Resilient Intrusion Tolerance through Proactive and Reactive Recovery*

Paulo Sousa Alysson Neves Bessani Miguel Correia
Nuno Ferreira Neves Paulo Verissimo
LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal
{pjsousa, bessani, mpc, nuno, pjv}@di.fc.ul.pt

Abstract

Previous works have studied how to use proactive recovery to build intrusion-tolerant replicated systems that are resilient to any number of faults, as long as recoveries are faster than an upper-bound on fault production assumed at system deployment time. In this paper, we propose a complementary approach that combines proactive recovery with services that allow correct replicas to react and recover replicas that they detect or suspect to be compromised. One key feature of our proactive-reactive recovery approach is that, despite recoveries, it guarantees the availability of the minimum amount of system replicas necessary to sustain system's correct operation. We design a proactive-reactive recovery service based on a hybrid distributed system model and show, as a case study, how this service can effectively be used to augment the resilience of an intrusion-tolerant firewall adequate for the protection of critical infrastructures.

1 Introduction

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). 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* [21]. 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.

*This work was partially supported by the EC through project IST-2004-27513 (CRUTIAL) and NoE IST-4-026764-NOE (RESIST), and by the FCT through project POSI/EIA/60334/2004 (RITAS) and the Large-Scale Informatic Systems Laboratory (LaSIGE).

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* [14]. These previous works propose intrusion-tolerant replicated systems that are resilient to any number of faults [5, 23, 4, 11, 16]. 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 can disturb system normal operation without sacrificing periodic rejuvenation, which ensures that even dormant faults will be removed from the system.

This paper proposes the combination of proactive and reactive recovery in order to increase the overall performance and resilience of intrusion-tolerant systems that seek perpetual unattended correct operation. 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 replicas available to sustain correct operation. To the best of our knowledge, we are the first to combine reactive and proactive recovery in a single approach.

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 [8, 1, 10]. However, we argue that an important class of malicious faults can be detected, namely the ones generated automatically by malicious programs such as virus and worms. These kinds of attacks usually 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 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 proposed in this paper.

In order to show how the proactive-reactive recovery service can be used to enhance the dependability of a system and to evaluate the effectiveness of this approach, we applied it to the construction of an intrusion-tolerant protection device (a kind of firewall) for critical infrastructures. This device, called CIS (CRUTIAL Information Switch) Protection Service, CIS for short, is a fundamental component of an architecture for critical infrastructures protection proposed by some of the authors recently [20] in the context of the EU-IST CRUTIAL project[1]. The CIS augmented with proactive-reactive recovery represents a very strong and dependable solution for the critical infrastructures protection problem: this firewall is shown to resist powerful Denial-of-Service (DoS) attacks from both outside hosts (e.g., located in the Internet) and inside compromised replicas, while maintaining availability and an adequate throughput for most critical infrastructures' applications.

This paper presents the following contributions: *(i.)* it introduces the concept of proactive-reactive recovery and presents a design for a generic proactive-reactive recovery service that can be integrated in any intrusion-tolerant system; *(ii.)* it shows how imperfect failure detection (i.e., suspicions) can be managed to recover suspected replicas without sacrificing the availability of the overall system; and *(iii.)* it presents and evaluates an intrusion-tolerant perpetually resilient firewall for critical infrastructure protection, which uses the proactive-reactive recovery service.

2 Proactive-Reactive Recovery

Recently, some of the authors showed that proactive recovery can only be implemented with a few synchrony assumptions [17]: 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 approach proposed in this paper is based on a hybrid system model [19]. Before presenting the proactive-reactive approach and its foundation model, we precisely state the system model on which it is based.

[1] CRitical UTility InfrastructurAL resilience: `http://crutial.cesiricerca.it`.

2.1 System Model

We consider a hybrid system model [19] 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 $P_1, ..., P_n$. For the purpose of our paper, 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 presented in Section 3 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 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 [13].

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: wormhole clocks have a known precision, obtained by a clock synchronization protocol; there is point-to-point timed reliable communication between every pair of local wormholes; there is a timed reliable broadcast primitive with bounded maximum transmission time [9]. All of these services can be implemented in the crash-failure synchronous system model [22].

