How Resilient are Distributed f Fault/Intrusion-Tolerant Systems?[*]

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

Abstract

Fault-tolerant protocols, asynchronous and synchronous alike, make stationary fault assumptions: only a fraction f of the total n nodes may fail. Whilst a synchronous protocol is expected to have a bounded execution time, an asynchronous one may execute for an arbitrary amount of time, possibly sufficient for $f + 1$ nodes to fail. This can compromise the safety of the protocol and ultimately the safety of the system. Recent papers propose asynchronous protocols that can tolerate any number of faults over the lifetime of the system, provided that at most f nodes become faulty during a given interval. This is achieved through the so-called proactive recovery, which consists of periodically rejuvenating the system. Proactive recovery in asynchronous systems, though a major breakthrough, has some limitations which had not been identified before. In this paper, we introduce a system model expressive enough to represent these problems which remained in oblivion with the classical models. We introduce the predicate exhaustion-safe*, meaning freedom from exhaustion-failures. Based on it, we predict the extent to which fault/intrusion-tolerant distributed systems (synchronous and asynchronous) can be made to work correctly. Namely, our model predicts the impossibility of guaranteeing correct behavior of asynchronous proactive recovery systems as exist today. To prove our point, we give an example of how these problems impact an existing fault/intrusion-tolerant distributed system, the CODEX system, and having identified the problem, we suggest one (certainly not the only) way to tackle it.*

1 Introduction

Nowadays, and more than ever before, system dependability is an important subject because computers are pervading our lives and environment, creating an ever-increasing dependence on their correct operation. All else being equal, the dependability or trustworthiness of a system is inversely proportional to the number and strength of the assumptions made about the environment where the former executes. This applies to any type of assumptions, namely timing and fault assumptions.

Synchronous systems make timing assumptions, whereas asynchronous ones do not. For instance, if a protocol assumes the timely delivery of messages by the environment, then its correctness can be compromised by overload or unexpected delays. These are *timing faults*, that is, violations of those assumptions. The absence of timing assumptions about the operating environment renders the system immune to timing faults. In reality, timing faults do not exist in an asynchronous system, and this reduction in the fault space makes the former potentially more trustworthy. For this reason, a large number of researchers have concentrated their efforts in designing and implementing systems under the asynchronous model.

Fault assumptions are the postulates underlying the design of fault-tolerant systems: the type(s) of faults, and their number (f). The type of faults influences the architectural and algorithmic aspects of the design, and there are known classifications defining different degrees of severity in distributed systems, according to the way an interaction is affected (e.g., crash, omission, byzantine, etc.), or to the way a fault is produced (e.g., accidental or malicious, like vulnerability, attack, intrusion, etc.). The number establishes, in abstract, a notion of resilience (to f faults occurring). As such, current fault-tolerant system models feature a set of synchrony assumptions (or the absence thereof), and pairs $\langle type, number \rangle$ of fault assumptions (e.g., f omission faults; f compromised/failed hosts).

However, a fundamental goal when conceiving a dependable system is to guarantee that *during* system execution the *actual* number of faults never exceeds the maximum number f of tolerated ones. In practical terms, one would like to anticipate the maximum number of faults bound to occur during the system execution, call it N_f, so that it is designed to tolerate $f \geq N_f$ faults. As we will show, the difficulty of achieving this objective varies not only with the type of faults but also with the synchrony assumptions. Moreover,

[*]This work was partially supported by the FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE).

the system models in current use obscure part of these difficulties, because they are not expressive enough.

Before delving into the formal embodiment of our theory, we give the intuition of the problem. Consider a system where only accidental faults are assumed to exist. If it is synchronous, then one can bound its execution. In consequence, one can forecast the maximum possible number of accidents (faults) that can occur during the bounded execution time, say N_f. That is, given an abstract f fault-tolerant design, there is a justifiable expectation that, in a real system based on it, the maximum number of tolerated faults is never exceeded. This can be done by providing the system with enough redundancy to meet $f \geq N_f$[1]. If the system is asynchronous, then its execution time has not a known bound – it can have an arbitrary finite value. Then, given an abstract f fault-tolerant design, it becomes mathematically infeasible to justify the expectation that the maximum number of tolerated faults is never exceeded, since the maximum possible number of faults that can occur during the unbounded execution time is also unbounded. One can at best work under a partially-synchronous framework where an execution time bound can be predicted with some high probability, and forecast the maximum possible number of faults that can occur during that estimated execution time.

Consider now a system where arbitrary faults of malicious nature can happen. One of the biggest differences between malicious and accidental faults is related with their probability distribution. Although one can calculate with great accuracy the probability of accidents happening, the same calculation is much more complex and/or less accurate for intentional actions perpetrated by malicious intelligence. In the case of a synchronous system with bounded execution time, the same strategy applied to accidental faults can be followed here, except that: care must be taken to ensure an adequate coverage of the estimation of the number of faults during the execution time. If the system is asynchronous, the already difficult problem of prediction of the distribution of malicious faults is amplified by the absence of an execution time bound, which again, renders the problem unsolvable, in theory.

An intuition about these problems motivated the groundbreaking research of recent years around proactive recovery which made possible the appearance of asynchronous protocols and systems [3, 20, 2, 13] that allegedly can tolerate any number of faults over the lifetime of the system, provided that fewer than a subset of the nodes become faulty within a supposedly bounded small window of vulnerability. This is achieved through the use of proactive recovery protocols that regularly rejuvenate the system.

