. In this subsection, we explain how the WS1-arming-request automaton is specified for each side, and the WS1 client and provider automata are checked at runtime by the corresponding CIS.

Rules
Permission(DS SS, DSO for SS, Access, DS SS Distrib. Circuits, emergency) ∧ Empower(DS SS, virtual-user2(Subject), DSO for SS) ∧ Consider(DS SS, activate(action), Access) ∧ Use(DS SS, object-arm-MCDTU(object), DS SS Distrib. Circuits) ∧ Hold(DS SS, virtual-user2, activate, object-arm-MCDTU, emergency) ∧ ⇒ ispermitted(virtual-user2, activate, object-arm-MCDTU)

Figure 3.26: OrBAC rule for the arming order at DSSS side.

According to the WS1 contract, and as illustrated in Figure 3.27, at the TS CC side (the WS1 client) the WS1 automaton waits for an arming request invocation (coming from the TSO). When this invocation is intercepted, the corresponding transition (WS1-armingrequest) is activated in the automaton, a timer t is initialized, and the WS1 automaton of TS CC arrives to a state where it waits for a WS1-arming-request-ack. This acknowledgement should be sent by the DS CC. If the timeout expires without receiving the acknowledgment from the DS CC, a WS1-arming-request-error message triggers the exception corresponding to this situation. This exception will be handled by the specific automaton TS CC-errorhandling4. Conversely, in normal situations, when the TS CC receives the WS1-armingrequest-ack, its automaton goes the state where it is ready for an emergency action.

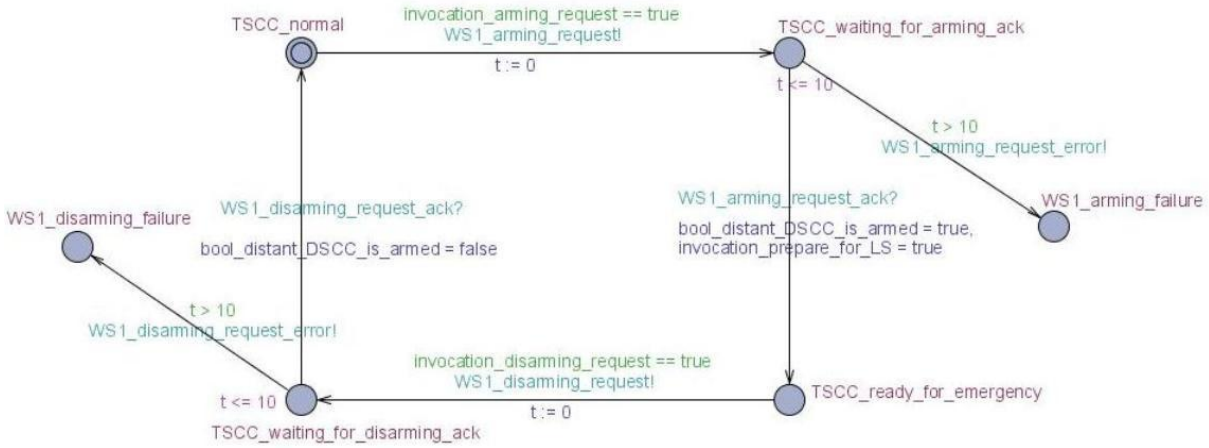


Figure 3.27: TSCC-WS1-arming automata.

In this state, when the critical situation disappears, the TSO can decide to disarm the distribution substations that were armed. Then, the TS CC sends the WS1-disarmingrequest synchronization to the DS CC, and waits for the WS1-disarming-request-ack from the DS CC. If the timeout (set to 10 time units in Figure 3.27 expires without receiving the WS1-disarming-request-ack synchronization, a WS1-disarming-request-error synchronization will be sent and then will be handled by the specific automaton TS CC-errorhandling.

Let us now analyze how the DS CC WS1 automaton (the WS1 provider side) reacts to invocations coming from the client side (Figure 3.28). Actually, this automaton is waiting for the WS1-arming-request synchronization from the TS CC; when it receives it, a timer is initialized.

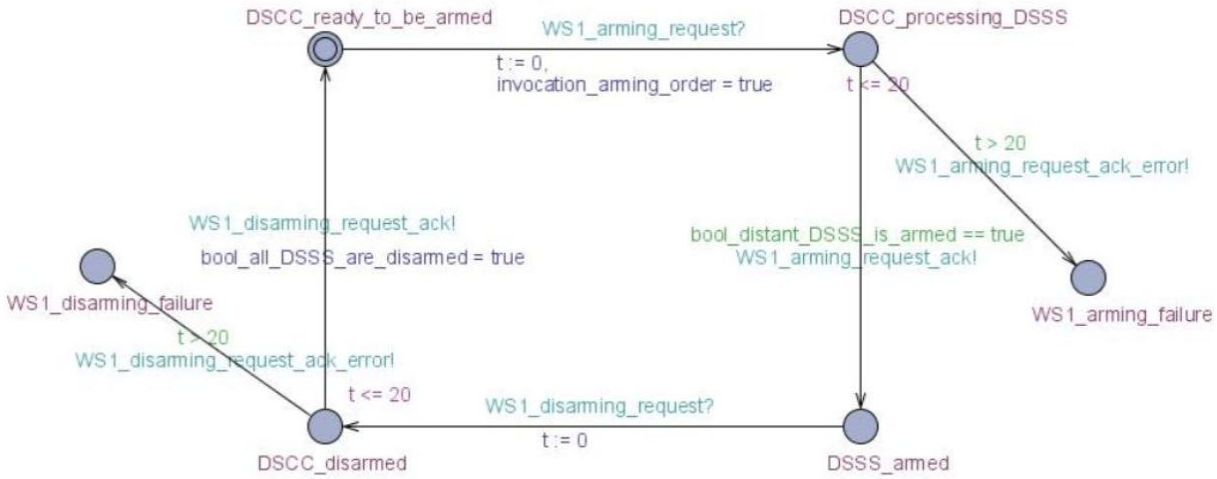


Figure 3.28: DSCC-WS1-arming automata.

Then the DS CC is in a state where it can arm some specific DS SSs. Note that these substations are chosen by the DSO (using the WS2-arming-order).

If the arming operation is not processed after the expected timeout, a WS1-armingrequest-error is sent, and then, is handled by the DS CCerror-handling automaton. If the DS CC is ready for emergency actions, it sends WS1-arming-request-ack. If the WS1 automaton of the DS CC side receives a WS1-disarming-request from the TS CC, the disarming operation is carried out and an acknowledgment (the WS1-disarming-requestack) is sent to the TS CC. In abnormal situations, if the arming is not processed after the timeout expiration, a WS1-disarming-request-error is sent and handled by the DS CC-error-handling automaton.

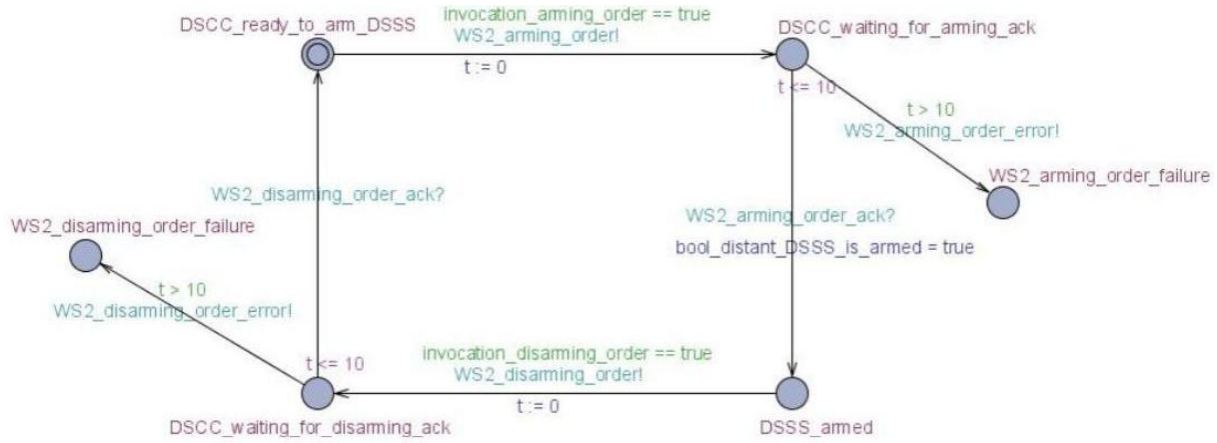


Figure 3.29: DSCC-WS2-arming-order automata.