2.2 The Proactive Resilience Model (PRM)

The PRM [16] 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. 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 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 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

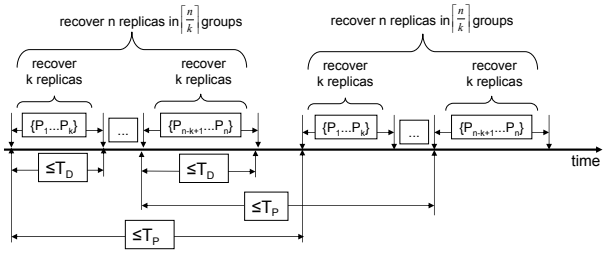


Figure 1. Relationship between rejuvenation period T_P, rejuvenation execution time T_D, and k.

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 sufficient replicas to tolerate f Byzantine servers and k simultaneous recoveries.

Figure 1 illustrates the rejuvenation process. Replicas are proactively 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 $T_P \geq \lceil \frac{n}{k} \rceil T_D$.

2.3 The Proactive-Reactive Recovery Wormhole (PRRW)

The PRRW offers a single service: 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 $W\_detect(j)$.

A payload replica i calls $W\_suspect(j)$ to notify the PRRW that the replica j is suspected of being faulty. This means that replica i suspects replica j but it does not know for sure if it is really faulty. Otherwise, if replica i knows without doubt that replica j is faulty, 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 [6] (or a muteness detector [8]) 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 [1, 10], 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 service provided by the proactive-reactive recovery wormhole 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) [6, 8, 1], system monitoring [7] and/or intrusion detection [12] 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 accuracy.

2.3.1 Scheduling recoveries without harming availability

The proactive-reactive recovery service initiates recoveries both periodically (time-triggered) and whenever something bad is detected or suspected (event-triggered). As explained in Section 2.2, 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 reactive 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 [18]. 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

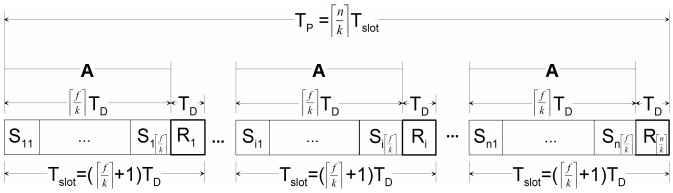


Figure 2. Recovery schedule (in an S_{ij} or R_i subslot there can be at most k parallel recoveries).

with the payload detected/suspected of being faulty are (aperiodically) recovered. The time needed for executing one A and one R_i is called the *recovery slot i* and is denoted by T_{slot}. Every slot i has $\lceil \frac{f}{k} \rceil$ *recovery subslots* belonging to the A task, each one denoted by S_{ip}, plus a R_i. Figure 2 illustrates how periodic and aperiodic recoveries are combined.

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 paper 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 if a replica is suspected of being 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 that replica i is faulty (i.e., if $f+1$ or more 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.

2.3.2 The PRRW algorithm

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

Parameters and variables. This algorithm uses six parameters: i, n, f, k, T_P, and T_D. The id of the local wormhole is represented by i; n specifies the total number of replicas and consequently the total number of local wormholes; f defines the maximum number of faulty replicas; k specifies the maximum number of replicas that recover at the same time; T_P defines the maximum time interval between consecutive triggers of the *recovery* procedure (depicted in Figure 2); and T_D defines the worst case execution time of the recovery of a replica. Additionally, four variables are defined: t_{next} stores the instant when the next periodic recovery should be triggered by local wormhole i; the *Detect* set contains the processes that detected the failure of replica i; the *Suspect* set contains the processes that suspect replica i of being failed; and *scheduled* indicates if a reactive recovery is scheduled for replica i.

Algorithm 1 Wormhole proactive-reactive recovery service.

```
{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 ≜ (⌈f/k⌉ + 1)T_D {Slot duration time}
{Variables}
integer t_next = 0 {Instant of the next periodic recovery start}
set Detect = ∅ {Processes that detected me as failed}
set Suspect = ∅ {Processes suspecting me of being failed}
bool scheduled = false {Is a reactive recovery scheduled for me ?}
{Reactive recovery interface threads with priority 3}
service W_suspect(j)
 1: send(j, ⟨SUSPECT⟩)
service W_detect(j)
 2: send(j, ⟨DETECT⟩)
upon receive(j, ⟨SUSPECT⟩)
 3: Suspect ← Suspect ∪ {j}
upon receive(j, ⟨DETECT⟩)
 4: Detect ← Detect ∪ {j}
{Periodic recovery thread with priority 1}
procedure proactive_recovery()
 5: synchronize_global_clock()
 6: t_next ← global_clock() + (⌈(i−1)/k⌉ T_slot + ⌈f/k⌉ T_D)
 7: loop
 8:     wait until global_clock() = t_next
 9:     recovery(); t_next ← t_next + T_P
10: end loop
procedure recovery()
11: recovery_actions()
12: Detect ← ∅ ; Suspect ← ∅ ; scheduled ← false
{Reactive recovery execution threads with priority 2}
upon |Detect| ≥ f + 1
13: recovery()
upon (|Detect| < f + 1) ∧ (|Suspect ∪ Detect| ≥ f + 1)
14: if ¬scheduled then
15:     scheduled ← true
16:     ⟨s, ss⟩ ← allocate_subslot()
17:     if s ≠ ⌈i/k⌉ then
18:         wait until global_clock() mod T_P = sT_slot + ssT_D
19:         if |Suspect ∪ Detect| ≥ f + 1 then recovery()
20:     end if
21: end if
```

