Proactive Resilience Revisited: The Delicate Balance Between Resisting Intrusions and Remaining Available[*]

Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo
Univ. of Lisboa
{pjsousa,nuno,pjv}@di.fc.ul.pt

William H. Sanders
Univ. of Illinois at Urbana-Champaign
whs@uiuc.edu

Abstract

In a recent paper, we presented proactive resilience as a new approach to proactive recovery, based on architectural hybridization. We showed that, with appropriate assumptions about fault rate, proactive resilience makes it possible to build distributed intrusion-tolerant systems guaranteed not to suffer more than the assumed number of faults during their lifetime. In this paper, we explore the impact of these assumptions in asynchronous systems, and derive conditions that should be met by practical systems in order to guarantee long-lived, i.e., available, intrusion-tolerant operation. Our conclusions are based on analytical and simulation results as implemented in Möbius, and we use the same modeling environment to show that our approach offers higher resilience in comparison with other proactive intrusion-tolerant system models.

1 Introduction

Malicious attacks are an increasing problem in providing Internet services. These attacks may compromise the correctness of a service by accessing or changing its state, or may simply block service output, which is typically characterized as a Denial-of-Service attack (DoS). Depending on the service domain, a successful attack may have harmful or even catastrophic consequences, since the use of on-line services to perform critical services has grown significantly over the past few years. On the one hand, the percentage of services with societal impact being deployed mainly or exclusively on-line is increasing (e.g., e-commerce, e-health, and e-government). On the other hand, a number of services provided by infrastructures that are inherently critical are increasingly being deployed in a remote manner, sometimes through the Internet (e.g., electrical, water, and gas facilities). Given this dependence, the goal is to deploy system architectures that are able to resist intrusions while maintaining availability. Although in theory these are not conflicting goals, in reality, many current approaches to resisting intruders affect availability, and intrusion-tolerance mechanisms are haunted by the problem of exhaustion of resources.

Building intrusion-tolerant services is a difficult task, given that one must defend against an adversary who is acting both unpredictably and intentionally. This is in contrast with the traditional scenario, in which services were only at the mercy of accidental faults, a predictable (to a certain extent) adversary. For this reason, it is typically assumed that faults can be of an arbitrary nature. However, if the system is to keep working throughout its mission time, then one should also ensure that the assumed number of faults is never exceeded during that interval. In consequence, if the maximum execution time is not known, or if it is known but the fault rate is excessively high, a recovery mechanism must be added to the system. Then the goal is to guarantee that the recovery rate is greater than the fault rate, so that, again, the assumption on the maximum number of faults is never violated.

It is easy to see that Internet services typically fit the scenario in which execution time is unknown: their goal is to be constantly available, and usually a mission time does not exist. Thus, recovery becomes a requirement. Recovery can be done reactively, proactively, or through a combination of both. Proactive recovery is of particular interest for many environments (including Internet services) in which it is not always easy to diagnose faults [9]. In theory, systems that provide proactive recovery tolerate any number of faults during the lifetime of the system, as long as no more than f faults occur during a window of vulnerability defined by the interval between two consecutive recoveries.

The following four problems may affect intrusion-tolerant systems that employ proactive recovery: (1) a malicious adversary may deploy more power than originally assumed, and corrupt nodes at a pace faster than recovery; (2) he or she may attempt to slow down the pace of recovery, in order to leverage the chances of intruding the system with the available power; (3) he or she may perform stealth attacks on the system timing, which in asynchronous[1] or

[*]This work was partially supported by the EC through project IST-2004-27513 (CRUTIAL) and by the FCT through the Large-Scale Informatic Systems Laboratory (LaSIGE) and project POSC/EIA/60334/2004 (RITAS). This material is also based upon work supported by the National Science Foundation under Grant No. CNS-0406351.

[1]Notice that it is possible to do attacks on the timing of asynchronous systems, that is, on the way they make progress according to real time.

partially synchronous systems may not even be perceived by the essentially time-free logic of the system, leaving it defenseless; and (4) recovery procedures may make it necessary to bring individual nodes to a temporarily inactive state, lowering the redundancy quorum and thus system resilience.

The first problem (violation of attacker power assumptions) is outside the scope of this paper, and it is fundamentally an unsolvable problem, since those assumptions are at the heart of the intrusion-tolerance body of research. It must be addressed with techniques that mitigate any leverage an attacker may unexpectedly try to get, such as diversity, mutation, obfuscation, or trusted components, which can complement the approach we describe here. In this paper, it is shown how an architecture and generic algorithmic approach to proactive recovery, globally designated *proactive resilience*, addresses each of the remaining problems. In the conditions above, our design methodology allows us to build resilient intrusion-tolerant services that never suffer more than the assumed number of faults and are always available. To our knowledge, this is the first time that solutions to these problems have been presented and analyzed, and the elimination of these problems drastically reduces the state-space of the attacker, as compared with previous work. In consequence, our results may have importance in generic on-line services, and even more in critical infrastructure settings.

Our conclusions are based on analytical calculations and on simulation results obtained from the Möbius modeling tool [5]. Both use a refinement of the exhaustion-safety definition presented in [11] with a much more detailed model of attacks and recoveries. The simulation model also incorporates a technique that allows the system to live under two different timebases: one representing the pace of generation of faults and/or attacks, and another representing the internal execution, e.g., of recoveries. The former largely depends on physical events happening in real time (such as accidental fault generation and hacker attacks) and thus should be modeled as having a synchronous behavior, whereas the latter depends on the internal system synchrony assumptions. The innovation in this separation is more important than meets the eye, for at the root of our initial findings [11] was the discovery that the asynchronous system models used so far did not depict accurately enough the subtle timing relations between the pace at which faults occur and the pace at which a system executes, leading to unexpected failures by exhaustion of system resources. For instance, a simple attack on a node's clock drift rate may slow down the rate of recoveries, and thus increase the probability that another type of faults (potentially more dangerous) will be successful. Notice that such a (timing) attack is undetectable under asynchronous assumptions.