At the DS CC side, to carry out DS SS arming, the WS2-arming-order synchronization is sent to all the DS SS that need to be armed. A timer is then initialized and the WS2 automaton of DS CC (is in a state where it) waits for arming acknowledgments (WS2arming-order-ack) coming from each armed DS SS. If the timeout expired without receiving the acknowledgments, a WS2-arming-order-error is sent and is handled by the DS CCerrorhandling automaton. Conversely,

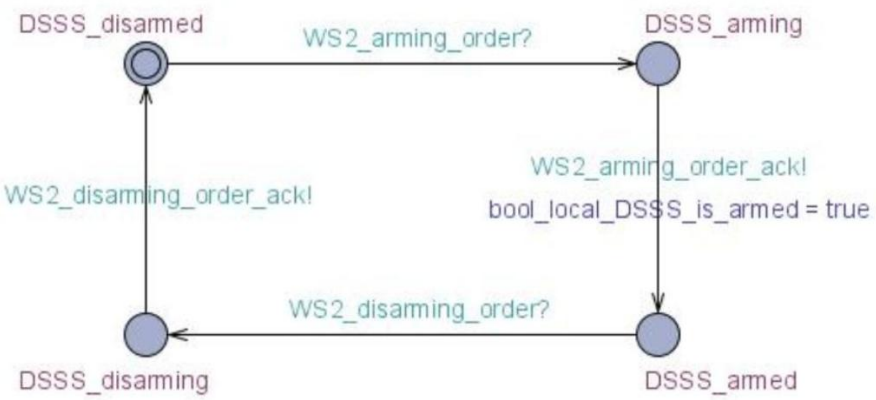


Figure 3.30: DSSS-WS2-arming-order automata.

in normal situations, when the DS CC receives the WS2arming-order-ack, it become ready for emergency actions, and a WS1-arming-request-ack is finally sent back to the TS CC.

Under certain situations, the DSO may decide to disarm some armed substations. In that case, it sends a WS2-disarming-request to each DS SS to disarm. Then, the DSCC WS2 automaton waits for the WS2-disarmingrequest-ack from the DS SS. If the timeout expired without receiving the acknowledgment (WS2-disarming-request-ack), a WS2disarming-requesterror is sent and is handled by the corresponding automaton (DS CCerrorhandling). At the DS SS side, the WS2 automaton waits for the WS2-arming-order synchronization. When the DS SS arming is carried out, an acknowledgment (WS2-arming-order-ack) is sent to the DS CC; the DS SS is now armed. Afterwards, if the DSSS receives a WS2disarming-order synchronization, it performs the disarming operation and sends back the WS2-disarming-order-ack to the DS CC.

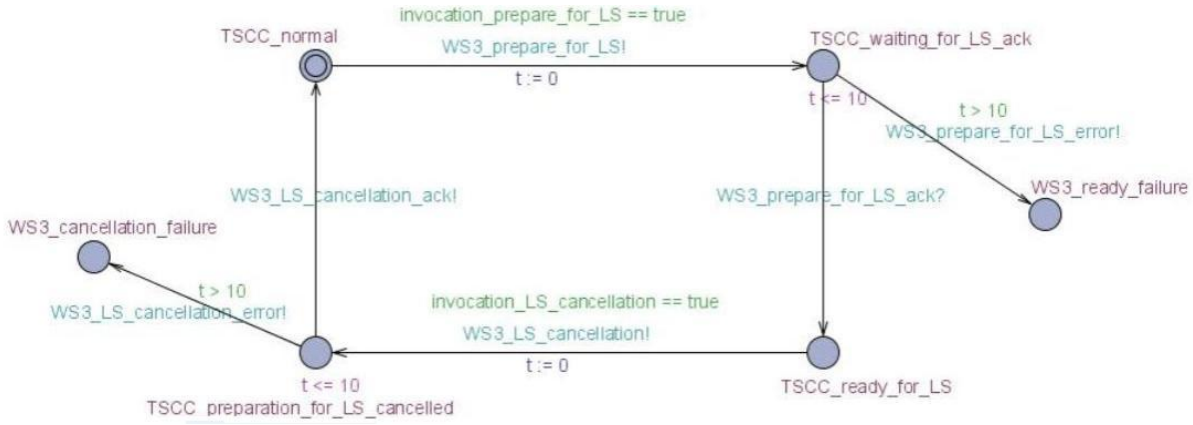


Figure 3.31: TSCC-WS3-loadshedding automata.

In emergency situations, when the TSO decides to launch the load shedding activity, he actually sends a prepare-for-LS invocation to the TS CC; the latter invokes WS3-preparefor-LS of the TS SS. Afterwards, if the TSO decides to cancel the loadshedding preparation, he sends a LS-cancellation to the TS CC, which then sends the WS3-LS-cancellation to the TS SS and waits for the acknowledgment (WS3-LS-cancellation-ack). At the TS SS side, the WS3 automaton receives the WS3-prepare-for-LS synchronization. The load shedding can now be carried out, and

the WS3-ready-for-LS-ack is sent back to the TS CC.

If the TS SS WS3 automaton intercepts the WS3-LS-cancellation synchronization, the load shedding preparation can be canceled and the corresponding acknowledgment (WS3-LS-cancellation-ack) is sent back.

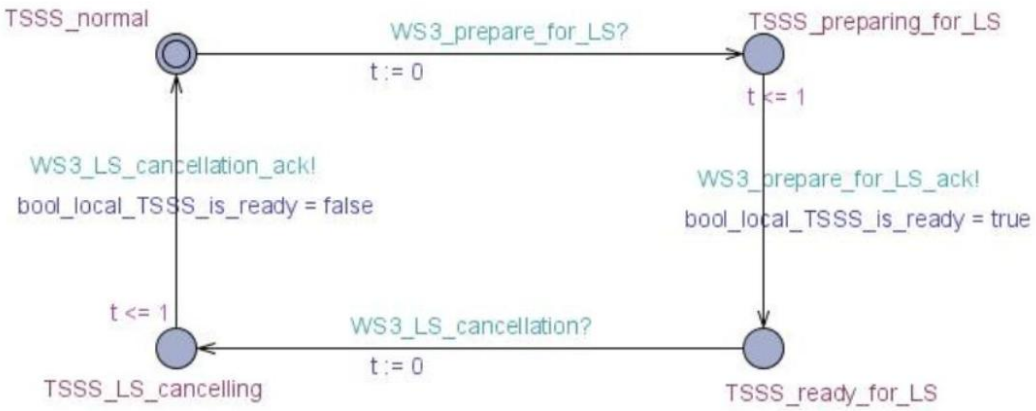


Figure 3.32: TSSS-WS3-loadshedding automata.

At the TS SS side, the WS4 automaton specifies that when some specific conditions are fulfilled (e.g., emergency situations), the TS SS can decide to launch the WS4loadshedding on the armed DS SS. At the DS SS side, the loadshedding operation is then automatic.

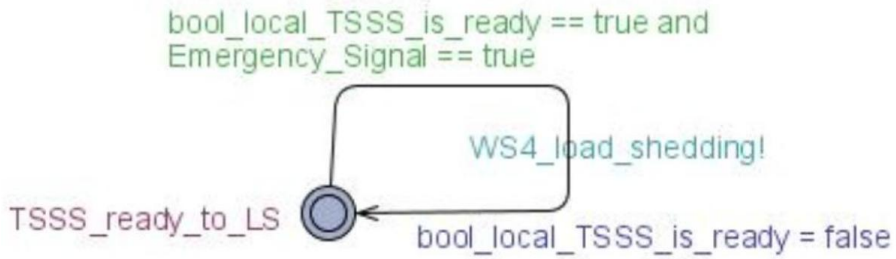


Figure 3.33: TSSS-WS4-loadshedding automata.

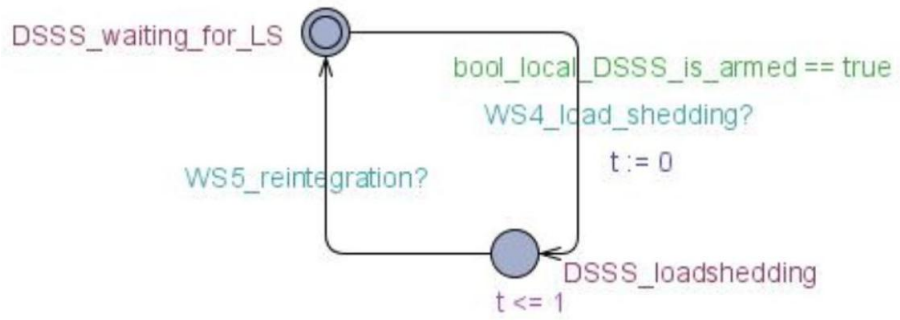


Figure 3.34: DSSS-WS4-loadshedding automata.

If the TSO decides that the emergency situation is not anymore true, he decides to reintegrate one or more substations, he informs the DSO, who invokes WS5-reintegration of the DS SS.

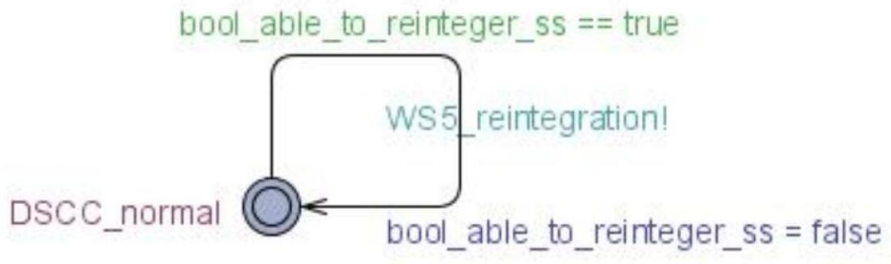


Figure 3.35: DSCC-WS5-reintegration automata.