Reactive recovery service interface. $W\_suspect(j)$ and $W\_detect(j)$ send, respectively, a SUSPECT or DETECT message to wormhole j, which is the wormhole in the suspected/detected node (lines 1-2). When a local wormhole i receives such a message from wormhole j, j is inserted in

the *Suspect* or *Detect* set according to the type of the message (lines 3-4). The content of these sets may trigger a recovery.

Proactive recovery. The *proactive_recovery*() procedure is triggered by each local wormhole i at boot time (lines 5-10). 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 7-10). The *recovery*() procedure (lines 11-12) starts by calling the abstract function *recovery_actions*() (line 11) 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 line of the *recovery*() procedure re-initializes some variables because the replica should now be correct.

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 13); (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 14), the aperiodic task finds the closest slot where the replica can be recovered without endangering the availability of the replicated system. The idea is to allocate one of the (reactive) recovery subslots depicted in Figure 2. This is done through the function *allocate_subslot*() (line 16 – explained later). Notice that if the calculated subslot $\langle s, ss \rangle$ is located in the slot where the replica will be proactively recovered, i.e., if $s = \lceil \frac{i}{k} \rceil$, then the replica does not need to be reactively recovered (line 17). If this is not the case, then local wormhole i waits for the allocated subslot and then recovers the corresponding replica (lines 18-19). Notice that the expression $global\_clock()$ mod T_P returns the time elapsed since the beginning of the current period, i.e., the position of the current global time instant in terms of the time diagram presented in Figure 2.

Recovery subslot allocation. Subslot management is based on accessing a data structure replicated in all wormholes through a timed total order protocol, as described in [15].

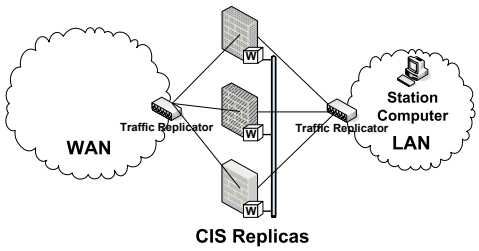


Figure 3. CIS protection service.

3 Case Study: The CIS Protection Service

In this section we describe how the PRRW component can be extended to make a perpetual-resilient operational system. The described system is a protection device for critical information infrastructures (a kind of improved application layer firewall) called *CIS Protection Service*. Here we only present a high-level view of the system focusing in the PRRW integration. The complete description is presented in [3].

3.1 Context

Recently, some of the authors proposed a reference architecture to protect critical infrastructures, in the context of the CRUTIAL EU-IST project [20]. The idea is to model the whole infrastructure as a set of protected LANs, representing the typical facilities that compose it (e.g., power transformation substations or corporate offices), which are interconnected by a wider-area network (WAN). Using this architecture, we reduce the problem of critical infrastructures protection to the problem of protecting LANs from the WAN or other LANs. 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). A fundamental component of this architecture is the *CRUTIAL Information Switch* (CIS), which is deployed at the borders of a LAN. The *CIS Protection Service* ensures that the incoming and outgoing traffic in/out of the LAN satisfies the security policy of the infrastructure.

A CIS can not be a simple firewall since that would put the critical infrastructure at most at the level of security of current (corporate) Internet systems, which is not acceptable since intrusions in those systems are constantly being reported. Instead, the CIS has several different characteristics, being the most important its intrusion tolerance, i.e., it operates correctly even if there are intrusions in some of its components and withstands a high degree of hostility from the environment, seeking unattended perpetual operation. In the next sections we show how the basic intrusion-tolerant design for the CIS can be integrated to the PRRW to make it attain perpetual correction.