In summary, this paper makes the following original contributions:

1. It enumerates and discusses four problems that may affect intrusion-tolerant systems employing proactive recovery;

2. It shows, analytically and through simulation, how an architecture and generic algorithmic approach to proactive recovery, designated *proactive resilience*, addresses the three solvable problems pointed out in item 1;

3. A collateral contribution of item 2 is a study (the first, to the best of our knowledge) of recovery strategies with the goal of simultaneously assuring intrusion tolerance and availability; and

4. The simulation model used in item 2 incorporates a novel technique that allows the system to live under two different timebases: one representing the pace of generation of faults and/or attacks, and another representing the pace of internal execution. Experiments with arbitrary correlations and interleaving of these timebases can easily be defined.

2 Related Work

Intrusion tolerance is often built using agreement and/or replication techniques that tolerate Byzantine faults. Typically, these techniques make the assumption that the number of faults is bounded by a known value [1, 3, 6, 8]. Although this assumption is necessary in the context of an abstract algorithm design, special attention should be given when the same algorithm is used to implement a system that is supposed to resist a malicious adversary. The system architect must guarantee that, in the worst case, the fault rate is such that the assumed number of faults is never violated during the lifetime of the system.

Reactive recovery works under the assumption that malicious behavior can be detected before any crucial property of the service is violated, such as the assumed number of faults. The coverage of such an assumption depends on the nature of the service (which may force the adversary to reveal himself before doing really harmful things) and on the power of the adversary. Consider, for example, an intrusion-tolerant service able to resist a maximum number f of node failures. An adversary may silently compromise $f + 1$ nodes without doing any action that triggers detection and subsequent recovery, and, after contaminating those nodes, launch the final attack.

In order to survive attacks that circumvent detection mechanisms, the system architect must enhance the system with purposely-triggered recoveries, an approach known as *proactive recovery* [9]. Recent work uses the proactive recovery approach in order to tolerate more than the assumed number f of malicious faults [4, 14, 2, 7]. Given a window of vulnerability defined by the interval between two consecutive recoveries, if no more than f faults ever occur during such an interval, then the system tolerates any number of faults during its lifetime.

The approach described and analyzed in this paper is

The difference between synchronous and asynchronous systems is that the safety of protocols running in the former may depend on timing guarantees, whereas it should not for protocols running in the latter.

comparatively more resilient, since it eliminates the chances of successful attacks, conspicuous or stealth, on the timing of the system's recovery mechanisms; enforces timeliness of system recovery even in asynchronous settings; and presents recovery strategies that enforce resilience to intrusions whilst maintaining availability.

This last point is especially relevant, and has not been studied in depth. Depending on the number of spares or replicas, when recovery occurs, the redundancy quorum may lower to the point that the system is either temporarily "non-intrusion-tolerant" or temporarily "non-available." For example, if four replicas are needed to resist one fault, a system with four nodes becomes fragile each time it recovers one replica, if its service remains available. Otherwise, if the service is suspended during the recovery process, it becomes temporarily unavailable (i.e., the quorum $n \geq 3f + 1$ is no longer enforced during the recovery interval). To our knowledge, this is the first study of recovery strategies with both goals in mind.

3 Node-Exhaustion-Safety

Our approach is now described in detail. In a recent paper [12], *proactive resilience* was presented as a new approach to proactive recovery, based on architectural hybridization. It was shown that, for a given fault rate, proactive resilience makes it possible to build distributed intrusion-tolerant systems guaranteed not to suffer more than the assumed number of faults, a predicate that was named *exhaustion-safety*. In this paper, we explore the limits of proactive resilience in practical systems, and derive precise conditions that should be met in order to guarantee both resilience to intrusions and availability. In consequence of our findings, we present a design methodology that makes it possible to build resilient intrusion-tolerant services that are always available.

Given that it is impossible to foresee what a malicious adversary will do during service execution, the system architect should make the weakest set of assumptions possible. If an arbitrary behavior by the adversary is assumed, this means a qualitatively worst-case scenario. Still, a door remains open with regard to the quantitative aspects of the adversary action: *How fast can system components be attacked, so that its resources become exhausted?*

Recently, a novel theoretical distributed systems model was presented that takes into account the qualitative as well as the quantitative aspects of fault production [11]. That is, the model considers not only the type of faults assumed (e.g., omission, Byzantine), but also the number of faults is depicted as a function of time, as they affect resources (e.g., replicas) along the timeline of system execution: this is in contrast to the usual stationary f-fault quantifier. This model allows us to represent the concept of *resource exhaustion*, the situation when the system no longer has the necessary resources to execute correctly (e.g., the required number of replicas). In consequence, *exhaustion-safety* was introduced as a new predicate that should be considered when designing a fault-tolerant distributed system: a system is exhaustion-safe if it is assured that no resource exhaustion occurs during any execution [11]. Therefore, exhaustion-safety is a generic, application-independent predicate, which determines the actual resilience of a system in a more precise way than allowed by previous models.

In this section we revisit the exhaustion-safety definition, and focus on fault-tolerant distributed systems and on a specific resource: the system nodes. Typically, a fault-tolerant distributed system with a number n of nodes is able to resist a specified maximum number f of node failures. In other words, that system needs at least $n-f$ correct nodes in order to guarantee that its specification is satisfied. Thus, we say that a distributed system is *node-exhaustion-safe* when it is guaranteed that no more than f node failures ever occur during any of its executions. The difficult task here is to estimate, during system design, the appropriate number f (and thus n) necessary to achieve node-exhaustion-safety. This estimation depends on two factors, namely the power of the adversaries and the power of system defenses. Intuitively, the difference between these two opposite forces should never be such that $f + 1$ nodes may be compromised at any time.

We model a distributed system adversary through a parameter: the minimum inter-failure time, $mift$, which measures the speed at which the adversary is capable of attacking and causing individual node failures. We want to keep the model simple enough, so for the purpose of the paper we assume that whenever there are dependencies between node failures creating common failure modes (e.g., two nodes using the same operating system and thus being vulnerable to the same type of attacks on the OS vulnerabilities), this can be absorbed by a smaller $mift$. Despite this simplification, the model is sufficiently representative to discuss the problems presented in the introduction.