3.3.2.7 Implementation

In order to illustrate the feasibility of the scenarios investigated in the previous sections with PolyOrBAC, we have developed a prototype implementing and simulating the web services as well as the e-contracts enforcement.

We use a decentralized architecture where each organization is responsible for its users authentication and locally controls the access to its own resources and services, according to the WS security architecture (i.e., with PDPs, PEPs, PAP and PIPs [86]). Basically, after receiving an access request (i.e., a web service invocation) from a contracting organization, the provider's Policy Enforcement Point (PEP), located in its CIS, sends the request to the Policy Decision Point (PDP), which evaluates the request and sends back a response. The response can be either access permitted or denied, with the appropriate obligations. The PDP comes to a decision after evaluating the relevant policies. To get the policies, the PDP uses the PAP (Policy Access Point) to extract the security rules (e.g., *Permission*(*Organization*, *Role*, *View*, *Activity*, *Context*)). The PDP may also invoke the Policy Information Point (PIP) service to retrieve the attribute values related to the organization, the subject, the web service (resource), or the environment (the context). This consists in evaluating the associations *Empower* (*org*, *s*, *r*), *Consider* (*org*, *a*, *activ*), *Use* (*org*, *o*, *v*) and *Hold* (*org*, *s*, *a*, *o*, *c*). The authorization decision is sent by the PDP to the PEP. The PEP fulfils the obligations and, based on the authorization decision sent by the PDP, either permits or denies access.

Actually, for simplicity reasons, we made the choice to simulate the whole network using OpenVZ, an open software providing virtualization with high performances (compared to other solutions such as VMware and Xen).

To implement and deploy our architecture, we used the J2EE specification, in particular its Java RMI (*Remote Method Invocation*), Servlet and Axis technologies. Indeed, to manage and deploy web services, we installed an Apache Tomcat server and an Apache Axis Web Service Framework on the machines proposing web services (e.g., DS CC, TS SS and DS SS). Apache Tomcat provides a classical web server as well as an implementation of Servlet and JSP contain-

ers. Apache axis is an XML based Web Service Framework, i.e., it allows: (1) deploying web services (using a "Web Service Deployment Descriptor" (WSDD) xml file), (2) generating "Web Service Deployment Language" (WSDL) xml files. Moreover, as web service invocations must be intercepted by the PEP, we decided to deploy a web service on the CIS to dispatch every incoming and exiting calls; the RMI technology handles remote communications.

Besides that, OrBAC rules and relationships are expressed in XML files while Web services are implemented by classes. Furthermore, JSP is used to implement Web Interfaces and Servlets are used to manage WS calls as well as interactions with CISs. To verify that interactions are carried out according to the signed contracts, we implemented a dedicated runtime model checker with *UPPAAL* [107, 71]. UPPAAL automatically converts our automata to XML files that we used in our Java program. UPPAAL also allows proving that the exchange protocol can be run according to the contract clauses.

Note that UPPAAL can also be used to verify properties expressed in a subset of timed Computational Tree Logic (*CTL*) [18]. In our case, we can verify if all the possible executions of the system will never lead to a conflicting situation, which is actually equivalent to verify the *reachability* property (note that we can also verify the *safety* and *liveness* properties of UPPAAL formulae). These properties are often used in designing a model to perform sanity checks. A reachability property will validate the basic behavior of the model. For example, the following property "*E <> organization.Dispute*" (expressed in CTL) stands for: "*it exists at least one execution where the organization reaches the dispute state*". Inversely, the property: "*A[] not organization.Dispute*" means that "*none of the possible executions will lead the organization to a dispute state*".

3.4 Monitoring and Failure Detection

This section provides the overall definition of the middleware services devoted to monitoring and failure detection activities. First a theoretical introduction to diagnosis is presented, including the rationale behind correlation of diagnosis information; then, following a top-down approach, the diagnosis scenarios identified in CRUTIAL are presented in detail.

3.4.1 The Diagnosis Framework

From a theoretical viewpoint, the general framework [34] for the diagnosis activity involves the following three actors (see also Figure 3.36):

- *Monitored Component (MC)*: It is the system component under diagnosis.
- *Deviation Detection Mechanism (DD)*: It is the entity introduced in the system to observe the external behavior of the monitored component and to judge whether that behavior is suitable or not; more than one deviation detection mechanisms could be put in place to observe the behavior of the MC.
- *State Diagnosis Mechanism (SD)*: It is the entity introduced in the system in charge of guessing the internal state of the monitored component, based on information collected overtime from the deviation detection mechanism(s).

Two information flows are necessary: the first involves the monitored component and the deviation detection mechanism (MC↔DD), because the DD has to observe the behavior of the MC; the second information flow involves the deviation detection mechanism and the state diagnosis mechanism (MC↔DD), because the SD has to collect information provided by the DD. 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 [96].

When diagnosis is performed on-line, streams of data on component behavior are collected and filtered over time; the filtering function could use an heuristic approach (e.g. alpha count [22]) or a probabilistic approach (diagnosis based on the hidden Markov model [34]).

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.

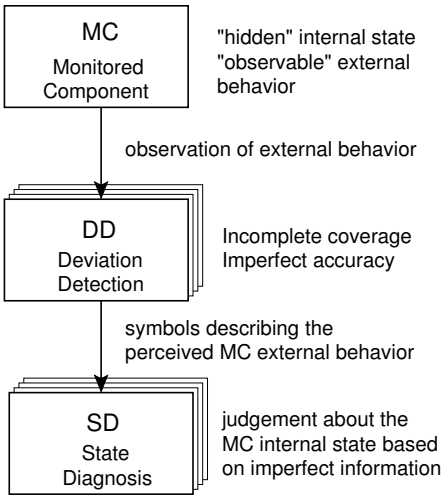


Figure 3.36: The diagnosis framework identifying the chain constituted by the monitored component, the error detection mechanism and the diagnosis mechanism.

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 by 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 [120] 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 Correlation of Diagnosis Information

We aim to identify borderline situations which altogether require an intervention. The treatment of "simple" errors is already incorporated within sub-systems; we aim to analyze situations requiring more complex tools. We aim to recognize and catalogue symptoms of several types coming from different sub-systems in order to be able to associate a name and a diagnosis to pattern of those symptoms (syndromes). Diagnosis is seen as the collection of symptoms which can be either already known or requiring classification as dangerous.

The monitoring activity on a certain system component is based on the observation of error signals occurring during the component lifetime; simple malfunctions, directly detected by component error detection mechanisms, are self evident error events. When the number and the complexity of monitored components grows up and the interdependencies among components behavior become larger, subtle malfunctions at component level can possibly give rise to erroneous manifestations elsewhere in the system, with patterns (set of symptoms) not immediately pointing to the actually faulty component. Moreover, system level improper conditions may arise from unexpected combinations of otherwise legal component behaviors, obviously signaled as correct events. In order to recognize these situations, it is necessary to look not only for single events, but also for sets of correlated events that altogether lead to system malfunction; we will call those sets as "poly-events".

The conceptual schema describing the operations of monitoring and failure detection within the system is presented hereafter. We assume that the information about the nature and effects of relevant events is recorded in a number of event sets ($\{SG\}$, $\{G\}$, $\{B\}$) and processed by some processes (*Collector*, *Normalizer*, *Aggregator*, *Recognizer*).

The conceptual schema, depicted in Figure 3.37, involves the following sets:

$\{$**SG**$\}$ is the repository of events occurred in the system, which are not yet associated to known poly-events; each event is kept in $\{SG\}$ until an event-specific deadline (possibly infinite) is reached.

$\{$**B**$\}$ is the set of "bad" poly-events, those which have been recognized as system malfunction syndromes; a "NYE" (Not Yet Established) flag is initially set for poly-events whose negative connotation is still to be confirmed.