3.2 How the CIS Works

The intrusion-tolerant CIS is replicated in a set of $n \geq 2f+1$ machines[2] connected both to the protected LAN and the insecure WAN through traffic replicators (e.g., a hub or a switch).

[2]The CIS design presented here assumes that access control policies are stateless. In [3] we explain how statefull policies could be supported.

Figure 3 depicts the intrusion-tolerant CIS architecture. Each replica of the CIS has a trusted and timely local wormhole [19] (the *W* boxes in Figure 3) that cannot be corrupted. These *local wormholes* are connected through an isolated network. Moreover, each CIS replica is deployed in a different operating system (e.g., Linux, FreeBSD), and the operating systems are configured to use different passwords.

In a nutshell, the message processing is done in the following way: each *CIS replica* receives all packets to the LAN and verifies if these packets satisfy some pre-defined application-level security policy. If a message is in accordance with the policy, it is *accepted* by the CIS, and must be forwarded to its destination in the LAN. Every message approved by a replica is issued to the wormhole to be signed. The local wormholes vote between themselves and, if the message is approved by at least $f+1$ replicas, it is signed using a secret key installed in the wormhole component. Once the message is signed, one of the replicas (the leader) is responsible for forwarding the approved message to its destination. Besides message signing, the wormhole is responsible also for leader election. The traffic replication devices in Figure 3 are responsible for broadcasting the WAN and LAN traffic to all replicas. The LAN replication device is specially useful to detect if malicious replicas send non-approved messages to the LAN.

3.3 Integrating the CIS and the PRRW

There are two main issues that must be addressed when integrating the PRRW into an intrusion-tolerant application: the implementation of the *recovery_actions*() procedure and defining in which situations the *W_suspect* and *W_detect* PRRW services are called by a replica.

In the case of the CIS, the *recovery_actions*() comprise 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 requires a leader and the wormhole is responsible to maintain it. In step (iii.) the wormhole can select one from several pre-generated operating system images to be installed on the recovered replica. These images can be substantially different (different operating systems, kernel versions, 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.

The PRRW services 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 (Figure 3) has a traffic replicator inside the LAN (ensuring that all replicas see all messages sent by every other replica to the LAN) and it is assumed that all messages sent by the CIS replicas to the LAN are authenticated.

Currently, there are two situations in which the PRRW services are called: *(i.) Some replica sends an invalid message to the protected network*: if a correct replica detects that some other replica sent an illegal message (one that was not signed by the wormhole) to the LAN, it can detect this replica as faulty and call *W_detect* informing that the replica presented a faulty behaviour. From Algorithm 1 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 forward the message). Due to the many imprecisions considered in the system (asynchrony, replica's message losses due to high traffic), it is perfectly possible that a correct leader did not received 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, so replicas call *W_suspect* and not *W_detect* in this case. From Algorithm 1 it can be seen that when $f+1$ replicas *suspect* a faulty replica, a recovery is scheduled for it.

3.4 Prototype

Our implementation uses the XEN virtual machine monitor [2] with the Linux operating system. XEN may host multiple guest operating systems, every one executed within an isolated VM or, in XEN terminology, a *domain*. The first domain, *dom0*, is created automatically when the system boots and has special privileges. Domain *dom0* builds other domains (*dom1, dom2,* etc). It also performs administrative tasks such as suspending and resuming other VMs, and it can be configured to execute with higher priority than the remaining VMs.

Every CIS replica uses XEN to isolate the payload from the wormhole part. Local wormholes run in replicas' domain *dom0*, and the CIS protection service executes in replicas' domain *dom1*. Domain *dom0* is configured to execute with higher priority than domain *dom1* in every replica, in order to emulate the real time behavior required by PRRW services. The local wormholes are connected through an isolated control network. More details on the prototype can be found in [15].

4 Experimental Evaluation

The experimental setup was composed by a set of four machines representing the CIS replicas ($n = 4$) connected to the three networks defined in our prototype architecture: LAN, WAN, and the control network. We used three additional PCs

	Shutdown	Rejuv.	Reboot	Complete
Average	0.6	72.2	70.1	142.9
Std. Deviation	0.5	1.2	0.3	0.9
Maximum	1.0	74.0	71.0	146.0

Table 1. Time needed (in seconds) for the several steps of a recovery (1.7 GB OS images).