We model the distributed system itself through three parameters: the maximum execution time, met; the maximum inter-recovery time, $mirt$; and the maximum recovery degree, mrd. The first parameter represents the maximum duration of a meaningful execution (e.g., protocol run, transaction, or server action). The last two parameters typify system recoveries. A recovery execution can take several steps and recovers all the nodes of the system; hence, regardless of whether they failed before recovery, all the nodes become correct after a recovery. $mirt$ specifies the maximum interval between the triggering of a recovery procedure and the termination of the next consecutive one, whereas mrd specifies the maximum number of nodes that are recovered simultaneously at any recovery step (i.e., a recovery procedure is composed of at least $\lceil \frac{n}{mrd} \rceil$ recovery steps, where n is the total number of nodes). Notice that during a recovery step, the nodes in question are considered to have failed. Therefore, the system must have extra redundancy if it is to continue operating correctly through recoveries, one of our objectives. The discussion of how the state of the nodes is affected by a recovery is out of the scope of this paper. In fact, this is an advantage: our model is sufficiently generic that it can be applied to either stateful or stateless recovery.

If the system maximum execution time met is known, then recoveries are not necessary as long as, given f and worst-case the fault period $mift$, the system avoids exhaustion during execution. It is trivial to see that the following condition should be satisfied: the system should be resourced so as to tolerate $f \geq \lceil \frac{met}{mift} \rceil$ faults. However, if the system has an unbounded execution time, or if there is a bound on the amount of redundancy available, recoveries are mandatory.

Let us start by defining node-exhaustion-safety.

Definition 3.1. *Let S=$(n, f_a, f_c, met, mift, mirt, mrd)$ be a distributed system with n nodes, able to resist a maximum number $f_a \leq n$ of arbitrary failures, and a maximum number $f_c \leq n$ of crash failures, and let, for all nodes, met represent maximum execution time; $mift$ represent minimum inter-failure time; $mirt$ represent maximum inter-recovery time; and mrd represent maximum recovery degree.*

Then, S is node-exhaustion-safe iff n is such that the system resists $f_a \geq \lceil \frac{\min(met, mirt)}{mift} \rceil$ arbitrary failures and $f_c \geq mrd$ crash failures.

The intuition behind the definition is that a system needs to have enough redundancy to tolerate at least the number of arbitrary faults given by the actual maximum number of faults/intrusions that may happen between recoveries or until execution ends (in case the system never recovers). Moreover, and in order to maintain availability, the system should, in addition, tolerate at least the number of crash faults given by the maximum number of nodes that may recover simultaneously. It is easy to see that an asynchronous distributed system tolerant of a constant number f_a of arbitrary faults is not node-exhaustion-safe [11]:

Corollary 3.2. *Let S be a fault-tolerant distributed system under the asynchronous model, and able to resist a known maximum number f_a of arbitrary node failures. Then, S is not node-exhaustion-safe.*

Proof. If S is asynchronous, then there is no bound on how long S takes to process local or distributed actions. Thus, met and $mirt$ are unbounded, and it is impossible to guarantee $f_a \geq \lceil \frac{\min(met, mirt)}{mift} \rceil$. □

Notice that Corollary 3.2 applies to any type of fault-tolerant asynchronous distributed system, including ones using asynchronous proactive recovery [14, 15, 2, 4, 7]. The ultimate goal of asynchronous proactive recovery is to guarantee that the value of $mirt$ is such that more than f_a node failures never occur. However, as shown above, this is theoretically impossible by definition of asynchrony. In practical terms, it is also readily observable, since the asynchrony pattern leading to exhaustion can be induced by a malicious adversary, as we showed in [11].

This problem can be better understood if we consider the following. Variables met, $mift$, and $mirt$ measure real-time intervals from an omniscient observer's perspective. $mift$ measures the activity leading to the production of node failures due to faults or intrusions. $mift$ largely depends on physical events that happen in real time (e.g., accidental fault generation or external hacker attacks), and in the worst case it is independent of the system's speed of execution. Also, $mift$ needs to be lower-bounded, say by $Mift$: this is an assumption of the type made in several similar systems [14, 2, 4]. If the minimum inter-failure time ($mift$) were not lower-bounded with $Mift$, the adversary would have infinite power (e.g., $mift$=0), and it would be impossible to derive an exhaustion-safe design. Notice that this timing assumption is about fault production and thus is represented in the external timebase, not compromising at all the asynchrony of the system itself, which runs according to the internal timebase. met measures the longest duration of an execution, which is non-definable if the system is asynchronous or if the system executes forever.

Finally, $mirt$ defines the maximum interval between the triggering and termination of two consecutive recoveries. However, note that for a given target f_a and assumed $Mift$, we will obtain a design-time $Mirt \leq f_a \times Mift$. This constant $Mirt$ will be used by the internal system timing to trigger and execute periodic recoveries. It is perhaps important to clarify that the relation between $mirt$ and $Mirt$ is not quite the same as the one existing between $mift$ and $Mift$. $Mift$ is an assumed lower-bound on $mift$, which is necessary as explained above. On the other hand, $Mirt$ is a design-time parameter, which results from the assumed values for f_a and $Mift$. This is where the problems of an asynchronous system start. The mapping between the interval $Mirt$ as seen by the system internally and the actual real-time interval $mirt$ as seen by an omniscient observer depends on the internal system synchrony assumptions. For the sake of giving an example, let us consider an internal time factor (itf), with $mirt$=$Mirt \times itf$.

Consider now an asynchronous fault-tolerant distributed system with a $mift$=20 time units, expected to rejuvenate periodically in intervals shorter than $mirt$=30 time units. Assume also that mrd=1. From Definition 3.1, this system is node-exhaustion-safe iff it is able to resist $f_a \geq 2$ (and $f_c \geq 1$). Assume that one deploys such a system with f_a=2, programmed internally to rejuvenate in order that $Mirt$=30. The system should be node-exhaustion-safe in theory, but this is not necessarily true. Suppose the system's execution is slowed down by an internal time factor itf=3: that is, all system actions run three times slower than expected. This is normal behavior for an asynchronous system, by definition. Then, whatever triggers and executes rejuvenation is also affected by this delay: $mirt$=$Mirt \times itf = 30 \times 3 = 90$ time units. So in reality, the interval between the start and termination of two consecutive rejuvenation periods is 90 time units, instead of 30 time units. However, this interval is long enough for more than two arbitrary faults to occur, inducing a potential system failure.