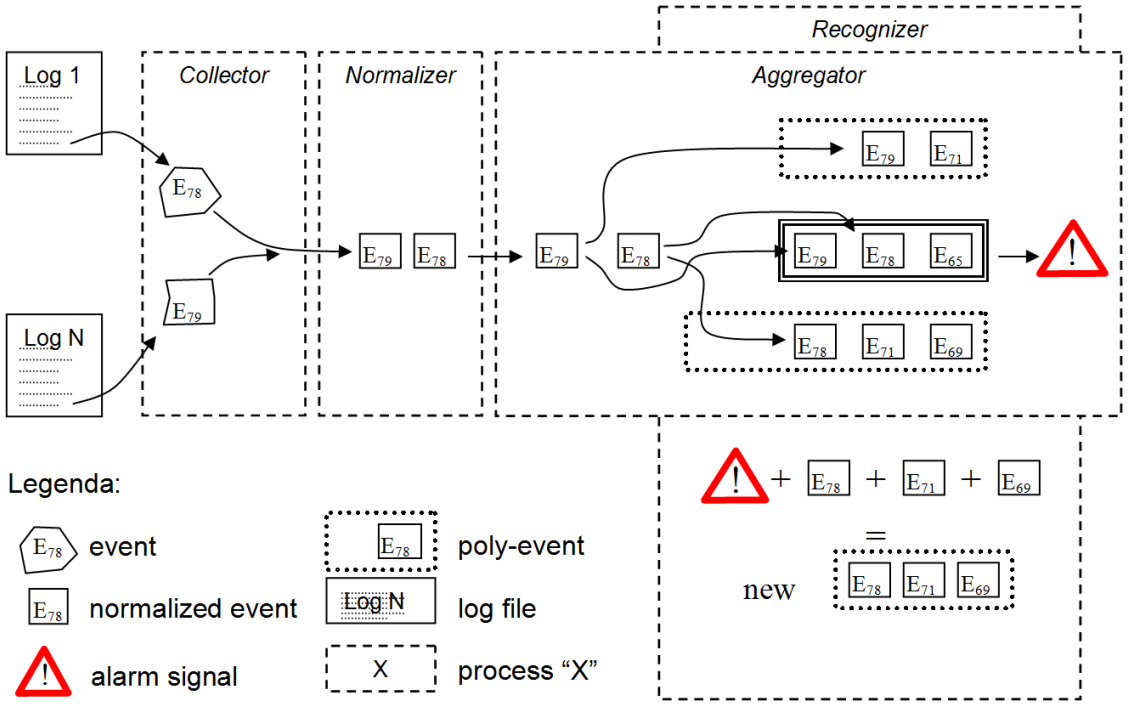


Figure 3.37: Data fusion: conceptual schema.

{**G**} is the set of "good" poly-events which represents normal situations; for the sake of efficiency, in the implementation only "good" poly-events that include several events found in {B} need possibly to be stored.

The above sets are managed by the processes described hereafter.

Collector. A *Collector* process collects information streams about events occurring in the system (e.g. events could be read from application- or system-log files); each event is characterized by several attributes (which are application- or system-dependent), some of which are the following:

1. *Timestamp* (to have a temporal reference)
2. *Type* (to classify them from a high point of view)
3. *Severity* (to be able to give priority to severe events)
4. *Architectural level and localization* (to be able to correlate events based on localization)
5. *TTL* (Time To Live) (to know how much time must be taken into account)
6. *Resettable* (or not) (to know whether a subsequent event can cancel it or not)

Normalizer. The format of the events collected by the *Collector* is source-dependent, so a *Normalizer* process works together with the *Collector* in order to translate events in records having a uniform format. The event record is build according to the format defined in the above attribute list. For easiness of presentation, let's use the same word "event" for both the source-formatted event collected by the *Collector* and the corresponding normalized event (record) translated by the *Normalizer*. Normalized events are stored in the $\{SG\}$ set.

Aggregator. The next step is to search $\{SG\}$ for anomalous conditions. An *Aggregator* process searches $\{SG\}$ for sub-sets of events that partially of fully match poly-events in $\{G\}$ and $\{B\}$ (i.e, making use of previous knowledge). The search criteria is the following: upon the notification of an event e_i, the *Aggregator* first checks its severity, to ascertain if an immediate action is to be taken - in that case it rushes over the proper action request. This step need still to be elaborated. Next, event e_i is matched against a number of event sets:

1. poly-events already open, waiting for further matches; add the event to all the open poly-events where a match is found; if a poly-event is now fully matched, the *Aggregator* sends the proper diagnostic result;
2. poly-events already known as symptoms of malfunction, collected in $\{B\}$; a matching poly-event is opened;
3. poly-events known as signs of normal operation, collected in $\{G\}$; a matching poly-event is opened.

Several matching criteria are used. To start, events in $\{SG\}$ obviously exhibit correlation in time, to some degree. The *Aggregator* process picks up a tentative poly-event, and looks for other correlations among events, trying to consolidate a more significant poly-event out of it: i.e. it associates other events, possibly taken from other elements in $\{B\}$, according to several rules, to extract a variegate picture of the system behavior. The rules are the following:

1. **t-rule**: events occurred in the last t time units with regard to e_i;
2. **s-rule**: events occurred in a set S of physical enclosures (circuit board, sub-assembly, rack, room) e_i occurred within;
3. **p-rule**: permanent events, always correlated until reset;
4. **a-rule**: events occurred as part of, or alongside to, an activity a affected by e_i.

For each rule a distance function is defined; the distance between two events is defined as a function (e.g. the RMS, root mean square) of the distances computed by each rule. The definition of the distance function is relevant for prevent the tendency of poly-events growing without bounds and small poly-events tending to be gobbled up by larger ones - while often small poly-events, perhaps single events, may be very significant! The tentative poly-event P_t is aggregated to a poly-event P_s if:

1. the number of events in P_t is lower than the number of events in P_s;
2. for each event e_t in P_t it is possible to find a corresponding event e_s in P_s such that the distance between e_t and e_s is lower than a given threshold.

If P_t cannot be aggregated to any pre-existent poly-event, it is set as a new element in $\{\mathrm{G}\}$, with a confidence parameter set to "low".

If none of the above matches succeeds, the *Aggregator* appends e_i to a time-ordered sequence which is being built up; this sequence will be truncated upon the occurrence of a "trigger" event, such as a bus error, an application crash, a network link down, an interval time expiring, a "max event count" is reached. The list of triggers is initially populated with known adverse events, and can be updated by the *Aggregator*, whenever a new error condition is shaped up. The events belonging to the truncated sequence are boxed into a tentative poly-event set, which is then inserted in $\{\mathrm{B}\}$ with the "NYE" flag set. The collection of a new sequence is started afterwards.

Recognizer. The *Recognizer* process is in charge of recognizing if a tentative poly-event, or a subset thereof, or even a superset thereof, is to be signaled to the error processing subsystem. It draws from previous knowledge of positively acknowledged malfunctions, but also from direct signals from higher levels, notifying out-of-spec behaviors possibly escaped to the monitoring subsystem. So, while the *Collector* and the *Aggregator* operate in a strict bottom-up fashion, the *Recognizer* acts both bottom-up and top-down; in this way it helps better to fill detection gaps.

The event set/distance approach depicted above offers both a clean architecture, and a straightforward learning process: new significant event sets (poly-events) are simply aggregated to existing ones, continuously incrementing the diagnostic knowledge. However, for the sake of a rapid trial implementation, for a preliminary evaluation of the validity of this approach in practical environments, we decided to use an already available tool to experiment with. This tool is the SEC (Simple Event Correlator) [109]: an open-source rule-based event correlation tool, created in an academic context.

SEC is conceived to do real-time log monitoring: it takes as input several streams of primitive events and processes them in order to detect composite events that correspond to event patterns in the event streams. SEC can be seen as a complex, context-aware filter which selects relevant detection information. SEC is based on a powerful implementation tool: regular expressions as matching rules. Rather complex matching patterns can be defined in a compact way, that would otherwise be quite awkward to express. In the SEC environment the previous knowledge is embedded in the coded matching rules, which are read in for execution at the program initialization. To provide for learning two more steps are needed:

- synthetization of new rules (regular expressions- based) upon occurrence of a significant, not yet recognized, poly-event;
- trigger a reloading of the updated rule-set, without interrupting the ongoing partial recognitions.

The last action is already supported by the current SEC version.

3.4.3 Diagnosis in CRUTIAL

The CRUTIAL infrastructure is organized as a WAN-of-LANs, where each LAN is connected to the WAN by a CIS (fore more details, please see Section 2.1). Given that the computers of the LANs cannot be modified/updated, all the monitoring and failure detection activities have to be performed inside each CIS. The following diagnosis scenarios arise [115]:

- *CIS self-diagnosis (local view)*: This is the diagnosis activity performed locally to a given CIS, aiming to monitor the CIS itself (e.g. to diagnose hardware or software faults).
- *LAN diagnosis (local view)*: This is the diagnosis activity aimed to monitor the nodes that are inside the LAN protected by the CIS (e.g to "measure" the trustworthiness level of a certain node).
- *CIS distributed diagnosis (global view)*: The CISs in the WAN 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).

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 a 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.

3.4.3.1 CIS Self-Diagnosis

From a local viewpoint the CIS is a sophisticated application level firewall, combined with equally sophisticated intrusion detectors; the CIS is hence required to be intrusion tolerant, to prevent resource exhaustion providing perpetual operation and to be resilient against fault assumption coverage uncertainty providing survivability. In order to comply with the above requirements, the CIS [19] 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:

- *Internal monitoring*: monitoring performed inside a single replica, trying to detect local failures (e.g. an intrusion).
- *External monitoring*: monitoring performed on the perceived behavior of the other replicas (e.g. to detect a replica crash).

The internal monitoring, given the information system malfunctions introduced in [83], has to be done on the following components/services:

- 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 CIS Protection Service.

How to use SEC for CIS Internal Self-diagnosis. CIS internal self-diagnosis can be implemented using SEC in CRUTIAL; this subsection reports considerations about this implementation.