However, having presented our conjecture that the problem of guaranteeing that the actual number of faults in a system never exceeds the maximum number f of tolerated ones, has a certain hardness for synchronous systems subjected to malicious faults, and is unsolvable for asynchronous systems, we may ask: How would this be possible with 'asynchronous' proactive recovery?

This is what we are going to discuss in the remainder of the paper. Firstly, in Section 2, we recall a concept well-known in classical fault-tolerant hardware design, spare exhaustion, and generalize it to *resource exhaustion*, the situation when a system no longer has the necessary resources to execute correctly (computing power, bandwidth, replicas, etc.). We propose to augment system models with the notion of the evolution of environmental resources along the timeline of system execution. Furthermore, we introduce the predicate *exhaustion-safe*, meaning freedom from exhaustion-failures. Based on it, in Section 3, we introduce precise criteria to describe the resilience of fault and/or intrusion-tolerant distributed systems under diverse synchrony assumptions, and we discuss the extent to which systems (synchronous and asynchronous) can be made to work correctly.

Our findings reveal problems that remained in oblivion with the classical models, leading to potentially incorrect behavior of systems otherwise apparently correct. Proactive recovery, though a major breakthrough, has some limitations when used in the context of asynchronous systems. Namely, some proactive recovery protocols depend on hidden timing assumptions which are not represented in the models used. In fact, our model predicts the impossibility of guaranteeing correct behavior of asynchronous proactive recovery systems as exist today. To prove our point, in Section 4, we give an example of how these problems impact an existing fault/intrusion-tolerant distributed system, the CODEX system, and having identified the problem, we suggest one (certainly not the only) way to tackle it. Section 5 concludes the paper and presents future work.

2 Physical System Model

2.1 Additional insight into system correctness

Distributed systems are usually dependent on a set of protocols. Protocol correctness is thus vital to guarantee system correctness. The process of building correct protocols is composed by many steps, from the algorithmic specification until its implementation and testing. We highlight the following:

1. assessing, at *algorithm design time*, if the algorithm underpinning the protocol is correct in an *abstract* computational system;

[1]Just for the sake of example: in an algorithm design where $f = \frac{N-1}{3}$, for N processes, then in the system design, given N_f, N would have to be $N \geq 3N_f + 1$.

2. assessing, at *system design time*, if the protocol will execute correctly in a *concrete* computational system;

3. assessing, at *implementation time*, if the protocol is correctly implemented and then verifying at *run time*, if the protocol executes according to its specification.

This paper is a contribution to steps 1 and 2. Typically, a computational system is defined by a set of assumptions regarding aspects like the processing power, the type of faults that can happen, the synchrony of the execution, etc. These assumptions are in fact an abstraction of the actual resources the protocol needs to work correctly (e.g., when a protocol assumes that messages are delivered within a known bound, it is in fact assuming that the network will have certain characteristics such as bandwidth and latency). The violation of these resource assumptions may affect the safety or liveness of the protocols and hence of the system. We propose to augment system models with the notion of the evolution of environmental resources along the timeline of system execution and its consequent impact on system assumptions.

In this paper we are precisely concerned with the event of 'violation of any of the resource assumptions', which we call *resource exhaustion*, and on the conditions for its avoidance. We start by giving a name to failures caused by resource exhaustion.

Definition 2.1. *An exhaustion-failure is a failure that results from accidental or provoked resource exhaustion.*

Our goal is to prevent exhaustion-failures from happening. Therefore, we define exhaustion-safety in the following manner.

Definition 2.2. *Exhaustion-safety is the ability of a system to ensure that exhaustion-failures do not happen.*

Consequently, an exhaustion-safe system is defined in the following way.

Definition 2.3. *A system is said to be exhaustion-safe if it satisfies the exhaustion-safety property.*

We argue that a system, namely a distributed system, in order to be dependable, has to satisfy the exhaustion-safety property. In other words, a dependable distributed system must be exhaustion-safe.

In the remainder of the paper, we are going to assess how an f fault/intrusion-tolerant distributed system behaves with regard to exhaustion-safety, for different combinations of synchronous/asynchronous timing and accidental/malicious faults. We will consider schemes where the system starts with a number of components, and continues to provide correct responses as long as sufficient components exist.

2.2 The model

Our main goal is to formally reason about how exhaustion-safety may be affected by different combinations of timing and fault assumptions. So, we need to conceive a model in which exhaustion-safety can be formally defined. This model has to take in account the relevant system resources and their evolution with time. For this reason, and short of a better name, we called it a Physical System Model (PSM, for short).

Our model considers systems that have a certain mission. Thus, the execution of this type of systems is composed of various processing steps needed for fulfilling the system mission (e.g., protocol executions). We define three events regarding the system execution: *start*, *termination* and *exhaustion*. Only the start event is mandatory to happen: we cannot talk of a system execution if the system does not start executing. The termination and exhaustion events may or may not happen. More importantly, the causal relation between them is crucial to assess system exhaustion-safety.

We now formally define PSM.

Definition 2.4. *Let A be a system. An A execution is defined by a triple:*
$\mathcal{A} = \langle A_{t_{start}}, A_{t_{end}}, A_{t_{exhaust}} \rangle$, *where*

- $A_{t_{start}} \in \Re_0^+$
  represents the real time start instant.

- $A_{t_{end}} \in [A_{t_{start}}, +\infty[$
  represents the real time termination instant.

- $A_{t_{exhaust}} \in [A_{t_{start}}, +\infty[$
  represents the real time instant when resource exhaustion occurs. If $A_{t_{exhaust}} \leq A_{t_{end}}$, system correctness may be corrupted through exhaustion-failures.