Several scenarios of interleaving of the external and internal time as just depicted will be modeled in Section 5 by the use of different timebases. Next, we will review a model and a methodology that allow the design of exhaustion-safe proactive recovery systems, even of an asynchronous nature.

4 Proactive Resilience Revisited

Proactive resilience is a new approach to proactive recovery based on architectural hybridization and hybrid distributed system modeling [13]. The Proactive Resilience Model (PRM) states that the architecture of a system enhanced with proactive recovery should be hybrid, i.e., divided in two parts: the proactive recovery subsystem and the payload system, the latter being proactively recovered by the former. Each of these two parts should be built under different timing assumptions and fault models.

The payload system executes the "normal" applications and may work under an asynchronous Byzantine environment. The proactive recovery subsystem executes the proactive recovery protocols that rejuvenate the applications running in the payload part. This subsystem is more demanding, by definition, in terms of timing and fault assumptions, but some of these assumptions depend on the specific proactive recovery protocol, which can be of many types. Thus, we chose to model the proactive recovery subsystem as an abstract component, the Proactive Recovery Wormhole (PRW), which allows many instantiations.

In [12], the PRM and PRW were presented, together with a design methodology under the PRM that was shown to be a way of building generic exhaustion-safe intrusion-tolerant systems. This section refines this design methodology using the definition of node-exhaustion-safety introduced in Section 3.

First, we start by revisiting the definition of the PRW, and then we describe and formally prove that the refined design methodology makes it possible to build node-exhaustion-safe intrusion-tolerant distributed systems.

4.1 The Proactive Recovery Wormhole

The Proactive Recovery Wormhole (PRW) is an abstract distributed component that aims to execute proactive recovery procedures. The architecture of a system with a PRW is suggested in Figure 1. An architecture with a PRW has a local module in some hosts, called the *local PRW*. Depending on the instantiation, the local PRWs may or may not be interconnected by a *control network*. This setup of local PRWs optionally interconnected by the control network is collectively called the *PRW*. The PRW is used to execute proactive recovery procedures of applications running between participants in the hosts concerned, on any normal distributed system architecture (e.g., the Internet). We say that applications run on a *payload system and network*, to differentiate it from the PRW part.

Conceptually, a local PRW should be considered to be a module inside a host, and separated from the OS. In practice, this conceptual separation between the local PRW and the OS can be achieved in either of two ways: (1) the local PRW can be implemented in a separate, tamper-proof hardware module (e.g., a PC appliance board) so that the separation is physical; or (2) the local PRW can be implemented on the native hardware, with a virtual separation and shielding between the local PRW and the OS processes implemented in software.

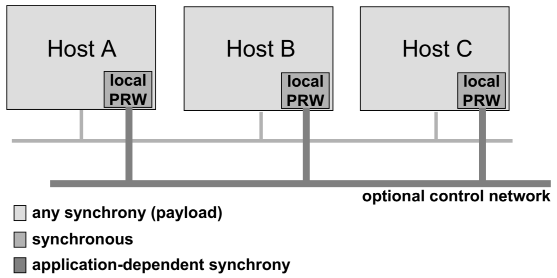


Figure 1. System architecture with a PRW.

The local PRWs are assumed to be fail-silent. Every local PRW preserves, by construction, the following property:

P1 There exists a known upper bound $T^{local}_{exec_{max}}$ on the processing delays.

As mentioned, a PRW instantiation may or may not have a control network. For instance, if a proactive recovery procedure only requires local information, then the control network is expendable. Even when the control network is required, its characteristics will depend on the specific requirements of the proactive recovery procedure.

Notice that the distributed PRW is not necessarily synchronous. Property P1 just says that there is a bound on local processing delays. If the PRW has a control network, this control network can even be asynchronous. The model is sufficiently generic to allow this [13].

The PRW offers a single service, defined as follows:

Definition 4.1. *Given any function F, with a calculated worst-case execution time of T_{Xmax}, an execution interval T_D, and a time interval (period) T_P that satisfies $T_{Xmax} < T_D < T_P$, then F is triggered by the PRW* ***periodic timely execution service*** *at real-time instants t_i (the i^{th} triggering occurs at instant t_i), with $T_D < t_i - t_{i-1} \leq T_P$, and F terminates within T_D from $t_i, \forall i$.*

In short, the PRW has the ability to execute well-defined functions periodically in known bounded time. Moreover, the PRW also accommodates the definition of a set of fail-safe procedures to be triggered in certain situations: these procedures may shut down the system if the *periodic timely execution* service fails to satisfy its specification.

A triple $\langle D, \langle F, T_P, T_D\rangle, S\rangle$ defines a PRW instantiation such that (1) D represents the set of *data* that is proactively recovered in all nodes; (2) $\langle F, T_P, T_D\rangle$ represents the *function F* that is periodically triggered with period T_P and is executed in a timely fashion within T_D of each triggering, through the *periodic timely execution* service, for each node; and (3) S represents the set of (optional) *self-checking* mechanisms, which have the goal of guaranteeing a fail-safe behavior of all the nodes.

4.2 Building Node-Exhaustion-Safe Intrusion-Tolerant Systems

We propose a design methodology to build node-exhaustion-safe distributed intrusion-tolerant systems under the Proactive Resilience Model. The methodology has three steps and, as already stated, is a refinement of the one presented in [12].

1. Define the data D to rejuvenate, define the rejuvenation procedure F, and calculate F's worst-case execution time (T_{Xmax}). Then, define the execution interval T_D (greater than T_{Xmax}) and the periodicity T_P (greater than T_D). Finally, define the actions S to be performed if F is not executed with the required periodicity and execution time.