The CIS is replicated with diversity, where diversity means also different operating systems for each replica: SEC is written in PERL, so it can be executed on several operating system platforms by using a PERL interpreter. SEC needs text-formatted streams of detection information: this is feasible, especially for system logs, which are typically text based. For example, Linux based operating systems support the "syslog" log-file; for the other operating systems, free tools are available to support the same logging system[5]. In order to speed-up the recognition of urgent signals, a "special" stream can be created ad-hoc (let's call it "emergency log") and the sensor driver can be forced to generate a log event not only in the log of the operating system, but also in the "emergency log".

SEC can perform the following activities:

- Reading log files, it processes streams of information about events occurring in the system (as the *Collector* is supposed to do);
- It triggers actions when specific events are recognized based on some rules (as the *Aggregator* is supposed to do);
- It correlates events on the base of some (originally static) rules; rules can be based on several properties of events, e.g.:
  a relative timing;
  b localization: each physical event generator is cataloged in a parallel-hierarchic data structure, reflecting the physical position - e.g., a CPU is located on a circuit board, which is in a card cage, which is in a server rack, which is in a room, etc. The distance between two elements is defined by traversing the structure along a path joining these elements.
  c architectural levels: a structure similar to b) above, where the vicinity is in general expressed in terms of interactions: two components directly interacting are "neighbor", if their interaction is mediated through 3 levels of other components their distance is "proportional" to 3, and so on. Rules are defined in specific configuration files (text format) called "rule files".

The SEC self-learning capabilities are masked, but existing. SEC is self-learning in the sense that configuration files can be refreshed at run-time, keeping the status of the correlation. SEC can

[5]The freely available "Snare for Windows" tool from Intersect Alliance can be used to convert Microsoft event logs to "syslog" messages.

be restarted from the hosting operating system using specific inter-process signals ("SIGABRT" or "SIGTERM"), possibly performing a "soft" reset, so that configuration files (rule files) are reloaded, but the "status" of the ongoing correlations is saved. In particular, a soft restart consists of the following steps:

1. Rule files are reopened (new files can be opened too);
2. Event correlation operations started from rule files that have been modified or removed after the previous configuration load will be canceled;
3. Other operations and other event correlation entities (contexts, variables, child processes, etc.) will remain intact.

The contexts that can be defined within SEC correspond to the chunks.

An Example of External Monitoring - Diagnosing the CIS Protection Service. The Protection Service (PS) (see also Section 3.3.1) is the middleware service performing egress/ingress access control, implementing an instance of the global security policy. The PS captures packets passing through the CIS, checking whether these packets satisfy the security policy or not; packets satisfying the security policy are forwarded to their destination node, whilst the others are discarded.

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

Monitoring is performed at payload level, where each instance of the PS 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.

3.4.3.2 LAN Diagnosis

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 nodes (e.g. replacing hardware, refreshing the software, changing passwords, ...).

A trustworthiness indicator for each protected node N is defined (it could be 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 a 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).

3.4.3.3 CIS Distributed Diagnosis

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 CIS Protection Service.

4 Runtime Support Services, APIs and Protocols

4.1 CIS Proactive-Reactive Recovery

This section describes the CIS proactive-reactive recovery service, its interface and protocols. We start by motivating the necessity of such a service (Section 4.1.1), and by explaining the system model in which the proactive-reactive recovery service is based (Section 4.1.2). Then, the service and its interface are described (Section 4.1.3) and the protocols used to implement it explained (Section 4.1.4). Finally, we will discuss how this service was integrated with the CIS, to give it self-healing properties (Section 4.1.5).

4.1.1 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* [113]. 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* [88]. These previous works propose intrusion-tolerant replicated systems that are resilient to any number of faults [28, 127, 26, 77, 102]. 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 [42, 41, 13, 57]. 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.

4.1.2 Model of the System

Recently, it was shown that proactive recovery can only be implemented with a few synchrony assumptions [103, 34]: 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 [111] 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 $P_1, ..., P_n$. 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 [87].

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 [56];
4. there is a timed atomic broadcast primitive with bounded maximum transmission time [56].

One should note that all of these services can be easily implemented in the crash-failure synchronous distributed system model [116].

4.1.3 Service Description and Interface

The Proactive Resilience Model (PRM). The proactive-reactive recovery service builds on the Proactive Resilience Model (PRM) introduced in CRUTIAL deliverable D10. 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 4.1 illustrates the rejuvenation process. Replicas are recovered in groups of at most k elements, by some specified order: for instance, replicas $\{P_1,...,P_k\}$ are recovered first, then replicas $\{P_{k+1},...,P_{2k}\}$ 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 $T_P \geq \lceil \frac{n}{k} \rceil T_D$.

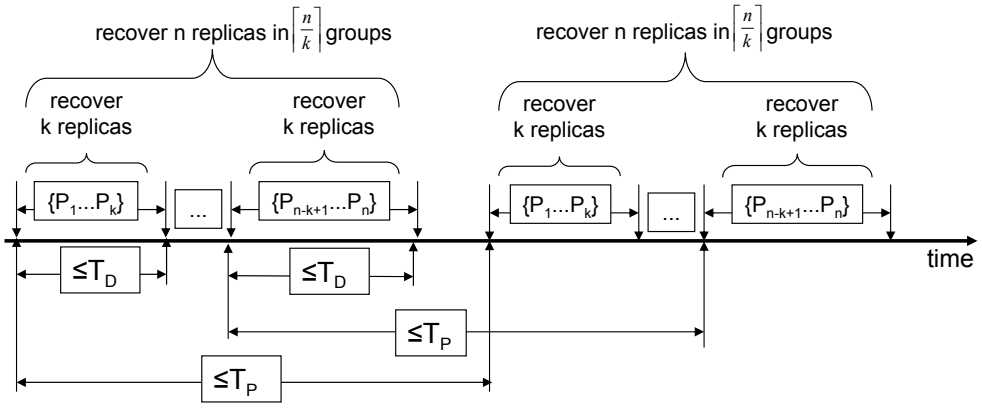


Figure 4.1: Relationship between the rejuvenation period T_P, the rejuvenation execution time T_D, and k.

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 4.2 presents this idea.

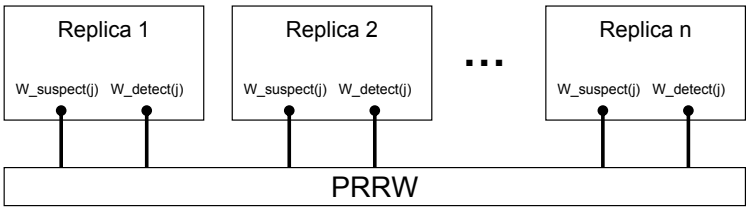


Figure 4.2: PRRW architecture.

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 [29] (or a muteness detector [42]) 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 [13, 57], 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) [29, 42, 13, 57], system monitoring [34] and/or intrusion detection [38, 82] 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 [104]. 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 4.3 illustrates how time-triggered periodic and event-triggered aperiodic recoveries are combined.

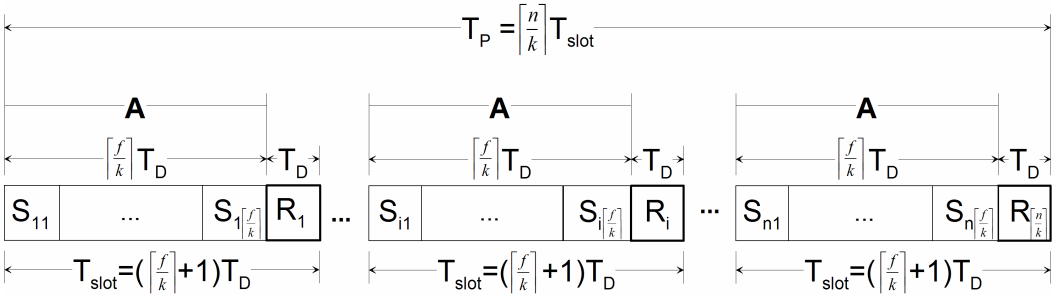


Figure 4.3: Recovery schedule (in an S_{ij} or R_i subslot there can be at most k parallel replica recoveries).

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.

4.1.4 Service Protocols

We developed two protocols to implement the proactive-reactive recovery service. The protocols are presented in Algorithm 10 and Algorithm 11. 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 4.1.3. We start with the description of Algorithm 10 and then Algorithm 11 is described.

Parameters and variables. Algorithm 10 uses six parameters: i, n, f, k, T_P, and T_D. The identifier (id) of the local wormhole is represented by $i \in \{1,...,n\}$; 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 4.3); 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 10 Proactive-reactive recovery service: main protocol.

