Highly Available Intrusion-Tolerant Services with Proactive-Reactive Recovery

Paulo Sousa, *Member, IEEE,* Alysson Neves Bessani, Miguel Correia, *Member, IEEE,*
Nuno Ferreira Neves, *Member, IEEE,* Paulo Verissimo, *Fellow, IEEE*

***Abstract*—In the past, some research has been done on 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 enhances proactive recovery with additional reactive mechanisms giving correct replicas the capability of recovering other replicas that are detected or suspected of being compromised. One key feature of our proactive-reactive recovery approach is that, despite recoveries, it guarantees the availability of a minimum number of system replicas necessary to sustain correct operation of the system. 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 increase the resilience of an intrusion-tolerant firewall adequate for the protection of critical infrastructures.**

***Index Terms*—Intrusion Tolerance, Proactive Recovery, Reactive Recovery, Firewall.**

I. 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 gained momentum recently, promising to satisfy this requirement, is *intrusion tolerance* [1]. 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.

Classical intrusion-tolerant solutions based on Byzantine fault-tolerant replication algorithms assume 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 [2].

Recently, some research showed that this problem can be minimized if replicas are rejuvenated periodically, using a technique called *proactive recovery*. These previous works propose intrusion-tolerant replicated systems that are resilient to any number of faults [3], [4], [5], [6]. The idea is simple: replicas are rejuvenated from time to time 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 *to force the recovery of a replica that is detected or suspected of being faulty*. 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 the normal operation of a system, without sacrificing periodic rejuvenation. This ensures that even dormant faults will be removed from the system. The key property of the approach is that, as long as the faulty behavior exhibited by a replica is *detectable*, this replica will be recovered as soon as possible, ensuring that there are always a certain number of system replicas available to sustain system's 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 [7], [8], [9], [10]. However, we argue that an important class of malicious faults can be detected, specially the ones generated automatically by malicious programs such as viruses, worms, and even bots. These kinds of exploit programs 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

P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo are with the Department of Computer Science, University of Lisboa, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: {pjsousa,bessani,mpc,nuno,pjv}@di.fc.ul.pt.

policy in a way such that there is always a sufficient number of replicas for the system to be available. In fact, dealing with imperfect failure detection is one of the most complex aspects of the proactive-reactive recovery service proposed in this work.

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), is a fundamental component of an architecture to increase the resilience of critical infrastructures proposed in the context of the EU-IST CRUTIAL (CRitical UTility InfrastructurAL resilience) project [11], [12]. The CIS augmented with proactive-reactive recovery represents a very strong and dependable solution for protecting the infrastructures — this firewall is shown to resist powerful Denial-of-Service (DoS) attacks from both outside hosts (e.g., located somewhere in the Internet) and inside compromised replicas, while maintaining availability and an adequate throughput for most critical infrastructures' applications.

This work 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.

II. Proactive-Reactive Recovery

In previous works, some authors have showed that proactive recovery can only be implemented in systems with some synchrony [13], [14]. 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 work is based on a hybrid system model [15]. Before presenting the proactive-reactive approach and its foundation model, we precisely state the system model on which it is based.

A. System Model

We consider a hybrid system model [15] in which the system is composed of two parts, with distinct properties and assumptions. Let us call them *payload* and *wormhole*:

Payload. *Any-synchrony subsystem* with $n \geq af + bk + 1$ replicas $P_1, \ldots, P_n$. For the purpose of our work, 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 III 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 [16], [17].

Wormhole. *Synchronous subsystem* with n local synchronous and trusted components (called local wormholes) in which at most f of these components can *fail by crash*. These components 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;
4) there is a timed atomic broadcast primitive with bounded maximum transmission time.

One should note that all of these services can be easily implemented in the crash-failure synchronous distributed system model [18], [19]. In practice, local wormholes can be small tamper-proof hardware modules (e.g., smartcards, or PC appliance boards [20]) running a real-time operating system (e.g., RTAI [21]) and connected by a switched Ethernet, which has been shown to offer real-time guarantees under controlled traffic loads [22].

B. The Proactive Resilience Model (PRM)

The PRM [6], [23] applies the system model described in Section II-A to systems enhanced with proactive recovery. Under the PRM, the proactive recovery subsystem is deployed in the wormhole part and executes the (proactive recovery) protocols that rejuvenate the applications/protocols running in the payload part. The proactive recovery subsystem 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. Local PRWs are 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 trigger, 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,

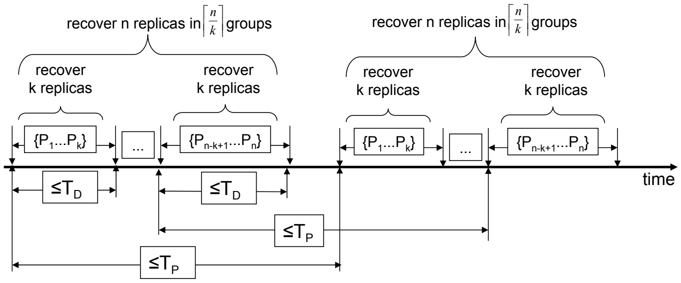


Figure 1. Relationship between the rejuvenation period T_P, the rejuvenation execution time T_D, and k.

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 simultaneously recovering servers.

Figure 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 $T_P \geq \lceil \frac{n}{k} \rceil T_D$.

C. The Proactive-Reactive Recovery Wormhole (PRRW)

The PRRW is an extension of the PRW that combines proactive and reactive recoveries. Therefore, the PRRW offers a single integrated 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 $W\_detect(j)$. Figure 2 presents this idea.

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. It should be noted 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 [24] (or a muteness detector [7]) 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 [9], [10], calling $W\_detect(j)$ in this case.

If $f+1$ different replicas suspect and/or detect that replica j is faulty, 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 faulty. 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 are 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.

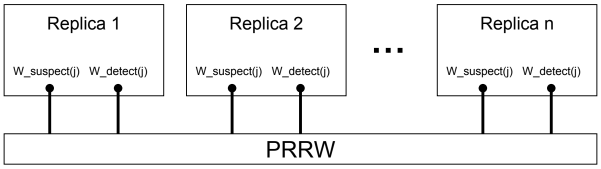


Figure 2. PRRW architecture.