2. Build a PRW instantiation $\langle D, \langle F, T_P, T_D \rangle, S \rangle$. Notice that T_P and T_D may be increased if necessary. This will only impact the required fault-tolerance degree, as explained in step 3.

3. Define the required degrees f_a and f_c of fault-tolerance, such that $f_a \geq \lceil \frac{\min(met, T_P+T_D)}{mift} \rceil$ and $f_c \geq mrd$. That is, the system must resist at least that number of faults to be node-exhaustion-safe. Needless to say, this ultimately determines n, the required number of nodes.

Given that at most f_a faults are produced between consecutive rejuvenations, it is guaranteed that no more than f_a faults will ever be produced at the same time during the entire execution of the system. Moreover, the system is always able to resist the crash of the maximum number mrd of nodes that recover simultaneously.

Theorem 4.2. *Let $S = (n, f_a, f_c, met, mift, mirt, mrd)$ be a distributed system able to resist at most f_a arbitrary node failures, and at most f_c crash node failures. If S is periodically recovered through a PRW instantiation with parameters T_P and T_D, then S is node-exhaustion-safe iff $f_a \geq \lceil \frac{\min(met, T_P+T_D)}{mift} \rceil$ and $f_c \geq mrd$.*

Proof. By Definition 3.1, S is node-exhaustion-safe iff $f_a \geq \lceil \frac{\min(met, mirt)}{mift} \rceil$ and $f_c \geq mrd$. Thus, it suffices to show that $T_P + T_D = mirt$. Definition 4.1 states that the interval between the triggering of a recovery and the termination of the next consecutive one may assume values in $]T_D, T_P + T_D]$. Thus, no inter-recovery interval may take more than $T_P + T_D$, and then $T_P + T_D = mirt$. $\square$

5 Evaluation

This section presents the results of evaluating, through simulation, whether or not our intrusion-tolerance approach, *proactive resilience*, is sufficient to achieve node-exhaustion-safety for an assumed fault rate. It also shows how previous approaches fail to guarantee such behavior. It starts by presenting the modeling formalism and describing both the models developed to represent an abstract distributed system that periodically recovers, and the adversary that tries to break the system. Then, it presents the results of simulating such an environment when using and not using proactive resilience.

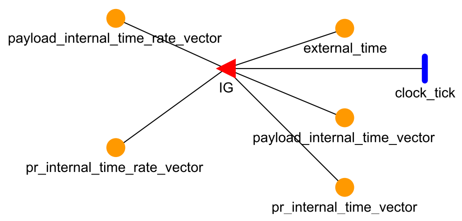

payload_internal_time_rate_vector
external_time
IG
clock_tick
payload_internal_time_vector
pr_internal_time_rate_vector
pr_internal_time_vector

Figure 2. SAN model for the external/internal timebases.

5.1 SAN Models

We use stochastic activity networks (SANs) as the modeling formalism. SANs are probabilistic extensions of activity networks; the type of the extension is similar to the extension that constructs stochastic Petri nets from classical ones. In fact, SANs are a variant of stochastic Petri nets. Due to space limitations, it is not possible to explain SANs in detail. A comprehensive description can be found in [10].

We built atomic SAN submodels for a node and the typical adversary that is constantly trying to corrupt system nodes. We also built submodels of the external/internal timebases and of two types of time adversaries: a conspicuous one that delays the overall system execution, including both the application and the recovery process (e.g., a DoS attack); and a stealth time adversary that slows the internal timebases of the various nodes, thus avoiding detection. The complete model of the system is composed using join operations. We first present a description of each submodel, and then show how the submodels are combined to form the composed model.

In the remainder of the section, Figures 2, 3, and 4 present the SAN models. It is not necessary to understand them in order to follow the explanations in the text. To understand fully the graphical representation of the models and the models themselves, the interested reader can find detailed explanations of the generic SAN's formalism in [10], and the complete documentation of the models at http://www.navigators.di.fc.ul.pt/∼pjsousa/mobius/.

5.1.1 SAN Model for the External/Internal Timebases

The External/Internal Timebases SAN in Figure 2 models the rate at which the external and internal timebases progress. The external timebase advances at the same rate as the simulator clock, while the internal timebase advances at a rate specified by a node parameter, the *internal time rate*. In fact, we model two different internal timebases: the payload one used by normal applications, and the proactive recovery one used by the recovery mechanism. Although these two internal timebases are coincident in a typical homogenous system, we need to separate them in order to model our hybrid architecture.

The SAN model shown in Figure 2 is the basis of our novel approach to intrusion-tolerant modeling and evaluation. The different timebases will be used by the subsequent submodels according to the type of dependencies they have

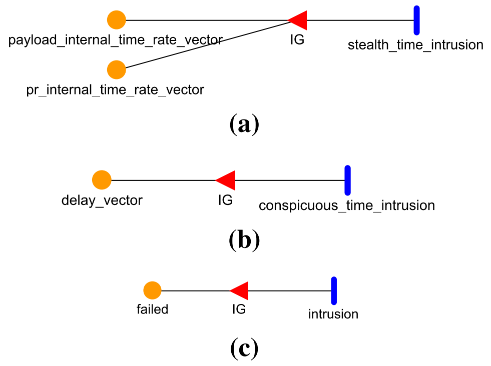

payload_internal_time_rate_vector
IG
stealth_time_intrusion
pr_internal_time_rate_vector
(a)
delay_vector
IG
conspicuous_time_intrusion
(b)
failed
IG
intrusion
(c)

Figure 3. SAN models for the adversaries. (a) Stealth time adversary, (b) Conspicuous time adversary, (c) Classic adversary.

or do not have on the system timing assumptions.

5.1.2 SAN Model for the Stealth Time Adversary

The Stealth Time Adversary SAN in Figure 3(a) models a special kind of adversary. This adversary does not behave like the classic one (described in Section 5.1.4) that simply tries to compromise system nodes, but instead has a more specific goal: to detach a node's internal timebase from the external one. Such an adversary makes it possible to model both random and intentional (i.e., maliciously triggered) asynchrony. The stealth time adversary behavior is specified through two parameters:

stealth time attack period represents the minimum *external time* interval between stealth time attacks. In each attack, the adversary randomly chooses its victim.

stealth time attack factor represents how much "slowness" is injected in each attacked node internal timebase.