{Parameters}
integer i {Id of the local wormhole}
integer n {Total number of replicas}
integer f {Maximum number of faulty replicas}
integer k {Max. replicas that recover at the same time}
integer T_P {Periodic recovery period}
integer T_D {Recovery duration time}

{Constants}
integer $T_{slot} \triangleq (\lceil \frac{f}{k} \rceil + 1)T_D$ {Slot duration time}

{Variables}
integer $t_{next} = 0$ {Instant of the next periodic recovery start}
set $Detect = \varnothing$ {Processes that detected me as failed}
set $Suspect = \varnothing$ {Processes suspecting me of being failed}
bool $scheduled = false$ {Indicates if a reactive recovery is scheduled for me}

{Reactive recovery interface threads with priority 3}
service $W\_suspect(j)$
8: $send(j, \langle \text{SUSPECT} \rangle)$
service $W\_detect(j)$
9: $send(j, \langle \text{DETECT} \rangle)$
upon $receive(j, \langle \text{SUSPECT} \rangle)$
10: $Suspect \leftarrow Suspect \cup \{j\}$
upon $receive(j, \langle \text{DETECT} \rangle)$
11: $Detect \leftarrow Detect \cup \{j\}$

{Periodic recovery thread with priority 1}
procedure $proactive\_recovery()$
12: $synchronize\_global\_clock()$
13: $t_{next} \leftarrow global\_clock() + (\lceil \frac{i-1}{k} \rceil T_{slot} + \lceil \frac{f}{k} \rceil T_D)$
14: **loop**
15: **wait until** $global\_clock() = t_{next}$
16: $recovery()$
17: $t_{next} = t_{next} + T_P$
18: **end loop**
procedure $recovery()$
19: $recovery\_actions()$
20: $Detect \leftarrow \varnothing$
21: $Suspect \leftarrow \varnothing$
22: $scheduled \leftarrow false$

{Reactive recovery execution threads with priority 2}
upon $|Detect| \geq f + 1$
23: $recovery()$
upon $(|Detect| < f + 1) \wedge (|Suspect \cup Detect| \geq f + 1)$
24: **if** $\neg scheduled$ **then**
25: $scheduled \leftarrow true$
26: $\langle s, ss \rangle \leftarrow allocate\_subslot()$
27: **if** $sooner(s, \lceil \frac{i}{k} \rceil)$ **then**
28: **wait until** $global\_clock() \bmod T_P = sT_{slot} + ssT_D$
29: $recovery()$
30: **end if**
31: $scheduled \leftarrow false$
32: **end if**

Reactive recovery service interface. As mentioned before, input information from payload replicas is needed in order to trigger reactive recoveries. This information is provided through two interface functions: $W\_suspect()$ and $W\_detect()$. $W\_suspect(j)$ 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 as it will be explained later in this section.

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 set the *scheduled* flag to *false* and re-initialize the *Detect* and *Suspect* sets 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 4.3. This is done through function *allocate_subslot*() (line 19 – explained later). Notice that if the calculated slot is going to occur later than the slot where the replica will be proactively recovered, 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 4.3.

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 11. This algorithm uses one more parameter and one more variable besides the ones defined in Algorithm 10. 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 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*()

Algorithm 11 Proactive-reactive recovery service: slot allocation protocol.

{Parameters (besides the ones defined in Algorithm 10)}
integer T_Δ {Bound on message delivery time}

{Variables (besides the ones defined in Algorithm 10)}
table $Subslot[\langle 1,1\rangle ... \langle \lceil \frac{n}{k} \rceil, \lceil \frac{f}{k} \rceil \rangle] = 0$ {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: $curr\_subslot \leftarrow \langle \lfloor \frac{t_{round}}{T_{slot}} \rfloor + 1, \lfloor \frac{t_{round} \bmod T_{slot}}{T_D} \rfloor + 1 \rangle$
7: $first \leftarrow true$
8: **loop**
9: $curr\_subslot \leftarrow next\_subslot(curr\_subslot)$
10: **if** $first$ **then**
11: $first \leftarrow false$
12: $first\_subslot \leftarrow curr\_subslot$
13: **else if** $curr\_subslot = first\_subslot$ **then**
14: **return** $\langle \lceil \frac{n}{k} \rceil + 1, \lceil \frac{f}{k} \rceil + 1 \rangle$
15: **end if**
16: **if** $Subslot[curr\_subslot] < k$ **then**
17: $Subslot[curr\_subslot] \leftarrow Subslot[curr\_subslot] + 1$
18: **return** *curr_subslot*
19: **end if**
20: **end loop**

procedure *next_subslot*($\langle s, ss\rangle$)
21: **if** $ss < \lceil \frac{f}{k} \rceil$ **then**
22: $ss \leftarrow ss + 1$
23: **else if** $s < \lceil \frac{n}{k} \rceil$ **then**
24: $ss \leftarrow 1$
25: $s \leftarrow s + 1$
26: **else**
27: $ss \leftarrow 1$
28: $s \leftarrow 1$
29: **end if**
30: **return** $\langle s, ss\rangle$

upon $(t_{round} \leftarrow (global\_clock() \bmod T_P)) \bmod T_{slot} = 0$
31: **if** $\frac{t_{round}}{T_{slot}} = 0$ **then**
32: $prev\_slot \leftarrow \lceil \frac{n}{k} \rceil$
33: **else**
34: $prev\_slot \leftarrow \frac{t_{round}}{T_{slot}}$
35: **end if**
36: $\forall p, Subslot[\langle prev\_slot, p\rangle] \leftarrow 0$

(called in Algorithm 10, line 19). This function 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). Note that the total order multicast protocol delivers all messages to all wormholes in the same order. At this point the (determinist) function *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 allocate the same subslots in the same order. The local allocation algorithm is implemented by the already mentioned *local_allocate_subslot*() function (lines 5-20). 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 ($t_{send} + T_\Delta$) 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 7-30). Finally, in the beginning of each recovery slot, all the subslots of the previous recovery slot are deallocated (lines 31-36).

To illustrate how the recovery subslot allocation works, let us analyze a simple example scenario. Assume that $n = 4$, $f = 1$, $k = 1$, $T_D = 150$ seconds, and $T_\Delta = 1$. From these values, we derive $T_{slot} = 300$ seconds, and $T_P = 1200$ seconds. Consider now that replica i receives $f + 1$ SUSPECT messages and in consequence calls *allocate_subslot*() because no reactive recovery is scheduled. Consider also that the ALLOC message is multicasted by replica i when global_clock() = 1999 seconds (line 1). This means that all replicas receive this message (by the same order) within 1 second and they all call *local_allocate_subslot*(*1999*) (line 3 for replica i, line 4 for the other replicas). The following actions are executed deterministically by every replica. t_{round} is set to 800 (2000 mod 1200 = 800 in line 5) and *curr_subslot* is set to $\langle 3,2 \rangle$ (line 6). Then, the next subslot calculated in line 9 returns $\langle 4,1 \rangle$ and *curr_subslot* is updated to this value. Line 16 checks if the calculated subslot has less than $k = 1$ scheduled recoveries, and if this is true, the number of scheduled recoveries is increased and this subslot is returned (lines 17-18). Otherwise, the next subslot is calculated (line 9) and the same verification is done until an available subslot being found. In our scenario, the next subslot would be $\langle 1,1 \rangle$, then $\langle 2,1 \rangle$, and so on. The code between lines 10 and 15 takes care of the case when there is no available subslot, by checking if the loop returns to the subslot from where it started. If this happens, an invalid subslot is returned (line 14) and the function *sooner* of Algorithm 10 (line 20) returns false, aborting the reactive recovery. The replica is only recovered later in its periodic recovery slot. Notice that this exhaustion of recovery subslots is impossible to occur in our example scenario, given that there is a total of $\lceil \frac{n}{k} \rceil \lceil \frac{f}{k} \rceil = 4$ subslots, one per replica. In general, recovery subslots exhaustion can only occur when the condition $\lceil \frac{n}{k} \rceil \lceil \frac{f}{k} \rceil < n$ is true, i.e., when $k > 1$.

The deallocation of (reactive) recovery subslots is triggered at the beginning of each recovery slot (lines 31-36). In the example scenario, it would be triggered each $T_{slot} = 300$ seconds. The deallocation procedure starts by calculating the previous slot ($\frac{t_{round}}{T_{slot}}$). If the previous slot is zero (line 31), it means that the actual previous slot was the last of the previous period (line 32), because slots are numbered between 1 and n. Otherwise, the previous slot corresponds to the one calculated (line 34). The deallocation ends by setting to zero the number of scheduled recoveries of each subslot of the previous slot.

4.1.5 Integrating PRRW in the CIS

In this section we discuss how the PRRW was integrated in the CIS Protection service to give it self-healing properties (SH-CIS, as described in Section 3.3.1.6).

There are three main issues that must be addressed when integrating the PRRW on a intrusion tolerant system (and, in particular, the SH-CIS): the addition of extra replicas to allow availability even during recoveries, the implementation of the *recovery_actions*() procedure (i.e., what actions are done when it is time to recover a replica) and defining in which situations the *W_suspect* and *W_detect* PRRW interface functions are called by a replica.

Extra Replicas. As explained in Section 4.1.3, at most k correct replicas are recovered at the same time. This means that the intrusion-tolerant CIS enhanced with the PRRW needs a set of

$n \geq 2f + k + 1$ replicas to make use of the PRRW proactive-reactive recovery service without endangering availability. In the simplest scenario where one fault is tolerated between recoveries ($f = 1$) and a single replica is recovered at the same time ($k = 1$), the CIS combined with the PRRW needs a total of four replicas.