As will be made clear in the next sections, we do not provide any means for a replica to "unsuspect" some other replica it previously suspected. We choose not to provide this service to avoid difficulties with the computation of the number of suspects (some replicas could see it while others not) and because suspicions are cleaned when a replica is recovered.

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) [24], [7], [9], [10], system monitoring [25] and/or intrusion detection [26], [27] 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 directly dependent on the detection/diagnosis accuracy.

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 II-B, 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 $\lceil \frac{f}{k} \rceil$ time intervals between periodic recoveries such that they can be used by event-triggered recoveries. This amount of time is allocated to allow at most f recoveries (in groups of k replicas) 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 [28]. 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:

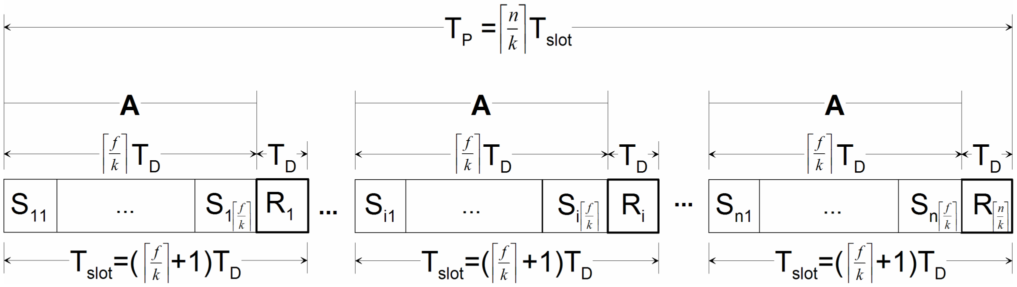

$T_P = \lceil \frac{n}{k} \rceil T_{slot}$
A
A
A
$\lceil \frac{f}{k} \rceil T_D$
T_D
$\lceil \frac{f}{k} \rceil T_D$
T_D
$\lceil \frac{f}{k} \rceil T_D$
T_D
S_{11}
...
$S_{1\lceil \frac{f}{k} \rceil}$
R_1
...
S_{i1}
...
$S_{i\lceil \frac{f}{k} \rceil}$
R_i
...
S_{n1}
...
$S_{n\lceil \frac{f}{k} \rceil}$
$R_{\lceil \frac{n}{k} \rceil}$
$T_{slot}=(\lceil \frac{f}{k} \rceil+1)T_D$
$T_{slot}=(\lceil \frac{f}{k} \rceil+1)T_D$
$T_{slot}=(\lceil \frac{f}{k} \rceil+1)T_D$

Figure 3. Recovery schedule (in an S_{ij} or R_i subslot there can be at most k parallel replica recoveries).

- 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 $(\lceil \frac{f}{k} \rceil + 1)T_D$.

Each 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_{ij}, plus a R_i subslot. Thus, every slot i has a total of $\lceil \frac{f}{k} \rceil + 1$ recovery subslots. Figure 3 illustrates how time-triggered periodic and event-triggered 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 than or equal to $\lceil \frac{n}{k} \rceil (\lceil \frac{f}{k} \rceil + 1)T_D$, 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 according to the described mechanism if replica i is only suspected of being faulty (it is not assuredly faulty), i.e., if less than $f+1$ replicas have called $W\_detect()$ (but the total number of suspicions and detections is greater than or equal to $f+1$). 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 for as one of the f faulty replicas.

2) The PRRW Algorithm: The proactive-reactive recovery service is now explained in detail. The main procedures are presented in Algorithm 1, while Algorithm 2 deals with recovery slot allocation. 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 (see Section II-A), and thus are compensated in the protocol parameters (i.e., in the time bounds for the execution of certain operations).

Parameters and variables. Algorithm 1 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 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 faulty; and *scheduled* indicates if a reactive recovery is scheduled for replica i.

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

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

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

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

{Periodic recovery — thread with priority 1}
procedure $proactive\_recovery()$
 5: $synchronize\_global\_clock()$
 6: $t_{next} \leftarrow global\_clock() + (\lceil \frac{i-1}{k} \rceil T_{slot} + \lceil \frac{f}{k} \rceil T_D)$
 7: loop
 8:     wait until $global\_clock() = t_{next}$
 9:     $recovery()$
10:     $t_{next} = t_{next} + T_P$
11: end loop
procedure $recovery()$
12: $recovery\_actions()$
13: $Detect \leftarrow \varnothing$
14: $Suspect \leftarrow \varnothing$
15: $scheduled \leftarrow false$

{Reactive recovery execution — threads with priority 2}
upon $|Detect| \geq f+1$
16: $recovery()$
upon $(|Detect| < f+1) \wedge (|Suspect \cup Detect| \geq f+1)$
17: if $\neg scheduled$ then
18:     $scheduled \leftarrow true$
19:     $\langle s, ss \rangle \leftarrow allocate\_subslot()$
20:     if $sooner(s, \lceil \frac{i}{k} \rceil)$ then
21:         wait until $global\_clock()$ mod $T_P = (s-1)T_{slot} + (ss-1)T_D$
22:         $recovery()$
23:     end if
24:     $scheduled \leftarrow false$
25: end if
```

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 faulty 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 faulty. 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 S_{ij} depicted in Figure 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 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 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 2. This algorithm uses one more parameter and one more variable besides the ones defined in Algorithm 1. The parameter T_Δ specifies the upper-bound on the delivery time of a message sent through the synchronous control network connecting all the local wormholes. Variable *Subslot* is a table that stores the number of replicas (up to k) scheduled to recover at each subslot of a recovery slot, i.e., $Subslot[\langle s, ss \rangle]$ gives the number of processes using subslot ss of slot s (for a maximum of k). This variable is used to keep the subslot occupation, allowing 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 the function *allocate_subslot*() (called in Algorithm 1, 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 (deterministic) function *local_allocate_subslot*() is called and the next available subslot is allocated to the replica

Algorithm 2 PRRW recovery slot allocation.