5.1.3 SAN Model for the Conspicuous Time Adversary

The Conspicuous Time Adversary SAN in Figure 3(b) models a different type of time adversary that simply delays system actions. This adversary may be seen as a particular case of the generic classic adversary (described in Section 5.1.4) that develops, for instance, a DoS attack. However, we have decided to model it separately in order to discuss in Section 5.2 the theoretical difference between a conspicuous and a stealth time adversary, and how malicious behavior may be detected in one case but not in the other. The conspicuous time adversary behavior is specified through two parameters:

conspicuous time attack period represents the minimum *external time* interval between conspicuous time attacks. In each attack, the adversary randomly chooses its victim.

conspicuous time attack factor represents how much delay is injected in the actions of each attacked node.

5.1.4 SAN Model for the Classic Adversary

The Classic Adversary SAN in Figure 3(c) models the typical adversary that is constantly trying to corrupt system nodes, and that ultimately exhausts the distributed system

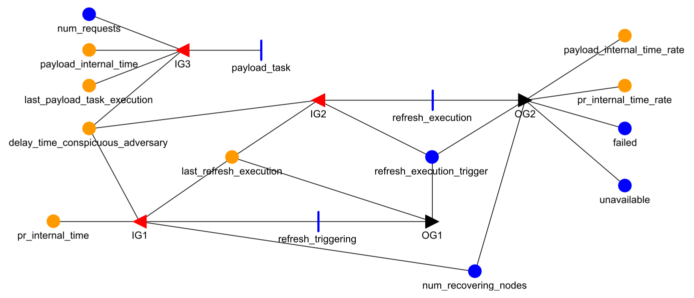


Figure 4. SAN model for a node.

when more than the assumed number of nodes are compromised. The classic adversary behavior is specified through one parameter:

minimum inter-failure time represents the minimum *external time* interval between attacks. In each attack, the adversary randomly compromises one node.

Notice that both the classic and time adversaries have an almost entirely deterministic behavior. The only source of randomness derives from the node targeted in each attack. We could have modeled attack periodicity as random variables following some probabilistic distribution, but we prefer instead to consider the worst-case scenario, given that malicious intelligence will try to make it happen.

5.1.5 SAN Model for a Node

The last submodel is the Node SAN presented in Figure 4, which models the periodic recoveries done at each node. This submodel is more complex than the other ones, mainly because activities, such as refresh triggering and refresh execution, are scheduled according to the *internal timebase*. After a node recovery, the node becomes correct, its internal timebase is re-synchronized with the external one, and any delay injected by the conspicuous adversary is removed. However, during node recovery, the node is considered as being failed even if it was correct before recovery was started. The node behavior is specified through three parameters:

T_P represents the maximum *internal time* interval between two consecutive recoveries.

T_D represents the maximum *internal time* interval between the start and termination of a recovery.

maximum recovery degree represents the maximum number of nodes recovered simultaneously.

5.1.6 Composed Model

The composed model for the simulation environment consists of the five atomic SAN submodels presented above, organized in the following way: one Node SAN per system node (a maximum of 7 nodes is used), one Classic Adversary SAN, one Stealth Time Adversary SAN, one Conspicuous Time Adversary SAN, and one External/Internal Timebases SAN. The composed model also includes a Monitor submodel, which serves only statistical purposes, by collecting statistics about the progress of other SANs.

The overall model behavior is specified by the parameters defined for the five submodels, plus the following: n represents the total number of system nodes, and f represents the assumed maximum number of node failures (the sum of arbitrary and crash failures). When more than f failures happen, the system is considered exhausted.

5.2 Simulation Results

We used the Möbius [5] tool to build the SANs, define the availability and the intrusion tolerance measures, design studies on the model, simulate the model, and obtain values for the measures defined on various studies. The goal of the simulations was to put in evidence the four problems of proactively recovered systems that were introduced in Section 1, recapitulated here: (1) a malicious adversary may deploy more power than originally assumed and corrupt nodes at a pace faster than recovery; (2) he or she may attempt to slow down the pace of recovery in order to leverage the chances of intruding the system with the available power; (3) he or she may perform stealth attacks on the system timing, which in asynchronous or partially synchronous systems may not even be perceived by the essentially time-free logic of the system, leaving it defenseless; and (4) recovery procedures often involve bringing individual nodes to a temporarily inactive state, lowering the redundancy quorum and thus system resilience.

In the next subsections, it is shown that previous approaches to proactive recovery can be affected by all these problems, and that the design methodology presented in Section 4.2 can be used to avoid all of them with the exception of problem 1, which is a fundamental one.

We used two metrics in our simulations: *percentage of exhausted time* and *percentage of unavailability time*. The former shows the amount of time the system has more than f nodes compromised and is thus very vulnerable to failures, especially those maliciously provoked (e.g., in a $3f + 1$ Byzantine-resilient system for f=1, it shows the percentage of time the system runs with at most 2 nodes). The latter shows the amount of time the system is unavailable due to recoveries, i.e., when the system cannot make progress due to an insufficient number of correct replicas, because some of them are recovering. Unless specified otherwise, the simulations were done with parameters n=4, f=1, mrd=1, met=10000, T_P=35, and T_D=4, with times in abstract units. Notice that, according to Theorem 4.2, and in these conditions, the sheer limit of the resistance of the system to attacks lies around $mift=\frac{T_P+T_D}{f}=39$, after which the attack is so powerful that the system starts to give in. This, in fact, is the above mentioned fundamental limitation (problem 1) for any design.

5.2.1 Impact of Time Adversaries on Exhaustion

We start by analyzing problems 2 and 3. The conspicuous time adversary is used to trigger problem 2, and the stealth time adversary is used to trigger problem 3.

Figure 5 illustrates how exhaustion time changes as a function of the combined strength of the classic and the conspicuous time adversaries, when using asynchronous recovery (a) and PRW recovery (b). With asynchronous recovery, the system starts to exhaust with much higher values of $mift$ than the baseline resistance (i.e., with $mift > 39$) in the presence of time attacks of decreasing period (Fig. 5(a)), whereas the PRW renders the system immune to those timing attacks (Fig. 5(b)).