So, under PSM, a system is defined by a set of triples $\mathcal{A}$, one for each of its executions. Next, we formally define what is an exhaustion-safe system under PSM.

Definition 2.5. *A system A is exhaustion-safe if and only if $A_{t_{end}} < A_{t_{exhaust}}, \forall \mathcal{A}$.*

Definition 2.5 states that a system is exhaustion-safe if and only if resource exhaustion does not occur during any execution. This does not mean that the system fails immediately after resource exhaustion. In fact, a system may even present a correct behavior between the exhaustion and the termination events. Thus, a non exhaustion-safe system may execute correctly during its entirely lifetime. However, after resource exhaustion there is *no guarantee* that an exhaustion-failure will not happen. Figure 1 illustrates the differences between an execution of an exhaustion-safe and a non exhaustion-safe system. An exhaustion-safe system is always guaranteedly immune to exhaustion-failures. A non exhaustion-safe system has at least one execution with a period or periods (see Section 3.3) of vulnerability to exhaustion-failures where resources are exhausted and where correctness may be compromised.

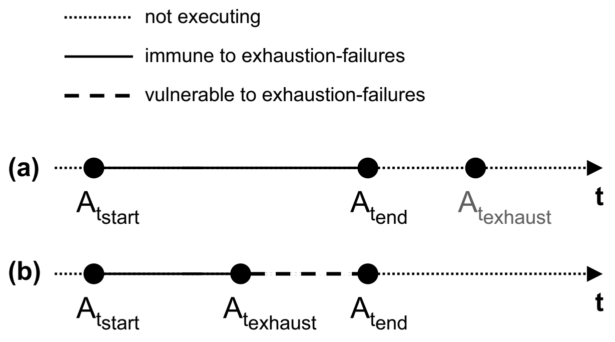

Figure 1. (a) Exhaustion-safe system; (b) non exhaustion-safe system.

As we will show, the main advantage of this more expressive model is that condition $A_{t_{end}} < A_{t_{exhaust}}$ can be evaluated, that is, we can determine whether it is maintained, or not, depending on the type of system assumptions. Note that the idea is not to know the exact values of $A_{t_{start}}, A_{t_{end}}$ and $A_{t_{exhaust}}$, but rather to reason about constraints imposed on them derived from environment assumptions. With this goal in mind, we start by defining two crucial properties of the model, which follow immediately from the above definitions.

Property 2.6. *If $A_{t_{end}}$ has a bounded value T_{end} (i.e., $A_{t_{end}} \leq T_{end}, \forall \mathcal{A}$), then A is exhaustion-safe if $A_{t_{exhaust}} > T_{end}, \forall \mathcal{A}$.*

It is easy to see that $A_{t_{exhaust}} > T_{end} \Rightarrow A_{t_{exhaust}} > A_{t_{end}} \Rightarrow A$ is exhaustion-safe.

Property 2.7. *If $A_{t_{exhaust}}$ has a bounded value $T_{exhaust}$ (i.e., $A_{t_{exhaust}} \leq T_{exhaust}, \forall \mathcal{A}$), then A is exhaustion-safe only if $A_{t_{end}} < T_{exhaust}, \forall \mathcal{A}$.*

It is also easy to see that A is exhaustion-safe $\Rightarrow A_{t_{end}} < A_{t_{exhaust}} \Rightarrow A_{t_{end}} < T_{exhaust}$.

3 Dependability under PSM

In the next sections we analyze and evaluate both worlds of synchronous and asynchronous systems, according to PSM. We will also consider that systems may suffer either from accidental or malicious failures.

3.1 Synchronous systems

Systems developed under the synchronous model are relatively straightforward to reason about and to describe. This model has three distinguishing properties that help us understand better the system behavior: there is a known time bound for the local processing of any operation, message deliveries are performed within a well-known delay, and components have access to local clocks with a known bounded drift rate with respect to real time [9, 19].

If one considers a synchronous system A with a bounded lifetime under PSM, then we can use the worst-case bounds defined during the design phase to assess the conditions of exhaustion-safety.

Corollary 3.1. *If A is a synchronous system with a bounded lifetime (i.e., $\exists T_{end} : A_{t_{end}} \leq T_{end}, \forall \mathcal{A}$), then A is exhaustion-safe if $A_{t_{exhaust}} > T_{end}, \forall \mathcal{A}$.*

Proof. See Property 2.6. □

Therefore, if one wants to design an exhaustion-safe synchronous system with a bounded lifetime, then one has to guarantee that no resource exhaustion occurs during the limited period of time delimited by T_{end}.

Note that Corollary 3.1 only applies to synchronous systems with a bounded lifetime. If the system lifespan is unbounded (e.g., server), then we can prove the following.

Corollary 3.2. *If A is a synchronous system with an unbounded lifetime (i.e., $\nexists T_{end} : A_{t_{end}} \leq T_{end}, \forall \mathcal{A}$), and $A_{t_{exhaust}}$ has a bounded value $T_{exhaust}$ (i.e., $A_{t_{exhaust}} \leq T_{exhaust}, \forall \mathcal{A}$), then A is not exhaustion-safe.*

Proof. If $A_{t_{end}}$ does not have a known bound, it is impossible to guarantee that $A_{t_{end}} < T_{exhaust}, \forall \mathcal{A}$, and therefore, by Property 2.7, A is not exhaustion-safe. □

In fact, synchronous systems may suffer accidental or malicious faults. These faults may have two bad effects: provoking timing failures that increase $A_{t_{end}}$; causing resource degradation which decreases $A_{t_{exhaust}}$. Thus, in a synchronous system, an adversary can not only perform attacks to either crash or control some resources, but also violate the timing assumptions, even if during a limited interval. For this reason, there is currently among the research community a common belief that synchronous systems are fragile, and that secure systems should be built under the asynchronous model.