```
{Parameters (besides the ones defined in Algorithm 1)}
integer $T_\Delta$ {Bound on message delivery time}

{Variables (besides the ones defined in Algorithm 1)}
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: $TO\text{-}multicast(\langle \text{ALLOC}, i, global\_clock() \rangle)$
 2: wait until $TO\text{-}receive(\langle \text{ALLOC}, i, t_{send} \rangle)$
 3: return $local\_allocate\_subslot(t_{send})$
upon $TO\text{-}receive(\langle \text{ALLOC}, j, t_{send} \rangle) \wedge j \neq i$
 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: $\forall p, Subslot[\langle prev\_slot, p \rangle] \leftarrow 0$
```

(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).

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 $T_\Delta = 1$ second. 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 1 (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 $k > 1$.

Reactive recovery subslots are reused during system operation, so they need to be periodically freed. 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.

III. Case Study: The CIS Protection Service

In this section we explain how the PRRW component can be extended and integrated to make a perpetually-resilient system. The described system is a protection device for critical information infrastructures called *CIS*. In generic terms, the CIS can be seen as an improved distributed application layer firewall. Here we only present a high-level view of the system focusing in the PRRW integration. For a complete description of the CIS, see [12], [29].

A. Context and Motivation

Today's critical infrastructures like the Power Grid are essentially physical processes controlled by computers connected by networks. Once these systems were highly isolated and secure against most security threats. However, in recent years they evolved in several aspects that greatly increased their exposure to cyber-attacks coming from the Internet. Firstly, the computers, networks and protocols used in these systems are no longer proprietary and specific, but standard PCs and networks. Second, most of them are connected to the Internet and corporate networks. Therefore, these infrastructures have a level of vulnerability similar to other systems connected to the Internet, but the socio-economic impact of their failure can be huge. This scenario, reinforced by several recent incidents [30], is generating a great concern about the security of these infrastructures, especially at government level.

There was a proposal for a reference architecture to protect critical infrastructures in the context of the CRUTIAL EU-IST project [11]. 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, the 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 CIS, which is deployed at the borders of a LAN. The CIS 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 because intrusions occur with some frequency in these systems. 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, to make it withstand a high degree of hostility from the environment, seeking unattended perpetual operation. In the next sections we show how the basic intrusion-tolerant design of the CIS can be integrated with the PRRW.

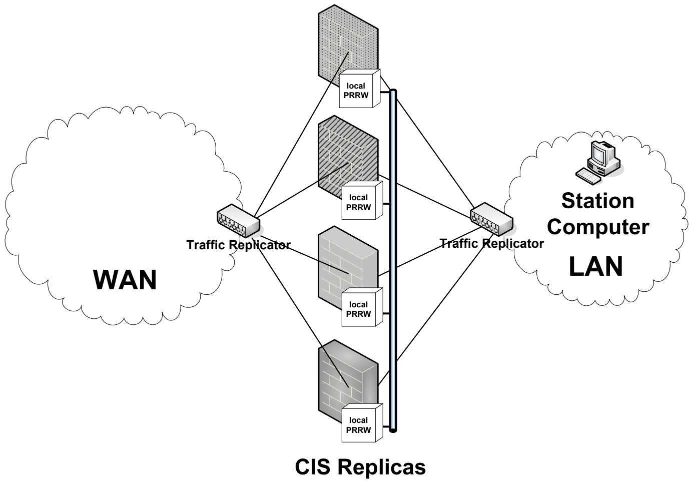


Figure 4. Intrusion-tolerant CIS protection service enhanced with the PRRW.

B. How the CIS Works

The intrusion-tolerant CIS is replicated in a set of $n \geq 2f+1$ machines[1] connected both to the protected LAN and the insecure WAN through traffic replication devices (e.g., a hub or a switch). Figure 4 depicts the CIS architecture. Note that the basic CIS requires only three replicas to tolerate one intrusion, but extra replicas are needed when the PRRW is used. Section III-C explains this in more detail.

The CIS design presents two challenges that make it essentially different from other Byzantine fault-tolerant 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 protocol itself to obtain intrusion tolerance. This also means that recipient nodes (inside the protected network) will ignore any internal CIS intrusion-tolerance mechanisms, and as such they can not protect themselves from messages (forwarded by faulty replicas) that do not satisfy the security policy. To address these two challenges, we resort to wormholes [15]: we assume that each replica of the CIS has a trusted component that cannot be corrupted (the local PRRWs), and that these components are connected through an isolated network. Moreover, each CIS replica employs diversity to avoid common mode failures, such as different operating systems (e.g., Linux, FreeBSD, Windows XP), 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 its local wormhole to be signed. The local wormholes vote approved messages between themselves and, if a message receives at least $f+1$ votes, i.e., if at least $f+1$ replicas approved the message and issued it to their local wormholes, then the message is signed using a secret key installed in the wormhole component. This secret key is known

[1] The CIS design presented here assumes that policies are stateless. In [29] we explain how stateful policies could be supported.

only by the local wormholes and by the LAN computers, i.e., it is not known by replicas[2]. To guarantee that LAN computers are able to understand which messages are signed and which are not, these signatures are in fact Message Authentication Codes (MACs) generated using the secret key and by making use of IPSEC [31], a set of standard protocols that are expected to be generalized in critical infrastructures information systems, according to best practice recommendations from expert organizations and governments [32]. Therefore, we assume that the IPSEC Authentication Header (AH) protocol [33] runs both in the LAN computers and in the local wormholes. 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 local wormholes are responsible also for leader election when the current leader becomes unavailable. The traffic replication devices in Figure 4 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 (i.e., non-signed or signed with an incorrect key) messages to the LAN.