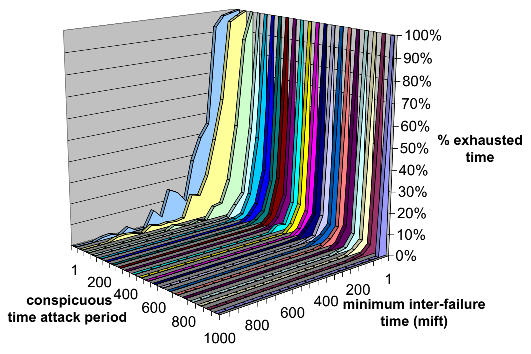
Figure 6 illustrates, for async recovery (a) and PRW recovery (b), the impact of a stealth time adversary that, every 100 time units, slows the internal timebase of a different node down. The graphs depict increasing amounts of speed-down (time attack factor). As in the previous scenario, the system exhausts much faster when the time attack factor increases (Fig. 6(a)), whereas the PRW also renders the system immune to this second type of attacks (Fig. 6(b)). However, note that these attacks are more efficient, since with less power (stealth attacks on, e.g., timers, or interrupt routines, vs. conspicuous direct attacks, e.g., of the denial-of-service type), they achieve a more dire effect, as shown in Figure 6(a): for attack factors of 200 and up, the system becomes almost permanently exhausted; for an attack factor of 1000, the system is exhausted 80% of the time.

Unlike the conspicuous time attack, in which the delay imposed is proportional to the power exerted, in the stealth attack the amount of delay inserted is virtually independent of the initial power used to gain control of the backdoors to the timing devices. Furthermore, the stealth attacker can more easily evade detection than the conspicuous one: the delays injected by the conspicuous adversary may be detected programmatically if the system is partially synchronous and if the internal timebase is not compromised. Moreover, typically these are delays that affect the entire system and may get the attention of monitoring devices.

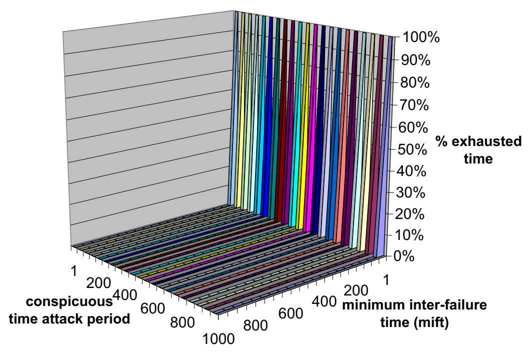
The attack on the internal timebase is completely different in the sense that the adversary is attacking the time references of the system and thus programmatic detections are not reliable, because they use these same time references. Moreover, the attack will not necessarily affect the entire system. For instance, the adversary may tamper with the kernel function that returns the current value of the local clock, and make it return different values to different applications. Or, alternatively, the attack may be applied to kernel scheduler/dispatcher code, only lengthening the execution of the functions used by some processes. Therefore, a stealth time adversary may be very difficult to defend against in classical asynchronous or even partially synchronous systems. The neutralization of this kind of attacks is one of the main results of this paper.

5.2.2 Recovery Strategy and the Trade-off Between Intrusion-Tolerance and Availability

Whenever a node recovers, system availability and/or intrusion-tolerance may be affected. The system architect has to add sufficient redundancy in order to maintain availability and intrusion tolerance during recoveries.

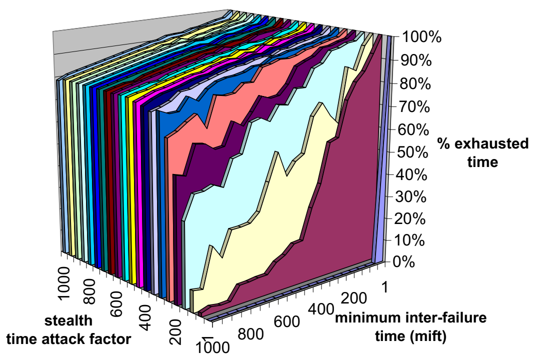


(a)

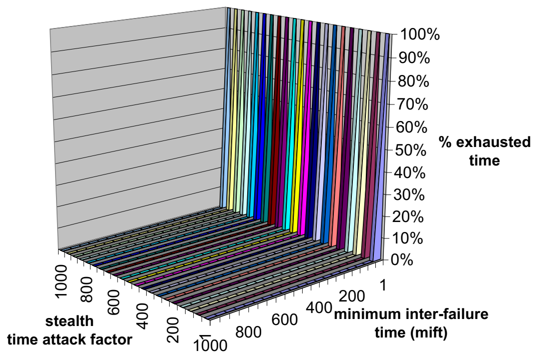


(b)

Figure 5. Exhausted time per conspicuous time attack period and minimum inter-failure time. (a) Async recovery, (b) PRW recovery.



(a)



(b)

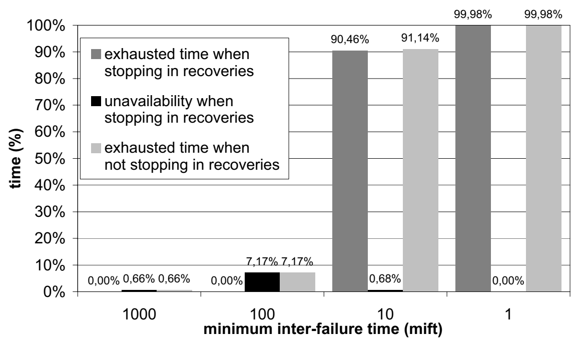
Figure 6. Exhausted time per stealth time attack factor and minimum inter-failure time. (a) Async recovery, (b) PRW recovery.