Recovery Actions. In the case of the CIS, the implementation of the *recovery_actions*() procedure comprises the execution of the following sequence of steps:

(i.) if the replica to be recovered is the current CIS leader, then a new leader must be elected: a message is sent by the local wormhole of the current leader to all local wormholes informing that the new leader is the last replica that finished its periodic recovery;

(ii.) the replica is deactivated, i.e., its operating system is shutdown;

(iii.) the replica operating system is restored using some clean image (that can be different from the previous one);

(iv.) the replica is activated with its new operating system image.

Step (i.) is needed only because our replication algorithm uses a leader to forward approved messages to their destinations, and the wormhole is responsible for maintaining it (as described in Section 3.3.1.6. In step (iii.) the wormhole can select one from several pre-generated operating system images to be installed on the replica. These images can be substantially different (different operating systems, kernel versions, configurations, access passwords, etc.) to enforce fault independence between recoveries. In step (iv.) we assume that when the system is rebooted the CIS software is started automatically.

Calling PRRW Interface Functions. The PRRW interface functions for informing suspicions and detections of faults are called by the CIS replicas when they observe something that was not supposed to happen and/or when something that was supposed to happen does not occur. In the case of the CIS, the constant monitoring of the protected network allows a replica to detect some malicious behaviors from other replicas. Notice that this can only be done because our architecture has a traffic replication device inside the LAN (ensuring that all replicas see every message) and it is assumed that all messages sent by the CIS replicas to the LAN are authenticated.

In the SH-CIS, there are two situations in which the PRRW interface functions are called:

(i.) Some replica sends an invalid message to the protected network: if a correct replica detects that some other replica transmitted an illegal message (one that was not signed by the wormhole) to the LAN, it can call *W_detect* informing that the replica behaved in a faulty way. From Algorithm 10 it can be seen that when $f + 1$ replicas detect a faulty replica, it is recovered;

(ii.) The leader fails to forward a certain number of approved messages: if a correct replica knows that some message was approved by the wormhole and it does not see this message being

forwarded to the LAN, it can conclude that something is wrong with the current leader (which was supposed to send the message). Due to the many imprecisions that may happen in the system (asynchrony, message losses due to high traffic), it is perfectly possible that a correct leader did not receive the message to be approved or this message was forwarded but some replica did not receive it from the LAN. To cope with this, we define an omission threshold for the leader which defines the maximum number of omissions that a replica can perceive from some leader replica before suspecting it to be faulty. Notice that it is impossible to know with certainty if the leader is faulty, therefore replicas call *W_suspect* and not *W_detect* in this case. From Algorithms 10 and 11 it can be seen that when $f+1$ replicas *suspect* the leader, a recovery is scheduled for it.

These two interactions between the CIS replicas and the PRRW service increases significantly the difficulty for a malicious adversary to launch any attack to the protected LAN (in the sense that it will have much less time before a recovery is started).

5 Conclusions

This document describes the architecture, services and protocols of the project CRUTIAL. The CRUTIAL Architecture intends to reply to a grand challenge of computer science and control engineering, on how to achieve resilience of critical information infrastructures, in particular in the electrical sector. We believe that the project has given important steps to address the problem, and in particular we have contributed to the definition of 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 document that we have indeed provide answers to meet these objectives. We have presented several designs of a CII protection device called CIS, which offers various trade-offs in terms of resilience and cost. Additionally, we have also given a specification of services and protocols for the CRUTIAL Architecture, including both Runtime Support Services and Middleware Services.

Bibliography

[1] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miege, C. Saurel, and G. Trouessin. Organization based access control. In *Proceedings of 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, June 2003.

[2] A. Abou El Kalam and Y. Deswarte. Multi-OrBAC: A new access control model for distributed, heterogeneous and collaborative systems. In *Proceedings of the IEEE Symposium on Systems and Information Security*, 2006.

[3] A. Abou El Kalam and Y. Deswarte. Critical infrastructures security modeling, enforcement and runtime checking. In *Proceedings of the 3rd International Workshop on Critical Information Infrastructures Security*, 2008.

[4] A. Abou El Kalam, Y. Deswarte, A. Baïna, and M. Kaâniche. Access control for collaborative systems: A web services based approach. In *Proceedings of the IEEE International Conference on Web Services*, 2007.

[5] M. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, 2000.

[6] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 353–364, 2003.

[7] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2), April 1994.

[8] D. Andersen, H. Balakrishnan, M. Frans Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 131–145, 2001.

[9] D. Andersen, H. Balakrishnan, M. Frans Kaashoek, and R. Rao. Improving web availability for clients with MONET. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 115–128, 2005.

[10] D. Andersen, A. Snoeren, and H. Balakrishnan. Best-path vs. multi-path overlay routing. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pages 91–100, 2003.

[11] Arbor Networks, http://www.arbornetworks.com. *The peakflow platform*. Visited in July 2008.

[12] A. Baina, A. Abou El Kalam, Y. Deswarte, and M. Kaaniche. A collaborative access control framework for critical infrastructures. In *Proceedings of the Second Annual IFIP WG 11.10 International Conference on Critical Infrastructure Protection*, March 2008.

[13] R. Baldoni, J.-M. Hélary, M. Raynal, and L. Tangui. Consensus in Byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, April 2003.

[14] H. Beitollahi and G. Deconinck. Analysis of peer-to-peer networks from a dependability perspective. In *proceedings of the 3th IEEE International Conference on Risks and Security of Internet and Systems*, pages 101–108, October 2008.

[15] H. Beitollahi and G. Deconinck. Dependable overlay networks. In *Proceedings of 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 104–111, December 2008.

[16] H. Beitollahi and G. Deconinck. An overlay protection layer against denial-of-service attacks. In *Proceedings of the 22rd IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, April 2008.

[17] S. Bellovin. Distributed firewalls. *;login: The USENIX Magazine*, November 1999.

[18] B. Berard, M. Bidiot, A. Finkel, F. Larousinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification, Model Checking Techniques and Tools*. Springer, 2001.

[19] A. Bessani, P. Sousa, M. Correia, N. Neves, and P. Verissimo. Intrusion-tolerant protection for critical infrastructures. DI/FCUL TR 07-8, Department of Informatics, University of Lisbon, April 2007.

[20] A. Bessani, P. Sousa, M. Correia, N. Neves, and P. Verissimo. The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy*, 6(6):44–51, November-December 2008.

[21] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano. Effective fault treatment for improving the dependability of COTS and legacy-based applications. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):223–237, 2004.

[22] 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.

[23] 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.

[24] E. Byres and J. Lowe. The myths and facts behind cyber security risks for industrial control systems. In *Proceedings of the VDE Kongress*, 2004.

[25] J. Cabrera, L. Lewis, X. Qin, W. Lee, R. Prasanth, B. Ravichandran, and R. K. Mehra. Proactive detection of distributed denial of service attacks using mib traffic variables- a feasibility study. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management*, pages 609–622, May 2001.

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

[27] C. Cachin and A. Samar. Secure distributed DNS. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 423–432, 2004.

[28] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[29] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), March 1996.

[30] M. Chow and Y. Tipsuwan. Network-based control systems: a tutorial. In *Proceeding of The 27th IEEE Annual Conference of Industrial Electronics Society*, pages 1593–1602, 2001.

[31] 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.

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

[33] F. Cuppens, N. Cuppens-Boulahia, and C. Coma. O2O: Virtual private organizations to manage security policy interoperability. In *Proceedings of the Second International Conference on Information Systems Security*, 2006.

[34] 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 *Proceedings of 25th IEEE Symposium on Reliable Distributed Systems*, pages 245–256, October 2006.

[35] T. Darmohray and R. Oliver. *Hot spares for DDoS attacks*. http://www.usenix.org/publications/login/2000-7/apropos.html.

[36] G. Deconinck, H. Beitollahi, G. Dondossola, F. Garrone, and T. Rigole. Testbed deployment of representative control algorithms. Deliverable D9, Project CRUTIAL EC IST-FP6-STREP 027513, January 2008.

[37] G. Deconinck, G. Dondossola, F. Garrone, and T. Rigole. Testbeds deployment of representative control algorithms (interim report). Deliverable D24, Project CRUTIAL EC IST-FP6-STREP 027513, January 2007.

[38] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[39] F. Depaoli and L. Mariani. Dependability in peer-to-peer systems. *IEEE Transactions on Internet Computing*, 8(4):54–61, July and August 2004.

[40] D. Dittrich. *The DoS project's trinoo distributed denial of service attack tool*. University of Washington, 1999.

[41] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash to byzantine failures. In *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, pages 24–50, 2002.

[42] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings of the 3rd European Dependable Computing Conference*, pages 71–87, September 1999.

[43] C. Douligeris and A. Mitrocotsa. DDoS attacks and defense mechanisms: Classification and state-of-the-art. *The International Journal of Computer and Telecommunications Networking*, 44(5):643 – 666, April 2004.

[44] D. Dzung, M. Naedele, T. Von Hoff, and M. Crevatin. Security for industrial communication systems. *Proceedings of the IEEE*, 93(6):1152–1177, 2005.

[45] F. Garrone (editor). Analysis of new control applications. Deliverable D2, Project CRUTIAL EC IST-FP6-STREP 027513, January 2007.

[46] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical approaches to DDoS attack detection and response. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 303–314, April 2003.