C. Integrating the CIS and the PRRW

There are three main issues that must be addressed when integrating the PRRW into an intrusion-tolerant application: 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.

1) Extra replicas: As explained in Section II-C, 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. Figure 4 illustrates this scenario.

2) 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 requires a leader and the wormhole is responsible for maintaining it. 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.

3) 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 (Figure 4) 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[3].

Currently, 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 calls *W_detect* informing that the replica behaved in a faulty way. 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 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 in this case, therefore replicas call *W_suspect* and not *W_detect*. From Algorithms 1 and 2 it can be seen that when $f+1$ replicas *suspect* the leader, a recovery is scheduled for it.

D. Prototype

Our implementation uses the XEN virtual machine monitor [34] with the Linux operating system to isolate the PRRW from the CIS replicas. The architecture is presented in Figure 5. A XEN system has multiple layers, the lowest and most privileged of which is XEN itself. XEN may host multiple

[2] Note that it is assumed that LAN computers are part of the same trust domain, so any corrupted LAN computer (that for instance sends the secret key to the replicas) is tantamount to the whole LAN being corrupted.

[3] The substantiation of this assumption in practice will be described in Section III-D.

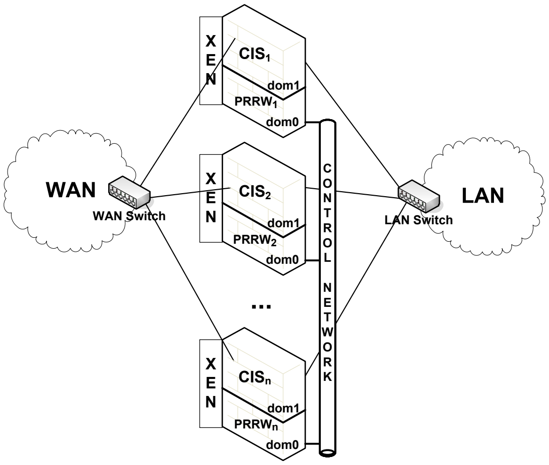


Figure 5. CIS Prototype architecture.

guest operating systems, every one executed within an isolated virtual machine (VM) or, in Xen terminology, a *domain*. Domains are scheduled by Xen to make effective use of the available physical resources (e.g., CPUs). Each guest OS manages its own applications. The first domain, *dom0*, is created automatically when the system boots and has special management privileges. Domain *dom0* builds other domains (*dom1, dom2, dom3, ...*) and manages their virtual devices. 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.

As depicted in Figure 5, every replica uses Xen to isolate the payload from the PRRW part. Local PRRWs 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 the PRRW service. The local PRRWs are connected through an isolated control network.

The recovery actions executed by the PRRW make use of Xen (command line) system calls *xm destroy* and *xm create* to, respectively, shutdown and boot a CIS replica. Note that *xm destroy* corresponds to a virtual power off and it is almost instantaneous, whereas a normal shutdown could take several seconds. We avoid the delay of a normal shutdown because we assume that the running OS may have been compromised and thus it cannot be reutilized.

In order to provide a virtual global clock to each local PRRW, we implemented a clock synchronization protocol. There are many clock synchronization algorithms available in the literature suitable to synchronous environments with crash faults [19]. The protocol we implemented combines techniques proposed on some of these works and it is now briefly explained. Local PRRWs are initially launched in any order as long as local PRRW 1 is the last one to begin execution. When local PRRW 1 starts executing, it broadcasts a small synchronization message to all PRRWs (including itself) and this message is used by every local PRRW to define the global time instant 0. Then, in order to maintain the virtual global clocks with a bounded precision, each local PRRW broadcasts a synchronization message exactly when a proactive recovery is triggered. Given that all local PRRWs know when proactive recoveries should be triggered, they can adjust their clocks accordingly. Both mechanisms assume that the broadcast reception instant is practically the same everywhere in the control network, which is substantiated by the fact that the control network is provided by a switch used only by the local PRRWs.

To simplify the design and to avoid changes in the kernel, the CIS prototype operates at the UDP level, instead of IP level as most firewalls do. Therefore, there was no need to implement packet interception because packets are sent directly to the CIS. Moreover, authentication is not done at the IP level (as when using IPSec/AH), but instead the PRRW calculates the HMAC[4] of the payload UDP packet, and then the two are concatenated. Notice that this type of authentication implies the same type of overhead of IP authentication. Given that the CIS prototype operates at the UDP level, we employ IP multicast to enforce broadcast in WAN and LAN communication. Therefore, we do not need physical traffic replication devices in our prototype[5]. Moreover, the LAN switch uses access control lists to prevent replicas from spoofing their MAC addresses, and each replica stores a table with the MAC address of each other replica.

Periodic and reactive recoveries reset the state and restore the code of a replica. While this is useful to restore the correctness of the replica, it would be interesting if we were able to introduce diversity in the recovery process. For instance, each recovery could randomize the address space of the replica (e.g., using PaX [36]) in order to minimize the usefulness of the knowledge obtained in the past to increase the chances of future attacks. Although it was not possible to integrate Xen with an address space layout randomization mechanism such as PaX, the prototype incorporates some diversity mechanisms: we maintain a pool of OS images with different configurations (e.g., different root passwords, different kernel versions) and each recovery (proactive or reactive) randomly chooses and boots one of these images.