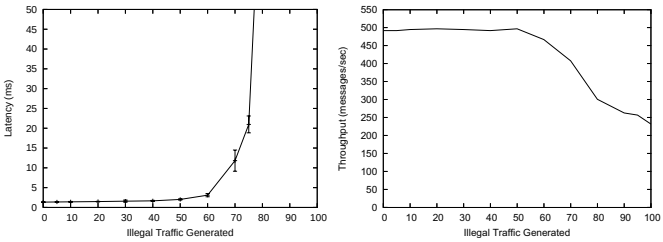
in the experiments. One PC was connected to the LAN emulating the station computer and, in the WAN side, two PCs were deployed: a *good* sender trying to transmit legal traffic to the station computer, and a *malicious* sender sending illegal messages to the LAN (equivalent to a DoS attack).

Recoveries performance. In the first experiment we tried to find appropriate values for parameters T_D (recover time) and T_P (recover period). We measured the time needed for each recovery task in a total of 300 recovery procedures executed during CIS operation. Table 1 shows the average, standard deviation, and maximum time for each recovery task: CIS shutdown, CIS rejuvenation by restoring its disk with a clean image randomly selected from a set of predefined images with different configurations, and the reboot of this new image.

From Table 1 one can see that a maximum of 146 seconds are needed in order to completely recover a virtual machine in our environment, being most of this time spent on two tasks: (1.) copying a clean pre-configured disk image from a local repository; and (2.) starting this new image (including starting the CIS protection service).

The results from the first experiment allowed to define $T_D = 150$ seconds for the remaining experiments described below. Considering that we had $n = 4$ replicas to tolerate $f = 1$ faults and $k = 1$ simultaneous recoveries, we used the expressions defined in Section 2.3.1 to calculate the maximum time between two recoveries of an individual replica as $T_P = 1200$ seconds (20 minutes). By applying these values to the PRM model [16], we conclude that a malicious adversary has at most $T_P + T_D = 22.5$ minutes to compromise more than f replicas and to harm the safety of the proposed system (i.e., make the CIS sign an illegal message) in our experimental setup.

Latency and throughput under a DoS attack from the WAN. In the second set of experiments, we tried to evaluate how much legal traffic our intrusion-tolerant firewall can deliver while it is being attacked by an outsider. In these experiments there is: a good sender (in the WAN) constantly transmitting 1470 bytes' packets of legal traffic, at a rate of 500 packets per second, to the station computer inside the LAN; and there is a malicious sender (in the WAN) launching a DoS attack against the CIS, i.e., sending between 0 and the maximum possible rate ($\sim$ 100 Mbps) of illegal traffic to it. We measured the received message rate at the station computer to obtain the throughput of the CIS (the rate at which it can approve messages) when it has to reject large amounts of illegal traffic. In a different experiment we measured the latency imposed by the CIS message approval, also in the presence of DoS attacks of different rates. In this latency experiment, the good sender sends a packet with 1470 bytes to the station computer that acknowledges it. This acknowledgment is not processed by the CIS and we measured the round-trip time in the good sender. All experiments (bandwidth and latency) were executed 1000 times and Figure 4 shows the average latency and maximum throughput measured in these experiments.

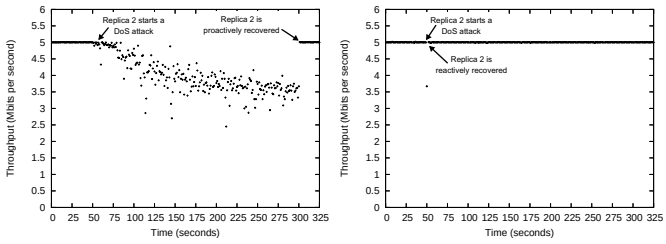


(a) (b)

Figure 4. Average latency (a) and maximum throughput (b) of the CIS in forwarding legal messages while a malicious host in the WAN is sending illegal traffic.

The graphs show that the system is almost unaffected by DoS attacks up to 50 Mbps, and then its behavior degrades gracefully until 70 Mbps. After this value, the latency presents a huge increase (Figure 4(a)) and the throughput drops to about 250 messages/sec (Figure 4(b)). These results suggest that our design adds modest latency (less than 2 ms) and no throughput loss even with a reasonably loaded network.