This section showed that it is possible to have exhaustion-safe f fault/intrusion-tolerant synchronous systems as long as they have a bounded lifetime. However, care must be taken that timing assumptions are not violated during system execution, namely in the presence of malicious faults.

3.2 Asynchronous systems

The distinguishing feature of an asynchronous system is the absence of timing assumptions, which means arbitrary delays for the execution of operations and message deliveries, and unbounded drift rates for the local clocks [7, 12, 6]. This model is quite attractive because it leads to the design of programs and components that are easier to port or include in different environments.

If one considers an asynchronous system A under PSM, then A can be built in such a way that termination is eventually guaranteed (sometimes only if certain conditions become true). However, it is impossible to determine exactly when termination will occur. In other words, the termination instant $A_{t_{end}}$ is unbounded. Therefore, it is necessary to analyze the relation between $A_{t_{end}}$ and $A_{t_{exhaust}}$, in order to assess if A is exhaustion-safe.

Can an asynchronous system A be exhaustion-safe? Despite the arbitrariness of $A_{t_{end}}$, the condition $A_{t_{end}} < A_{t_{exhaust}}$ must always be maintained. This can only be guaranteed in two situations: if $A_{t_{exhaust}}$ has an infinite value or if $A_{t_{exhaust}}$ is correlated with $A_{t_{end}}$ in a way that verifies the condition. Whereas the former condition would mean the impossibility of a failure occurring in the system, which common sense indicates as a very difficult or impossible goal to attain, the latter is very hard to achieve as well. We give a solution through an adequate system architecture, as we will explain later in the paper.

Traditionally, dependable asynchronous systems resort to some form of redundancy, to be able to handle component failures. A usual assumption in the design of these systems is to impose a limit on maximum number of components that can fail during execution. For instance, a reliable broadcast protocol requires that at most $\lfloor \frac{n-1}{3} \rfloor$ out of n components can fail maliciously [1].

In a system that starts with a certain level of redundancy, the assumption that a fixed number of f components may fail results in a (not necessarily known) bounded value for $A_{t_{exhaust}}$: the time necessary to crash/corrupt $f + 1$ components. Notice that this sort of "doom's timer" is started at system boot and tends to decrease as the system evolves. Many systems naively assume that all components are correct when each execution of a protocol is initiated. Unless a protocol begins to run at system boot, or the system is completely re-constructed just before the protocol starts, this assumption is not plausible. Although asynchronous algorithms are designed without timing considerations, once cast into a *system design* they gain an indirect relation with time through the inexorable path of resource exhaustion. Our enriched distributed system model captures this relation through $\mathcal{A}$ and allows one to reason formally about it.

Given that $A_{t_{end}}$ does not have a known bound in asynchronous systems, one can prove the following corollary of Property 2.7, similar to Corollary 3.2:

Corollary 3.3. *If A is an asynchronous system (i.e., $\nexists T_{end} : A_{t_{end}} \leq T_{end}, \forall \mathcal{A}$), and $A_{t_{exhaust}}$ has a bounded value $T_{exhaust}$ (i.e., $A_{t_{exhaust}} \leq T_{exhaust}, \forall \mathcal{A}$), then A is not exhaustion-safe.*

Proof. See Corollary 3.2. □

Even though real systems working under the asynchronous model have a bounded $A_{t_{exhaust}}$, they have been used with success for many years. This happens because, until recently, only accidental faults (e.g., crash or omission) were a threat to systems. This type of faults, being accidental by nature, occur in a random manner. Therefore, by studying the environment in detail and by appropriately conceiving the system, one can achieve an asynchronous system that behaves as if it were exhaustion-safe, with a high probability. That is, despite having the failure syndrome as we have proved, it would be very difficult to observe it in practice.

However, when we start considering malicious faults, a different reasoning must be made. This type of faults is intentional (not accidental) and therefore their distribution is not random: the actual distribution may be shaped at will by an adversary whose main purpose is to break the system. In these conditions, having a bounded $A_{t_{exhaust}}$ (e.g., stationary maximum bound for node failures) in an asynchronous system A may turn out to be catastrophic for the safety of the system. That is, our moderating comments above do not apply to intrusion-tolerant systems working under the asynchronous model.

Consequently, $A_{t_{exhaust}}$ should not have a bounded value if A is an asynchronous system operating in a environment prone to anything more severe than accidental faults. The goal should then be to somehow unbound $A_{t_{exhaust}}$ and maintain it always above $A_{t_{end}}$.

This section showed that it is impossible to have exhaustion-safe f fault/intrusion-tolerant asynchronous systems, namely in the presence of malicious faults.