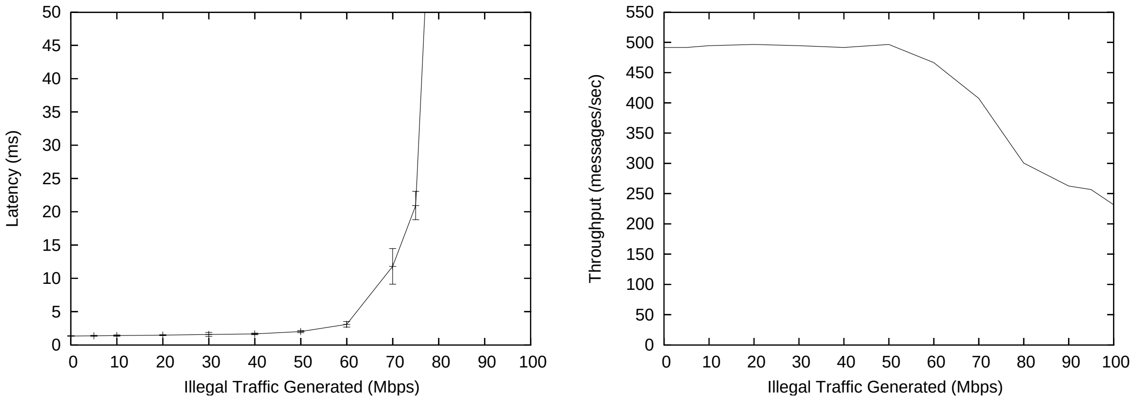
Domain *dom*0 executes with higher priority than domain *dom*1, but since it is based on a normal Linux OS, it provides no strict real-time guarantees. Currently, only modified versions of Linux, NetBSD, OpenBSD and Solaris can be run on *dom*0.

IV. Experimental Evaluation

The experimental setup was composed by a set of four machines representing the CIS replicas ($n = 4$, $f = 1$, $k = 1$) connected to the three networks defined in our prototype architecture: LAN, WAN, and the control network (see Figure 5). These networks were defined as separated VLANs configured on two Dell Gigabit switches. The LAN and control networks shared the same switch (using different VLANs), whereas the WAN network was deployed in a different switch. The LAN

[4] HMAC is a standard for calculating MACs, and in the prototype we used the SHA-1 hash function [35].

[5] Nevertheless, if the CIS would operate at the IP level, we could configure the WAN and LAN switches to use the failopen mode in order to force broadcast.



(a) Average latency.

(b) Maximum throughput.

Figure 6. Latency and throughput of the CIS in forwarding legal messages sent by a good sender in the WAN to the station computer in the LAN while a malicious host in the WAN is sending illegal traffic (not approved by the CIS).

and WAN were configured as 100 Mbps networks while the control network operated at 1 Gbps. We used three additional PCs 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). Every machine of our setup was a 2.8 GHz Pentium 4 PC with 2 GB RAM running Fedora Core 6 with Linux 2.6.18, and XEN 3.0.3 to manage the virtual machines. As explained in Section III-D, each CIS physical host uses the XEN virtual machine monitor to manage two virtual machines: a non-privileged one with 1536 MB of RAM (*dom1* – CIS protection service) and a trusted one with 512 MB of RAM (*dom0* – local PRRW).

A. *Performance of recoveries*

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 I 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. All disk images used in this experiment had sizes of approximately 1.7 GB.

	Shutdown	**Rejuvenation**	**Reboot**	**Total**
Average	0.6	72.2	70.1	144.6
Std. Deviation	0.5	1.2	0.3	0.9
Maximum	1.0	74.0	71.0	146.0

Table I
TIME NEEDED (IN SECONDS) FOR THE SEVERAL STEPS OF A REPLICA RECOVERY (1.7 GB OS IMAGES).

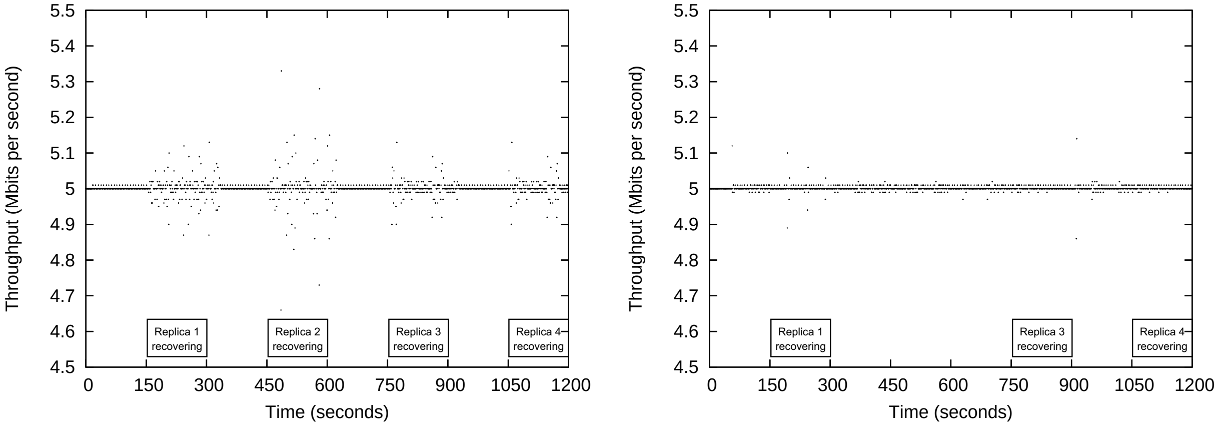
From Table I one can see that a maximum of 146 seconds (~ 2.5 minutes) are needed in order to completely recover a virtual machine in our environment, most of this time being spent on two tasks: (1.) copying a clean pre-configured disk image from a local repository; and (2.) starting this new image (including starting the CIS protection service). These tasks could have their time lowered if we were able to build smaller images, which was not possible with the current Linux distribution we are using (Fedora Core 6).

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 II-C1 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 Proactive Resilience model [6], [23], 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.

B. *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 byte's 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. Note that these experiments make a pessimistic assumption about the rate of legal traffic, given that 500 packets/second is a very high traffic for a critical infrastructure information system[6]. 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 CIS message approval also in the presence of DoS attacks of

[6] It may represent, for instance, 500 MMS (Manufacturing Message Specification) commands being sent to a device per second.



(a) No faults.

(b) One faulty replica (replica 2).

Figure 7. Throughput of the CIS during a complete recovery period (20 minutes) with $n = 4$ ($f = 1$ and $k = 1$), with and without crash faults.

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 6 shows the average latency and maximum throughput measured in these experiments.

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 6(a)) and the throughput drops to about 250 messages/sec (Figure 6(b)). For 90-100 Mbps of invalid traffic we observed that sometimes (in 1% of the experiments) the CIS loses some messages (from 15% to 21%), however, it occurs rarely. These results suggest that our design adds modest latency (less than 2 ms) and no throughput loss even with a reasonably loaded network. The results show also that to cope with significant DoS attacks (> 70 Mbps) coming from the unprotected network, complimentary mechanisms must be employed, given that CIS processing latency has a huge increase.

In the third and fourth set of experiments described next, the goal was to evaluate the impact of proactive/reactive recoveries and replicas' crash/Byzantine faults in the overall system throughput. In these experiments there is not a malicious sender in the WAN, just a good sender constantly transmitting legal traffic at a rate of 5 Mbps to the station computer.

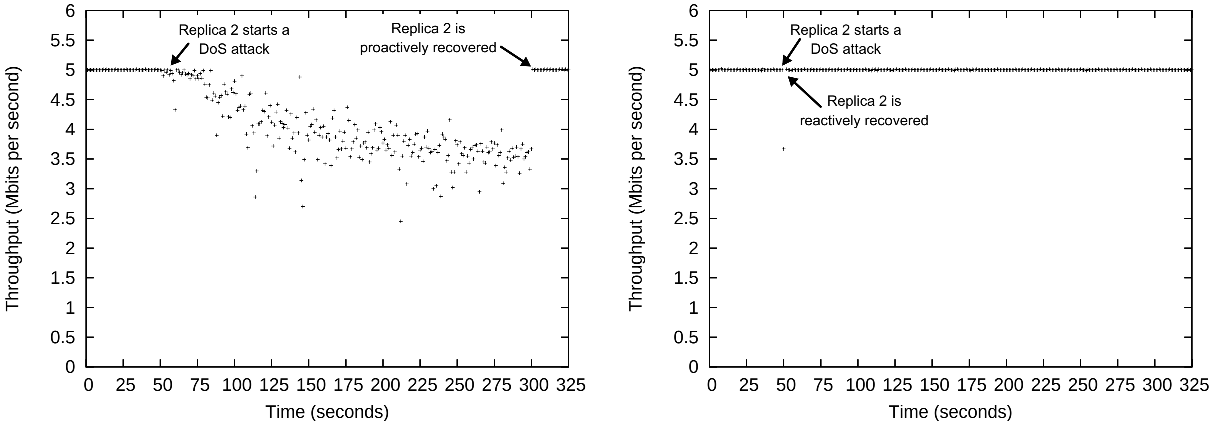
C. Throughput during recoveries and in the presence of crash faults

The third set of experiments evaluated the impact of proactive recovery and crash faults in the overall system throughput. Figure 7 presents two time diagrams that show the throughput of the CIS during a complete recovery period (20 minutes) without faults and with one crashed (silent) replica. In the latter experiment, replica 2 is the one crashed, and it stays crashed until the end of the recovery period.