[47] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. Internet Engineering Task Force, RFC 2267, January 1998.

[48] D. Ferraiolo, R. S. Sandhu, S. Gavrila, D. Kuhn, and R. Chandramouli. A proposed standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3), 2001.

[49] L. Fink and K. Carlsen. Operating under stress and strain. *IEEE Spectrum*, March 1978.

[50] 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.

[51] 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.

[52] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 protocol. Netscape Communications Corp., November 1996.

[53] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):47–59, 1982.

[54] L. Gordon, M. Loeb, W. Lucyshyn, and R. Richardson. CSI/FBI computer crime and security survey. Computer Security Institute, 2006.

[55] K. Gummadi, H. Madhyastha, S. Gribble, H. Levy, and D. Wetherall. Improving the reliability of internet paths with one-hop source routing. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 13–13, 2004.

[56] 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.

[57] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proceedings of the 2nd Workshop on Hot Topics in System Dependability*, 2006.

[58] J. Han, D. Watson, and F. Jahanian. Topology aware overlay networks. *Proceedings od the INFOCOM*, 4:2554–2565, March 2005.

[59] K. Hickman. The SSL protocol. Netscape Communications Corp., February 1995.

[60] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement*, pages 237–242, 2002.

[61] International Electrotechnical Commission. Inter control center protocol (ICCP). IEC 60870-6 TASE.2, 1997.

[62] International Electrotechnical Commission. Communication networks and systems in substations. IEC 61850, 2003.

[63] J. Ioannidis and S. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. In *Proceedings of the ISOC Network and Distributed System Security Symposium*, February 2002.

[64] C. Jin, H. Wang, and K. Shin. Hop-count filtering: An effective defense against spoofed traffic. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 30–41, October 2003.

[65] 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.

[66] S. Kent. IP authentication header. Internet Engineering Task Force, RFC 4302, December 2005.

[67] S. Kent and R. Atkinson. Security architecture for the internet protocol. IETF Request for Comments: RFC 2093, November 1998.

[68] A. Keromytis, V. Misra, and D. Rubenstein. SOS: An architecture for mitigating DDoS attacks. *Journal on Selected Areas in Communications*, 22(1):176 – 188, January 2004.

[69] Y. Kim, W. Lau, M. Huah, and J. Chao. PacketScore: Statistics-based overload control against distributed denial-of-service attacks. In *Proceedings of the INFOCOM*, pages 2594–2604, March 2004.

[70] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication authors. Internet Engineering Task Force, RFC 2104, February 1997.

[71] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Journal of Software Tools for Technology Transfer*, 1997.

[72] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang. Save: Source address validity enforcement protocol. In *Proceedings of the INFOCOM*, June 2002.

[73] N. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.

[74] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *ACM SIGCOMM Computer Communication Review*, 32(3):62–73, July 2002.

[75] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.

[76] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C. Chuah, Y. Ganjali, and C. Diot. Characterization of failures in an operational IP backbone network. *IEEE/ACM Transactions on Networking*, 16(4):749–762, 2008.

[77] 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.

[78] Mazu Networks, http://www.mazunetworks.com/white_papers. *Mazu technical white papers*. Visited in July 2008.

[79] J. Mirkovic. *D-WARD: Source-end defense against distributed denial-of-service attacks*. PhD thesis, University of California Los Angeles, 2003.

[80] J. Mirkovic, G. Prier, and P. L. Reiher. Attacking DDoS at the source. In *Proceedings of the 10th IEEE International Conference on Network Protocols*, pages 312–321, September 2002.

[81] D. Moore, G. Voelker, and S. Savage. Inferring Internet denial-of-service activity. *ACM Transactions on Computer Systems*, 24(2):115 – 139, May 2006.

[82] B. Mukherjee, L. Heberlein, and K. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, 1994.

[83] N. Neves and P. Verissimo (editors). Preliminary architecture specification. Deliverable D4, Project CRUTIAL EC IST-FP6-STREP 027513, January 2007.

[84] N. Neves and P. Verissimo (editors). Preliminary specification of services and protocols. Deliverable D10, Project CRUTIAL EC IST-FP6-STREP 027513, January 2008.

[85] OASIS. Universal Description, Discovery and Integration, v3.0.2. UDDI Specifications TC, 2005.

[86] OASIS. Organization for the advancement of structured information standards. OASIS Web Services Security TC, Web Services Security v1.1 (WS-Security), 2006.

[87] R. Obelheiro, A. Bessani, L. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Department of Informatics, University of Lisbon, 2006.

[88] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings 10th ACM Symposium on Principles of Distributed Computing*, pages 51–59, 1991.

[89] 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*, pages 15–26, August 2001.

[90] V. Paxson and M. Allman. Computing TCP's retransmission timer. Internet Engineering Task Force, RFC 2988, November 2000.

[91] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[92] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of network-based defense mechanisms countering the DoS and DDoS problems. *ACM Computing Surveys*, 39(1):3, 2007.

[93] T. Peng, K. Ramamohanarao, and C. Leckie. Protection from distributed denial of service attacks using history-based IP filtering. In *Proceedings of the IEEE International Conference on Communications*, pages 482–486, May 2003.

[94] K. Poulsen. *FBI busts alleged DDoS mafia*. www.securityfocus.com/news/9411, 2004.

[95] 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.

[96] L. Romano, A. Bondavalli, S. Chiaradonna, and D. Cotroneo. Implementation of threshold-based diagnostic mechanisms for COTS-based applications. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 296–303, October 2002.

[97] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2), 1996.

[98] F. Schneider. Implementing fault-tolerant services using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[99] F. Schneider and L. Zhou. Implementing trustworthy services using replicated state machines. *IEEE Security & Privacy*, 3(5):34–43, September 2005.

[100] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. *ACM SIGOPS Operating Systems Review*, 35(5):160 – 173, 2001.

[101] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *Proceedings of the 13th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 373–380, December 2007.

[102] P. Sousa, N. Neves, A. Lopes, and P. Verissimo. On the resilience of intrusion-tolerant distributed systems. DI/FCUL TR 06-14, Department of Informatics, University of Lisbon, September 2006.

[103] P. Sousa, N. Neves, and P. Verissimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 98–107, June 2005.

[104] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1), 1989.

[105] 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.

[106] 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*, pages 149– 160, August 2001.

[107] UPP Sweden and AAL Denmark. Uppaal. http://www.uppaal.com, 2008.

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

[109] R. Vaarandi. SEC: A lightweight event correlation tool. In *Proceedings of the IEEE Workshop on IP Operations and Management*, pages 111–115, 2002.

[110] P. Verissimo. Lessons learned with NavTech: a framework for reliable large-scale applications. DI/FCUL TR 02–17, Department of Informatics, University of Lisbon, December 2002.

[111] P. Verissimo. Travelling through wormholes: A new look at distributed systems models. *SIGACT News*, 37(1):66–81, 2006.

[112] P. Verissimo, N. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch. Intrusion-tolerant middleware: The road to automatic security. *IEEE Security & Privacy*, 4(4):54–62, Jul./Aug. 2006.

[113] P. Verissimo, N. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In A. Romanovsky R. de Lemos, C. Gacek, editor, *Architecting Dependable Systems*, LNCS 2677, pages 3–36, 2003.

[114] P. Verissimo, N. Neves, and M. Correia. CRUTIAL: The blueprint of a reference critical information infrastructure architecture. In *Proceedings of the 1st International Workshop on Critical Information Infrastructures*, August 2006.

[115] P. Verissimo, N. Neves, M. Correia, A. Abou El Kalam, Y. Deswarte, A. Bondavalli, and A. Daidone. The CRUTIAL architecture for critical information infrastructures. In R. de Lemos et al., editor, *Architecting Dependable Systems V*, LNCS 5135, pages 1–27, August 2008.

[116] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[117] W3C. SOAP, Version 1.2. W3C Recommendation, 2003.

[118] W3C. Extensible Markup Language (XML). W3C Recommendation, 2004.

[119] W3C. Web Services Description Language (WSDL), Version 2.0. W3C Recommendation, 2006.

[120] C. Walter, P. Lincoln, and N. Suri. Formally verified on-line diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, 1997.

[121] H. Wang, D. Zhang, and K. Shin. Detecting SYN flooding attacks. In *Proceedings of the INFOCOM*, pages 1530–1539, June 2002.

[122] J. Wang and A. Chien. Understanding when location-hiding using overlay networks is feasible. *Journal of Computer Networks*, 50(6):763 – 780, April 2006.

[123] J. Wang, X. Liu, and A. Chien. Empirical study of tolerating denial-of-service attacks with a proxy network. In *Proceedings of 14th USENIX Security Symposium*, July 2005.

[124] 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*, pages 43 – 52, October 2003.

[125] C. Wilson. Terrorist capabilities for cyber-attack. In M. Dunn and V. Mauer, editors, *International CIIP Handbook 2006*, volume II, pages 69–88. Center for Security Studies, ETH Zurich, 2006.

[126] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement form execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, 2003.

[127] L. Zhou, F. Schneider, and R. Van Rennesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.