3.3 Proactive recovery in asynchronous systems

One of the most interesting approaches to avoid resource exhaustion due to malicious compromise of components is through proactive recovery [14] (which can be seen as a form of dynamic replication [15]). The aim of this mechanism is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/failures. If the rejuvenation if performed frequently often, then an adversary is unable to corrupt enough resources to break the system. Proactive recovery has been suggested in several contexts. For instance, it can be used to refresh cryptographic keys in order to prevent the disclosure of too many secrets [11, 10, 8, 21, 2, 20, 13]. It may also be utilized to restore the system code from a secure source to eliminate potential transformation carried out by the adversary [14, 3]. Moreover, it may include substituting the programs to remove vulnerabilities existent in previous versions (e.g., software bugs that could crash the system or software errors exploitable by outside attackers). Thus, by using a well-planned strategy of proactive recovery, $A_{t_{exhaust}}$ can be constantly postponed in order that resource exhaustion never happens before $A_{t_{end}}$. This intuition is illustrated in Figure 2, featuring a system A with a proactive recovery subsystem A'. In Figure 2a we can see that as execution approaches $A_{t_{exhaust}}$, the system may risk exhaustion-failures. However (Figure 2b), the execution of a proactive recovery procedure by A' – triggered at instant $pr_{t_{start}}$ and terminated at instant $pr_{t_{end}}$ – causes the postponing of $A_{t_{exhaust}}$.

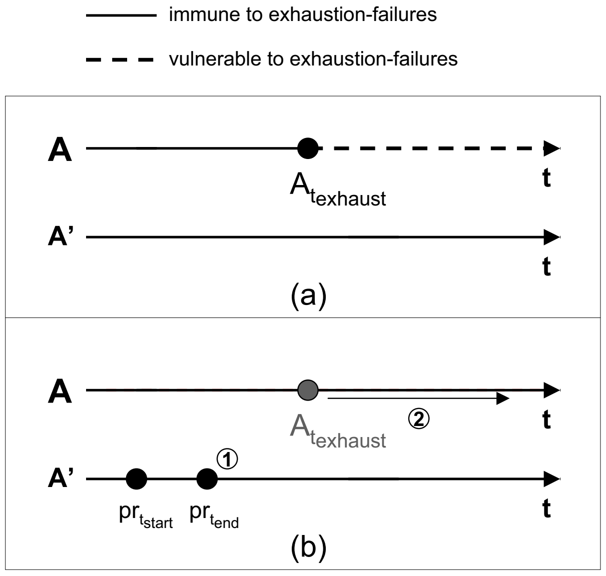
immune to exhaustion-failures
vulnerable to exhaustion-failures
A
A_texhaust
t
A'
t
(a)
A
A_texhaust
②
t
A'
①
pr_tstart
pr_tend
t
(b)

Figure 2. (a) Before proactive recovery being executed, A exhaustion-safety is in risk of being violated; (b) after the execution of proactive recovery (1), $A_{t_{exhaust}}$ is postponed (2).

The following theorem states a necessary condition for the behavior illustrated in Figure 2b.

Theorem 3.4. *Consider a system A enhanced with a proactive recovery subsystem A', which rejuvenates*[2] *system A. Consider that after a rejuvenation i, $A_{t_{exhaust}}$ is bounded by $T^i_{exhaust}$, and that the time of completion of the (next) rejuvenation $i+1$ is bounded by $T^i_{rejuvenation}, \forall \mathcal{A}, i$. If $A_{t_{end}}$ has an unbounded value (i.e., $\nexists T_{end} : A_{t_{end}} \le T_{end}, \forall \mathcal{A}$), then A is exhaustion-safe only if $T^i_{rejuvenation} < T^i_{exhaust}, \forall \mathcal{A}, i$.*

[2]For instance, the rejuvenation may be periodic.

Proof. In order to prove by contradiction, let us assume that $\exists \mathcal{A}, i : T^i_{exhaust} \le T^i_{rejuvenation}$ and A is exhaustion-safe. $T^i_{exhaust} \le T^i_{rejuvenation} \Rightarrow A_{t_{exhaust}} \le T^i_{rejuvenation} \le A_{t_{end}}$. Therefore, A is not exhaustion-safe. This contradicts the hypothesis. □

Observe that this theorem applies to any type of system A and A', independently of their synchrony assumptions.

Let us now focus on asynchronous proactive recovery. An asynchronous system with proactive recovery is represented as in Figure 3. The asynchronous system A is enhanced with a subsystem A' responsible for the proactive recovery operations. As expected, A' is also asynchronous because it is part of A.

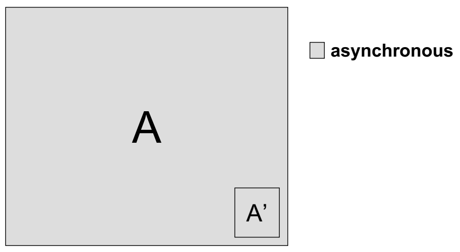

Figure 3. A system A enhanced with a proactive recovery subsystem A'. Both A and A' run asynchronously.

Some proactive recovery protocols for asynchronous systems have been proposed in the literature [21, 2, 3]. Despite having different goals, their effectiveness depends on the same assumption: regular execution. They assume that the proactive subsystem is regularly executed, and that the rejuvenation operation does not take a very long period to complete. Now suppose that a proactive recovery system makes timing assumptions (implicit or explicit) about the environment, which by definition, can be violated in an asynchronous setting. Figure 4 shows an example of how an adversary can deploy an exploit that takes advantage of these timing assumptions and consequently compromise A exhaustion-safety. Firstly, the adversary forces the violation of the timing assumptions by slowing the system down through more or less visible actions (e.g., by compromising the clock behavior) and in this way delays the triggering of the proactive recovery procedure. In consequence, the rejuvenation is not completed in time to avoid A's resource exhaustion. Between resources being exhausted at $A_{t_{exhaust}}$ and resources being rejuvenated at $pr_{t_{end}}$, there is an interval of time where A's correctness may be compromised. Although A immunity to exhaustion-failures is reestablished after $pr_{t_{end}}$, its correctness may already have been corrupted if some of its safety properties were violated in the interval $[A_{t_{exhaust}}, pr_{t_{end}}]$ through some exhaustion-failure. Notice that although we consider A' as a subsystem that is inside A, Figure 4 represents A' as being immune to exhaustion-failures during the period $[A_{t_{exhaust}}, pr_{t_{end}}]$ when A is vulnerable. Our intention was simply to ease the understanding of this figure. For the interested reader, the insight about this problem concerns coverage of timing assumptions and contamination by timing failures, and was equated in [17].