The time diagrams of Figure 7 lead to the following conclusions. First, it can be seen that, without faults, recoveries do not have a substantial impact on the perceived throughput of the system (Figure 7(a)). The minimum observed throughput during recoveries was 4.6 Mbps, which represents a 8% drop in comparison with the expected throughput of 5 Mbps. This can be explained by the fact that proactive recoveries are executed with higher priority than voting procedures and thus may delay their execution during the recovering periods. Note also that the few cases where the throughput increases are a side-effect of the decrease of throughput. When the throughout decreases, replicas' buffers accumulate more messages, which results in a higher processing rate (than the 5 Mbps send rate) during some instants. Second, the occurrence of crash faults also does not affect substantially the throughput of the system, even during periodic recoveries (Figure 7(b)). This happens because we use $k = 1$ extra replicas to ensure that the system is always available: even with one fault and one recovery, there are still two correct replicas ($f + 1$) to vote and approve messages. Note that without these k extra replicas, the system would become completely unavailable in the recovering periods of Figure 7(b). Third, by comparing the observed throughput with and without crash faults, one can conclude that the impact of proactive recoveries is smaller when there is a crashed replica during the entire execution. This happens because 3 replicas generate less vote messages in the wormhole control channel than 4 replicas. This reduction is sufficient to minimize the impact of the higher priority proactive recoveries on the vote processing time.

D. Throughput under a DoS attack from a compromised replica

Finally, the fourth 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 algorithm's 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



(a) With proactive recovery only.

(b) With proactive and reactive recovery.

Figure 8. Throughput of the CIS during 325 seconds with $n = 4$ ($f = 1$ and $k = 1$). Replica 2 is malicious and launches a DoS attack to the LAN after 50 seconds. We present graphs without (a) and with (b) reactive recovery.

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 recovery of the replica. 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 8.

Figure 8(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 3 and proactive recoveries may be triggered one after the other.

The utility and effectiveness of combining proactive and reactive recoveries is illustrated by Figure 8(b), which shows that the CIS throughput is minimally affected by the DoS attack from replica 2. This attack is detected by the remaining replicas given that invalid traffic is being sent to the LAN. In consequence, they call the PRRW interface function *W_detect*() passing as argument the id of replica 2 and local PRRW 2 receives enough DETECT messages to immediately trigger a reactive recovery (see Algorithm 1). 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.

V. Related Work

Rejuvenation [37], [38] has been proposed in the 90s as a proactive technique to deal with transient failures due to software aging. The idea is to periodically rejuvenate some software components to eliminate and prevent failures. Some research on this field, beginning with [38], developed techniques to choose the optimal rejuvenation period in order to minimize the downtime of the system. None of the works on software rejuvenation assume faults caused by malicious adversaries, so the problem that we deal in this work is probably harder than software aging.

Several works advocate the use of proactive recovery to make the system tolerate an unbounded number of malicious faults during its lifetime as long as no more than f faults occur during a bounded time period [3], [4], [5]. In previous works, some authors showed that all these works have some hidden problems that could be exploited by an adversary [14]. The main problem is that these works assume the asynchronous distributed system model and, under this model it is not possible to guarantee that recoveries are triggered and executed within known time bounds: an adversary can delay the execution of the recoveries and be able to corrupt more than f nodes of a system [13]. The work presented in this paper is based on a hybrid distributed system model and thus uses some trusted and timely components to ensure that replicas are always rejuvenated in accordance to predefined time bounds [6], [23].

There is a large body of research that aims to reactively recover systems as fast and efficiently as possible (e.g., [39], [40], [41]). Early work on this field (e.g., [39]) advocates the execution of some kind of recovery action (in general, restart the monitored process) after a fault is detected. More recently, the recovery oriented computing project proposed a set of methods to make recovery actions more efficient and less costly in terms of time [40]. Several techniques were developed in this project, either to detect failures, restart the minimum set of system components to put the system back to correct operation and to undo some operator configuration errors. Other works like [41] try to diagnose system faults through several monitors and evaluate which is the best set of recovery actions that must be taken in order to recover the system as fast as possible. These works do not consider Byzantine faults or security-compromised components and also do not rely on redundancy to ensure that the system stays available during recoveries, the main problems addressed in the present work.

The second half of the paper presents a case study for the PRRW service: an intrusion-tolerant firewall designed to protect critical infrastructures. As far as we know, there is only one other work about intrusion-tolerant firewalls: the privacy firewall [42]. This work presents an architecture for maintaining the privacy of a specific replicated state machine system [43], [3]: each message from the replicas is forwarded to the clients if and only if more than f replicas issue this message. In this way, a faulty replica cannot disclose private information to external processes. The aim of the CIS is completely different to that of the privacy firewall since it is a general intrusion-tolerant firewall that implements a protection system transparent and independent of the application messages that pass through it.