Throughput under a DoS attack from a compromised replica. Finally, the third set of experiments measured the resilience of the CIS against Byzantine faults, i.e., in the presence of up to f compromised replicas. Given that the CIS algorithms already tolerate up to f Byzantine faults, we choose a malicious behavior orthogonal to the algorithms logic that could nevertheless endanger the quality of the service provided by the CIS. In this way, we configured one of the CIS replicas (replica 2) to deploy a DoS attack 50 seconds after the beginning of CIS execution. This DoS attack floods the LAN with packets of 1470 bytes sent at a rate of 90 Mbps. We observed how the throughput is affected during this attack and until the replica being recovered. In order to show the effectiveness of our proactive-reactive recovery approach, we compared what happens when only proactive recoveries are used, and when they are combined with reactive recoveries. The results are presented in Figure 5. Figure 5(a) shows that the CIS throughput is affected during the DoS attack from replica 2 when only proactive recovery is used. The throughput decreases during the attack and reaches a minimum value of 2.45 Mbps (half of the expected throughput). The attack is stopped when the local wormhole of replica 2 triggers a proactive recovery after 300 seconds of the initial time instant. Notice that this recovery should be triggered 150 seconds later but given that we assume here that there are no reactive recoveries, we do not need the reactive recovery subslots depicted in Figure 2 and proactive recoveries may be triggered one after the other. The utility and effectiveness of combining proactive and reactive recoveries is illustrated by Figure 5(b), which shows that the CIS through-



(a) With proactive recovery only. (b) With proactive and reactive recovery.

Figure 5. Throughput of the CIS during 325 seconds. Replica 2 is malicious and launches a DoS attack to the LAN after 50 seconds.

put is minimally affected by the DoS attack from replica 2. This attack is detected by the remaining replicas and a reactive recovery is triggered immediately after the attack being launched. The reaction is so fast that the throughput drops to 3.67 Mbps just during one second and then it gets back to the normal values.

5 Conclusions

This paper proposed the combination of proactive and reactive recovery in order to increase the overall resilience of intrusion-tolerant systems that seek perpetual unattended correct operation. In addition to the guarantees of the periodic rejuvenations triggered by proactive recovery, our proactive-reactive recovery service ensures that, as long as a fault exhibited by a replica is *detectable*, this replica will be recovered as soon as possible, ensuring that there is always an amount of replicas available to sustain system's correct operation. To the best of our knowledge, this is the first time that reactive and proactive recovery are combined in a single approach.

We showed how the proactive-reactive recovery service can be used in a concrete scenario, by applying it to the construction of the CIS, an intrusion-tolerant firewall for critical infrastructures. The experimental results allow to conclude that the proactive-reactive recovery service is indeed effective in increasing the resilience and the performance of the CIS, namely in the presence of powerful DoS attacks launched either by outside hosts or inside compromised replicas.

References

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

[2] P. Barham, B. Dragovic, K. Fraiser, S. Hand, T. Harris, A. Ho, R. Neugebaurer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symp. on Operating Systems Principles - SOSP'03*, Oct. 2003.

[3] A. N. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Verissimo. Intrusion-tolerant protection for critical infrastructures. DI/FCUL TR 07-8, Dep. Informatics, Univ. Lisbon, Apr 2007.

[4] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. of the Int. Conf. on Dependable Systems and Networks - DSN 2002*, pages 167–176, June 2002.

[5] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), Mar. 1996.

[7] 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 *Proc. of the 25th IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 245–256, Leeds, UK, October 2006.

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

[9] 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.

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

[11] M. A. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE TDSC*, 1(1):34–47, Jan. 2004.

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

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

[14] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, pages 51–59, 1991.

[15] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. DI/FCUL TR 07–17, Dep. of Informatics, Univ. of Lisbon, September 2007.

[16] P. Sousa, N. F. Neves, A. Lopes, and P. Verissimo. On the resilience of intrusion-tolerant distributed systems. DI/FCUL TR 06–14, Dep. of Informatics, Univ. of Lisbon, Sept 2006.

[17] P. Sousa, N. F. Neves, and P. Verissimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proc. of Int. Conf. on Dependable Systems and Networks (DSN)*, pages 98–107, June 2005.

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

[19] P. Verissimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1), 2006, http://www.navigators.di.fc.ul.pt/docs/abstracts/ver06travel.html.

[20] P. Verissimo, N. F. Neves, and M. Correia. CRUTIAL: The blueprint of a reference critical information infrastructure architecture. In *Proc. of CRITIS'06 1st Int. Workshop on Critical Information Infrastructures Security*, Aug. 2006.

[21] P. Verissimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 2003.

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

[23] L. Zhou, F. Schneider, and R. Van Rennesse. COCA: A secure distributed online certification authority. *ACM TOCS*, 20(4):329–368, Nov. 2002.