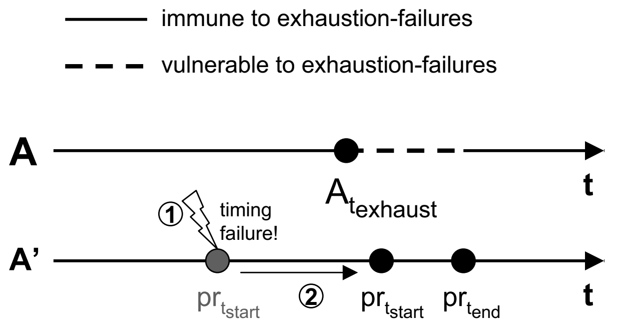
immune to exhaustion-failures
vulnerable to exhaustion-failures
A
A'
t
A_texhaust
① timing failure!
t
pr_tstart
②
pr_tstart
pr_tend

Figure 4. The violation of A''s timing assumptions (1), causes the delay of proactive recovery execution (2), which thus fails to guarantee A exhaustion-safety.

So, the asynchronous proactive recovery subsystem A' effectiveness depends on timing assumptions that can be violated, and for that reason A' cannot permanently guarantee the exhaustion-safety of the asynchronous system A. More formally:

Corollary 3.5. *Consider an asynchronous system A (i.e., $\nexists T_{end} : A_{t_{end}} \leq T_{end}, \forall \mathcal{A}$) enhanced with an asynchronous proactive recovery subsystem A', which rejuvenates system A. Consider that after a rejuvenation i, $A_{t_{exhaust}}$ is bounded by $T^i_{exhaust}$ and that the time of completion of the (next) rejuvenation $i+1$ is bounded by $T^i_{rejuvenation}, \forall \mathcal{A}, i$. Then, A is not exhaustion-safe.*

Proof. Theorem 3.4 states that if $A_{t_{end}}$ has an unbounded value, then A is exhaustion-safe only if $T^i_{rejuvenation} < T^i_{exhaust}, \forall \mathcal{A}, i$. Given that A is asynchronous, $A_{t_{end}}$ has an unbounded value and the asynchrony of A' implies that it cannot guarantee the condition $T^i_{rejuvenation} < T^i_{exhaust}, \forall \mathcal{A}, i$. Therefore, A is not exhaustion-safe. □

Regarding adversary models, some authors [21] distinguish between the adversary being an agent whose ability to compromise the system depends on the time available for attacks, and it relying on intrinsic aspects of the system, such as the operating system or application software. In reality these are false alternatives since both facets must be present: following [18], for there to be an intrusion, there must be a vulnerability (e.g., "an intrinsic aspect of the system"), which is attacked successfully (and that requires some time). From the discussion above, it should be evident that we follow this composite adversary model.

This section showed that even using asynchronous proactive recovery, it is impossible to have exhaustion-safe f fault/intrusion-tolerant asynchronous systems, namely in the presence of malicious faults.

To illustrate these conclusions in a real system, we will describe in the next section a possible attack to CODEX [13] that is based on the time-related vulnerability of the proactive recovery protocols it uses, predicted by our results under PSM and exhaustion-safety.

4 An attack to the proactive recovery scheme of CODEX

CODEX (COrnell Data EXchange) is a recent distributed service for storage and dissemination of secrets [13]. It binds secrets to unique names and allows subsequent access to these secrets by authorized clients. Clients can call three different operations that allow them to manipulate and retrieve bindings: *create* to introduce a new name; *write* to associate a (presumably secret) value with a name; and *read* to retrieve the value associated with a name.

The service makes relatively weak assumptions about the environment and the adversaries. It assumes an asynchronous model where operations and messages can suffer unbounded delays. Moreover, messages while in transit may be modified, deleted or disclosed. An adversary can also insert new messages in the network. Nevertheless, it is assumed fair links, which means that if a message is transmitted a number of times from one node to another, then it will eventually be received.

CODEX enforces three security properties. Availability is provided by replicating the values in a set of n servers. It is assumed that at most f servers can (maliciously) fail at the same time, and that $n \geq 3f + 1$. Cryptographic operations such as digital signatures and encryption/decryption are employed to achieve confidentiality and integrity of both the communication and stored values. These operations are based on public key and threshold cryptography. Each client has a public/private key pair and has the CODEX public key. In the same way, CODEX has a public/private key pair and knows the public keys of the clients. The private key of CODEX however is shared by the n CODEX servers using an $(n, f+1)$ secret sharing scheme[3], which

[3]In a $(n, f+1)$ secrete sharing scheme, there are n shares and any subset of size $f+1$ of these shares is sufficient to recover the secret. However, nothing is learnt about the secret if the subset is smaller than $f+1$.

means that no CODEX server is trusted with that private key. Therefore, even if an adversary controls a subset of f or less replicas, she or he will be unable to sign as CODEX or to decrypt data encrypted with the CODEX public key.