VI. 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 the faulty behavior 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. 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 of the CIS, namely in the presence of powerful DoS attacks launched either by outside hosts or inside compromised replicas.

Acknowledgements

We thank the anonymous referees for their helpful comments. This work was partially supported by the EC through project IST-2004-27513 (CRUTIAL), and by the FCT, through the Multiannual Funding and the CMU-Portugal Programs.

References

[1] P. Verissimo, N. F. Neves, and M. P. Correia, "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems*, ser. LNCS, 2003, vol. 2677.

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

[3] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM Tras. on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[4] L. Zhou, F. Schneider, and R. Van Rennesse, "COCA: A secure distributed online certification authority," *ACM Trans. on Computer Systems*, vol. 20, no. 4, pp. 329–368, Nov. 2002.

[5] M. A. Marsh and F. B. Schneider, "CODEX: A robust and secure secret distribution system," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 34–47, Jan. 2004.

[6] P. Sousa, N. F. Neves, and P. Verissimo, "Proactive resilience through architectural hybridization," in *Proc. of the 2006 ACM Symp. on Applied Computing (SAC'06)*, Apr. 2006, pp. 686–690.

[7] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness failure detectors: Specification and implementation," in *Proc. of the 3rd European Dependable Computing Conference*, Sep. 1999, pp. 71–87.

[8] A. Doudou, B. Garbinato, and R. Guerraoui, "Encapsulating failure detection: From crash to byzantine failures," in *da-Europe '02: Proc. of the 7th Ada-Europe Int. Conference on Reliable Software Technologies*. London, UK: Springer-Verlag, 2002, pp. 24–50.

[9] R. Baldoni, J.-M. Hélary, M. Raynal, and L. Tangui, "Consensus in Byzantine asynchronous systems," *J. Discrete Algorithms*, vol. 1, no. 2, pp. 185–210, Apr. 2003.

[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] P. Verissimo, N. F. Neves, and M. Correia, "CRUTIAL: The blueprint of a reference critical information infrastructure architecture," *Int. Journal of System of Systems Engineering*, vol. 1, no. 1/2, pp. 78–95, 2008.

[12] A. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Verissimo, "The CRUTIAL way of critical infrastructure protection," *IEEE Security & Privacy*, vol. 6, no. 6, pp. 44–51, Nov-Dec 2008.

[13] 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'05)*, Jun. 2005, pp. 98–107.

[14] ——, "Hidden problems of asynchronous proactive recovery," in *Proc. of the Workshop on Hot Topics in System Dependability*, June 2007.

[15] P. Verissimo, "Travelling through wormholes: a new look at distributed systems models," *SIGACT News*, vol. 37, no. 1, 2006. [Online]. Available: http://www.navigators.di.fc.ul.pt/docs/abstracts/ver06travel.html

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

[17] A. N. Bessani, R. R. Obelheiro, P. Sousa, and I. Gashi, "On the effects of diversity on intrusion tolerance," Department of Informatics, University of Lisbon, DI/FCUL TR 08–30, December 2008.

[18] V. Hadzilacos and S. Toueg, "A modular approach to the specification and implementation of fault-tolerant broadcasts," Dep. of Computer Science, Cornell Univ., New York, USA, Tech. Rep. 94-1425, May 1994.

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

[20] S. Kent, "Protecting externally supplied software in small computers," Ph.D. dissertation, Laboratory of Computer Science, Massachusetts Institute of Technology, 1980.

[21] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour, "DIAPM-RTAI position paper," in *Real-Time Linux Workshop*, Nov. 2000.

[22] A. Casimiro, P. Martins, and P. Verissimo, "How to build a timely computing base using real-time linux," in *Proc. of the 2000 IEEE Int. Workshop on Factory Communication Systems*. Porto, Portugal: IEEE Industrial Electronics Society, Sep. 2000, pp. 127–134.

[23] P. Sousa, N. F. Neves, P. Verissimo, and W. H. Sanders, "Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available," in *Proc. of the 25th IEEE Symp. on Reliable Distributed Systems (SRDS'06)*, Oct. 2006, pp. 71–80.

[24] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, Mar. 1996.

[25] A. Daidone, F. Di Giandomenico, A. Bondavalli, and S. Chiaradonna, "Hidden Markov models as a support for diagnosis: Formalization of the problem and synthesis of the solution," in *Proc. of the 25th IEEE Symp. on Reliable Distributed Systems (SRDS'06)*, Leeds, UK, October 2006, pp. 245–256.

[26] D. E. Denning, "An intrusion-detection model," *IEEE TSE*, vol. 13, no. 2, pp. 222–232, 1987.

[27] B. Mukherjee, L. Heberlein, and K. Levitt, "Network intrusion detection," *IEEE Network*, vol. 8, no. 3, pp. 26–41, 1994.

[28] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, 1989.

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

[30] C. Wilson, "Terrorist capabilities for cyber-attack," in *Int. CIIP Handbook 2006*, M. Dunn and V. Mauer, Eds. CSS, ETH Zurich, 2006, vol. II, pp. 69–88.

[31] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4301.txt

[32] K. Stouffer, J. Falco, and K. Kent, "Guide to Supervisory Control And Data Aquisition (SCADA) and industrial control systems security,"

Recommendations of the National Institute of Standards and Technology (NIST). Special Publication 800-82 (Initial Public Draft), Sep. 2006.

[33] S. Kent, "IP Authentication Header," RFC 4302 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4302.txt

[34] 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.

[35] National Institute of Standards and Technology, "Secure Hash Standard," Federal Information Processing Standards Publication 180-2, Aug. 2002.