Let us focus on a typical Byzantine fault-tolerant scenario in which n=4, f=1, and nodes are recovered in sequence, i.e., mrd=1. When a node is being recovered, the system temporarily has a total of three nodes. If, during this time, the system suffers the assumed Byzantine fault, then one of two things happens: service execution is somehow suspended until recovery is finished, or service continues to execute. In the former cases, the system becomes unavailable, whereas in the latter, it becomes exhausted (2 nodes are failed). Notice that this decision should be taken at system design time and performed in an automatic way.

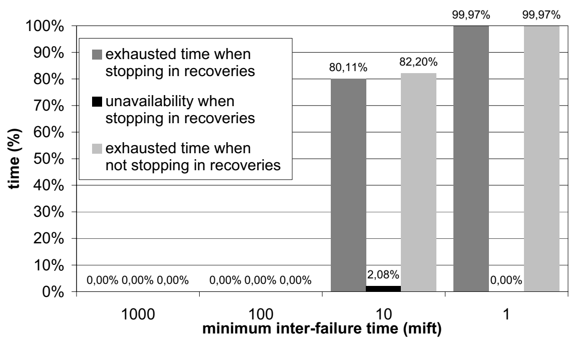
This trade-off between intrusion-tolerance and availability should not be hidden, although availability is a grey notion in asynchronous systems. If the system architect makes the safest option and chooses unavailability, then the system will be *systematically* temporarily unavailable. This is quite different from the normal *stochastic* asynchronous behavior, in which the system may become slower during certain periods of time. On the other hand, avoiding unavailability would make the system systematically exhausted and thus in danger of being compromised.

Figure 7(a) compares exhaustion time and unavailability in each of the scenarios. For this simulation, we set T_P=100 and T_D=10. We see that with $mift$=1000, system resources are never exhausted if the system stops during recoveries: otherwise, exhaustion will occur 0.66% of the time. Then, if the system stops during recoveries, it remains 0% exhausted with $mift$=100, but it is in turn unavailable for 7.17% of the time (precisely the amount of time the system is exhausted if it does not stop during recoveries). Thus, if the system does not stop during recoveries, it is naturally never unavailable due to recoveries, but it exhausts faster and thus has a greater probability of failing.

In order to ensure both intrusion-tolerance and availability, the system needs a sufficient redundancy quorum to avoid exhaustion between and during recoveries. From Theorem 4.2, the system should be able to resist at least $f_a \geq \lceil \frac{\min(met, T_P+T_D)}{mift} \rceil$ arbitrary node failures, and $f_c \geq mrd$ crash node failures. Using the values of the scenario above, and if we assume at design time that $mift \geq 110$, we see that $f = f_a + f_c \geq 1 + 1 = 2$. In order to confirm these calculations, we simulated such a configuration and obtained the progress of exhaustion time in comparison with the decrease of $mift$. Figure 7(b) illustrates the behavior of a distributed system with n=7, f=2. We see that, independent of the strategy followed, no exhaustion or unavailability occurs if the adversary behaves as assumed (i.e., if $mift \geq\sim 110$). Otherwise, we are in problem 1, which is unsolvable. Notice that we are not taking into consideration the effects of intentional or random asynchrony; therefore, T_P and T_D are guaranteed in both Figures 7(a) and 7(b) (the PRW was used in both experiments). The important message here is that, at design time, the system architect should calculate a sufficient redundancy quorum to resist



(a)



(b)

Figure 7. Trade-off between intrusion-tolerance and availability with T_P=100, T_D=10, mrd=1. (a) n=4, f=1, (b) n=7, f=2.

intrusions and maintain availability as long as assumptions on the behavior of the adversary are maintained. And, of course, these assumptions should be realistic.

6 Conclusions

Many Internet services and all critical infrastructure services need to be constantly available and may execute during an unbounded period of time, consequently being subject to an unbounded number of faults. Given that it is not always easy to diagnose faults (e.g., malicious attacks), proactive recovery can be used to tolerate any number of faults during the lifetime of the system.

In this paper, we pointed out the four problems that may affect intrusion-tolerant systems employing proactive recovery: (1) violation of adversary power assumptions; (2) conspicuous time attacks; (3) stealth time attacks; and (4) wrong assumptions on redundancy quorum. We showed, analytically and through simulation, how an architecture and generic algorithmic approach to proactive recovery, globally designated *proactive resilience*, addresses problems 2, 3, and 4. Problem 1 is outside the scope of the paper, and it is fundamentally an unsolvable problem.

The simulation model incorporates a novel technique that allows the system to live under two different timebases: one representing the pace of generation of faults and/or attacks, and another representing its internal execution, e.g., of recoveries. This technique allows us to represent precisely situations that would be hidden in single-timebase models. Specifically, when a stealth attacker, after slowing down important time references of a system, attacks the latter at his or her own pace, the system will react/recover at a pace it "thinks" is much higher than it actually is. This may lead to very serious (because unexpected) failures caused by the exhaustion of system resources.

Acknowledgment

We would like to thank Ms. Jenny Applequist for her editorial comments that improved the quality of this paper.

References

[1] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. of the ACM*, 32(4):824–840, 1985.

[2] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS '02: Proc. of the 9th ACM Conf. on Computer and Comm. Security*, pages 88–97, 2002.

[3] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. of the 25th Annual ACM Symp. on Theory of Computing*, pages 42–51, 1993.

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

[5] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius framework and its implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[6] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC '97: Proc. of the 29th Annual ACM Symp. on Theory of Computing*, pages 569–578, 1997.

[7] M. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Trans. Dependable Sec. Comput.*, 1(1):34–47, 2004.

[8] N. F. Neves, M. Correia, and P. Veríssimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

[9] 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*, pages 51–59, 1991.

[10] W. H. Sanders and J. F. Meyer. Stochastic activity networks: Formal definitions and concepts. In E. Brinksma, H. Hermanns, and J.-P. Katoen, editors, *European Educational Forum: School on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, pages 315–343. Springer, 2000.

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

[12] P. Sousa, N. F. Neves, and P. Veríssimo. Proactive resilience through architectural hybridization. In *Proc. of the 2006 ACM Symp. on Applied Computing*, pages 686–690, 2006.

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

[14] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

[15] L. Zhou, F. B. Schneider, and R. V. Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, 2005.