In CODEX, both requests and confirmations are signed with the private key of, respectively, the clients or CODEX (which requires the cooperation of at least $f + 1$ replicas). Values are stored encrypted with the public key of CODEX, which prevents disclosure while transit through the network or by malicious replicas. The details of the CODEX client interface, namely the message formats for each operation, can be found in [13]. At this moment, we just want to point out that by knowing the CODEX private key, one can violate the confidentiality property in different ways.

4.1 Overview of the proactive recovery scheme

An adversary must know at least $f + 1$ shares in order to construct the CODEX private key. CODEX assumes that a maximum of f nodes running CODEX servers are compromised at any time, with $f = \frac{n-1}{3}$. This assumption excludes the possibility of an adversary controlling $f + 1$ servers, but as the CODEX paper points out, "it does not rule out the adversary compromising one server and learning the CODEX private key share stored there, being evicted, compromising another, and ultimately learning $f+1$ shares". To defend against these so called *mobile virus attacks* [14], CODEX employs the APSS proactive secret sharing protocol [21]. "This protocol is periodically executed, each time generating a new sharing of the private key but without ever materializing the private key at any server". Because older secret shares cannot be combined with new shares, the CODEX paper concludes that "a mobile virus attack would succeed only if it is completed in the interval between successive executions of APSS". This scenario can be prevented from occurring by running APSS regularly, in intervals that "can be as short as a few minutes".

4.2 An example attack

We now describe an attack that explores the asynchrony of APSS with the goal of increasing its execution interval, to allow the retrieval of $f+1$ shares and the disclosure of the CODEX private key. Once this key is obtained, it is trivial to breach the confidentiality of the service.

The intrusion campaign is carried out by two adversaries, ADV1 and ADV2. ADV1's attack takes the system into a state where the final attack can be performed by the second adversary. As expected, both adversaries will execute the attacks without violating any of the assumptions presented in the CODEX paper. ADV1 basically delays some parts of the system – it slows down some nodes and postpones the delivery of messages between two parts of the system (or temporally partitions the network). The reader should notice that after this first attack the system will exhibit a behavior that could have occurred in any fault-free asynchronous system. Therefore, this attack simply forces the system to act in a manner convenient for ADV2, instead of having her wait for the system to naturally behave in such way.

Attack by adversary ADV1: ADV1 performs a mobile virus attack against $f+1$ servers. However, instead of trying to retrieve the CODEX private key share of each node, it does a much simpler thing: it adjusts, one after the other, the rate of each local clock. The adjustment increases the drift rate to make the clock slower than real time. In other words, 1 system second becomes λ real time seconds, where $\lambda \gg 1$.

APSS execution is triggered either by a local timer at each node or by a notification received from another node[4]. This notification is transmitted during the execution of APSS. The mobile virus attack delays at most $f + 1$ nodes from starting their own APSS execution, but it does not prevent the reception of a notification from any of the remainder $n - (f + 1)$ nodes. Therefore, various APSS instances will be run during the attack.

After slowing down the clock of $f + 1$ nodes, ADV1 attacks the links between these nodes and the rest of the system. Basically, it either temporally cuts off the links or removes all messages that could (remotely) initiate APSS. The links are restored once ADV2 obtains the CODEX private key, which means that messages start to be delivered again and the fair links assumption is never violated.

The reader should notice that the interruption of communications is not absolutely necessary for the effectiveness of the ADV2 attack. Alternatively, one could extend the mobile attack to the n nodes and in this way delay APSS execution in all of them.

Attack by adversary ADV2: ADV2 starts another mobile virus attack against the same $f + 1$ nodes that were compromised by ADV1. Contrarily to the previous attack, this one now has a time constraint: the APSS execution interval. Remember that $f + 1$ shares are only useful if retrieved in the interval between two successive executions of APSS. However, for all practical considerations, the time constraint is removed, since the clocks are made as slow as needed, by a helping accomplice – ADV1. Thus, the actual APSS interval is much larger than expected.

Without any time constraint, it suffices to implement the mobile virus attack suggested in the CODEX paper, learning, one by one, $f + 1$ CODEX private key shares. The

[4]These triggering modes were confirmed by the inspection of the CODEX code, which is available at `http://www.umiacs.umd.edu/~mmarsh/CODEX/`.

CODEX private key is disclosed using these shares. Using this key, ADV2 can decrypt the secrets stored in the compromised nodes. Moreover, she can get all new secrets submitted by clients through *write* operations.

The described attack explores one flaw on the assumptions of CODEX. It implicitly assumes that although embracing the asynchronous model, it can have access to a clock with a bounded drift rate. But, by definition, in an asynchronous system no such bounds exist [7, 12, 6]. Typically, a computer clock has a bounded drift rate ρ guaranteed by its manufacturer. However, this bound is mainly useful in environments with accidental failures. If an adversary gains access to the clock, she or he can arbitrarily change its progress in relation to real time.

More generally, the concept of proactive recovery has some compatibility problems with the asynchronous model. In [21], authors briefly discuss some of these problems, and conclude that the definition of the window of vulnerability in terms of events rather than the passage of time, can potentially afford attackers leverage. In fact, asynchronous systems evolve at an arbitrary pace, while proactive recovery has natural timeliness requirements: proactive recovery leverages the defenses of a system by *periodically* "removing" the work of an attacker. Despite these problems, we subscribe the discussion about APSS vs a PSS (synchronous proactive secret sharing) protocol in [21]: APSS will defend against any attack that the PSS protocol does and will also defend against some attacks that compromise the PSS protocol, such as attacks that invalidate PSS timing assumptions. This goes in line with what we say in the final of Section 3.1 about the greater fragility of synchronous systems. However, APSS is still vulnerable to time attacks such as the one presented above. Therefore, asynchronous systems enhanced with proactive recovery subsystems are in fact promising but care must be taken in their design.