[36] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. of the 11th ACM Conference on Computer and Communications Security*, 2004, pp. 298–307.

[37] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. of 25th Int. Symp. on Fault Tolerant Computing (FTCS-25)*, Jun. 1995, pp. 381–390.

[38] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Analysis of software rejuvenation using markov regenerative stochastic petri nets," in *Proc. of Int. Symp. on Software Reliability Engineering (ISSRE'95)*, Oct. 1995.

[39] Y. Huang and C. M. R. Kintala, "Software implemented fault tolerance: Technologies and experience," in *Proc. of 23rd Int. Symp. on Fault Tolerant Computing (FTCS-23)*, Jun. 1993, pp. 2–9.

[40] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery oriented computing (ROC): Motivation, definition, techniques and case studies," Computer Science Dep., Univ. of California at Berkeley, UCB/CSD TR 02–1175, Mar. 2002.

[41] K. R. Joshi, M. Hiltunen, W. H. Sanders, and R. Schlichting, "Automatic model-driven recovery in distributed systems," in *Proc. of the 24th IEEE Symp. on Reliable Distributed Systems (SRDS'05)*, Oct. 2005, pp. 26–38.

[42] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement form execution for Byzantine fault tolerant services," in *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP'03)*, 2003, pp. 253–267.

[43] F. B. Schneider, "Implementing fault-tolerant services using the state machine aproach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

Paulo Sousa is Invited Assistant Professor of the Department of Informatics, University of Lisboa Faculty of Sciences. He received a PhD in Computer Science from the University of Lisboa in 2007 and a Licenciatura in Computer Science from the same university in 2001. He was ranked best Computer Science student in 2000 and 2001 and his PhD work was awarded with the IBM Scientific Prize in 2007. Paulo Sousa is a member of the LaSIGE laboratory and the Navigators research group. He has been involved in several international and national research projects related to real-time, intrusion tolerance and security. He was the coordinator of the FOREVER research project funded by the EU through the ReSIST NoE. His main research interests include: intrusion tolerance, resilience, software security, distributed systems. More information about him is available at http://www.di.fc.ul.pt/∼pjsousa.

Alysson Neves Bessani is a visiting assistant professor of the Department of Informatics of the University of Lisboa Faculty of Sciences, Portugal, and a member of LASIGE research unit and the Navigators research team. He received his B.S. degree in Computer Science from Maringá State University, Brazil in 2001, the MSE in Electrical Engineering from Santa Catarina Federal University (UFSC), Brazil in 2002 and the PhD in Electrical Engineering from the same university in 2006. His main interests are distributed algorithms, Byzantine fault tolerance, coordination, middleware and systems architecture. More information about him is available at http://www.di.fc.ul.pt/∼bessani.

Miguel Correia is Assistant Professor of the Department of Informatics, University of Lisboa Faculty of Sciences, and Adjunct Faculty of the Carnegie Mellon Information Networking Institute. He is a member of the LASIGE research unit and the Navigators research team. He has been involved in several international and national research projects related to intrusion tolerance and security, including the MAFTIA and CRUTIAL EC-IST projects, and the ReSIST NoE. He is currently the coordinator of University of Lisboa's degree on Informatics Engineering and an instructor at the joint Carnegie Mellon University and University of Lisboa MSc in Information Security. His main research interests are: intrusion tolerance, security, distributed systems, distributed algorithms. More information about him is available at http://www.di.fc.ul.pt/∼mpc

Nuno Ferreira Neves is Associate Professor of Computer Science at the University of Lisbon since 1998. He received a Ph.D in Computer Science from the University of Illinois at Urbana-Champaign (1998), and Licenciatura and Masters from the Technical University of Lisbon (1992 and 1995). His research interests are in parallel and distributed systems, in particular in the areas of dependability and security. His work has been recognized with the IBM Scientific Prize in 2004 and the William C. Carter award at IEEE International Fault-Tolerant Computing Symposium in 1998. In the recent years, he has served in program committees of conferences in the area of dependable computing, and participated in several European and National research projects which had a focus on security. More information about him is available at http://www.di.fc.ul.pt/∼nuno.

Paulo Verissimo is currently a professor of the Department of Informatics (DI) of the University of Lisbon Faculty of Sciences, and Director of LASIGE, a research laboratory of the DI. He belongs to the European Security & Dependability Advisory Board, and is associate editor of the IEEE Transactions on Dependable and Secure Computing. He is past Chair of the IEEE Technical Committee on Fault Tolerant Computing and of the Steering Committee of the DSN conference, and belonged to the Executive Board of the CaberNet European Network of Excellence. He was coordinator of the CORTEX IST/FET project. He is senior member of the IEEE. Paulo Verissimo leads the Navigators research group of LASIGE, and is currently interested in: architecture, middleware and protocols for distributed, pervasive and embedded systems, in the facets of real-time adaptability and fault/intrusion tolerance. He is author of more than 130 refereed publications in international scientific conferences and journals in the area, and co-author of five books (ex. Distributed Systems For System Architects). More information about him is available at http://www.di.fc.ul.pt/∼pjv.