4.3 Combining proactive recovery and wormholes

In this section, we propose one solution to the problem of ensuring exhaustion-safe operation of proactive recovery systems. The solution is based on the concept of wormholes: subsystems capable of providing a small set of services, with good properties that are otherwise not available in the rest of the system [16]. For example, an asynchronous system can be augmented with a synchronous wormhole that offers a few and well-defined timely operations. Wormholes must be kept small and simple to ensure the feasibility of building them with the expect trustworthy behavior. Moreover, their construction must be carefully planed to guarantee that they have access to all required resources when needed. In the past, two incarnations of distributed wormholes have already been created, one for the security domain [5] (the TTCB) and another for the time domain [17] (the TCB).

Remember that as explained in Sections 3.2 and 3.3, it is impossible to guarantee the exhaustion-safety of an asynchronous system A when $A_{t_{exhaust}}$ has a bounded value (Corollary 3.3), even with an asynchronous proactive recovery scheme (Corollary 3.5). The reader however should notice that the main difficulty with proactive recovery is not the concept but its implementation – this mechanism is useful to artificially increase $A_{t_{exhaust}}$ as long as it has timeliness guarantees. Therefore, we probably can find a solution to this problem by revisiting the system and the proactive recovery subsystem under an architecturally hybrid distributed system model, and using a wormhole to implement the latter.

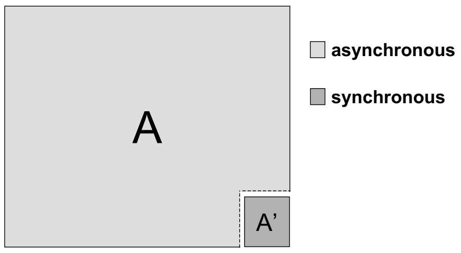

Figure 5. A system A enhanced with a proactive recovery subsystem A'. A runs asynchronously, but A' runs synchronously in the context of a secure and timely wormhole.

We could use the TTCB *Timely Execution Service* to timely execute proactive recovery protocols. The feasibility of building such a service in a real system is confirmed by the already available implementation[5] of the TTCB for the RTAI [4] operating system.

We let as future work the conception of a wormhole specifically tailored for proactive recovery. We envisage that this wormhole will be simpler and will require weaker environment assumptions than the TTCB. A representation of a system using a wormhole to execute proactive recovery procedures is depicted in Figure 5.

Because it makes synchronous assumptions, the wormhole is in theory subject to the same kind of problems described in Section 3.1. However, in practice, the wormhole can be a small and simple component, and thus, as explained in [5], it can be constructed in order to dependably guarantee secure and timely behavior.

5 Conclusions and future work

This paper has made a discussion about the actual resilience of synchronous and asynchronous systems. We

[5]Available at `http://www.navigators.di.fc.ul.pt/software/tcb/downloads.htm`.

proposed a system model that takes in account the environmental resources and their evolution along the timeline of system execution, and introduced the predicate *exhaustion-safe*, meaning freedom from exhaustion-failures.

Based on it, we predicted the extent to which fault/intrusion-tolerant distributed systems (synchronous and asynchronous) can be made to work correctly. We showed that it is possible to have exhaustion-safe f fault/intrusion-tolerant synchronous systems as long as they have a bounded lifetime, with the remark that timing assumptions must not be violated. We also showed that it is impossible to have exhaustion-safe f fault/intrusion-tolerant asynchronous systems. Even proactive recovery in asynchronous systems, though a major breakthrough in that context, has some limitations which had not been identified before. We explained these limitations and showed them in practice through an attack to the CODEX system that does not violate any of the assumptions underlying its operation. Finally, we proposed the combined use of proactive recovery and wormholes as a possible approach to circumvent these limitations.

As future work, we intend to study in more detail the combination of proactive recovery and wormholes. Our goal is to define a hybrid wormhole-enhanced architecture that guarantees the safety of the asynchronous (or synchronous) payload part, despite any number of arbitrary faults, through the wormhole-based timely execution of proactive recovery protocols.

Acknowledgments

We would like to thank Antónia Lopes, and the anonymous referees for their valuable comments on improving this paper.

References

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

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

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

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

[5] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 234–252, Oct. 2002.

[6] F. Cristian and C. Fetzer. The timed asynchronous system model. In *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 140–149, 1998.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[8] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theor. Comput. Sci.*, 243(1-2):363–389, 2000.

[9] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[10] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 100–110. ACM Press, 1997.

[11] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 339–352. Springer-Verlag, 1995.

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[13] M. A. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January–March 2004.

[14] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 51–59. ACM Press, 1991.

[15] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation (2nd Edition)*. Digital Press, 1992.

[16] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 108–113. Springer-Verlag, 2003.

[17] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, Aug. 2002. Preliminary version as DI/FCUL Technical Report 99–2.

[18] P. Veríssimo, N. F. Neves, C. Cachin, J. A. Poritz, D. Powell, Y. Deswarte, R. J. Stroud, and I. S. Welch. Intrusion-tolerant middleware: the MAFTIA approach. DI/FCUL TR 04–14, Department of Informatics, University of Lisbon, November 2004.

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

[20] 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, Nov. 2002.

[21] L. Zhou, F. B. Schneider, and R. van Renesse. Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Cornell University, New York, Oct. 2002.