How dependable are distributed f fault/intrusion-tolerant systems?

Paulo Sousa
Nuno Ferreira Neves
Paulo Veríssimo

DI–FCUL TR–05–3

February 2005

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

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

How dependable are distributed f fault/intrusion-tolerant systems?*

Paulo Sousa, Nuno Ferreira Neves, and Paulo Veríssimo

Faculdade de Ciências da Universidade de Lisboa
Bloco C6, Campo Grande, 1749-016 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 window of time. 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 a classification of system correctness based on the predicate *exhaustion-safe*, meaning freedom from resource exhaustion. 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, 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

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

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 them 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 systems 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 a priori the maximum number of faults predicted 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, 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 its execution time is bounded. 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 $f \geq N_f$.

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, 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, 14] 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 periodically 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, 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 complement system models with the notion of their environmental resources and their evolution along the execution time. Furthermore we introduce a classification of system correctness based on the predicate *exhaustion-safe*, meaning freedom from resource exhaustion. Based on it, we introduce precise criteria to describe the dependability 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, 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.

2 The Physical System Model

2.1 Additional Insight into System Correctness

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. From these collectively one can define the resources the protocol has access to, both in the abstract and in the real target computational system, both at design and at run time. These resources may include CPU, memory, clock and network with a given capacity. In the case of fault-tolerant protocols, it may also include a certain level of replicated components. The violation of these resource assumptions may affect the safety or liveness of the protocols and hence of the system.

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 define exhaustion-safety in the following manner.

Definition 1. *Exhaustion-safety is the ability of a system to assure that it does not fail due to accidental or provoked resource exhaustion.*

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

Definition 2. *A system is said to be exhaustion-safe if it satisfies the exhaustion-safety property.*

In the IEEE standard computer dictionary, correctness is defined as "the degree to which a system or component is free from faults in its specification, design, and implementation" [12].

We argue that not considering exhaustion-safety as part of a system specification, constitutes a specification fault, and ultimately affects system correctness.

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

2.2 The model

Our system model should be expressive enough to allow for an outside omniscient observer to assess system correctness, not only as usual, but also taking in consideration the property *exhaustion-safety*. In consequence, the system model

must encompass the system resources and their evolution with time, that is, it must represent the physical environment where all computational components execute, and its laws of evolution. For this reason, and short of a better name, we called it the Physical System Model.

An outside observer capable of measuring real times knows exactly when a system begins operating, its execution interval and when it terminates. Moreover, it can also find out when, with what rate, and in what way, components fail in the system. One should notice however that these times are independent from any internal timebase of the system, which can even be asynchronous. An outside observer with complete knowledge of the system is also able to make predictions about the future system behavior. For instance, it can calculate a minimum time necessary for system corruption by resource exhaustion by estimating how long it takes for a certain number of components to fail.

We now formally define our model.

Definition 3. *Given a system A, the Physical System Model that allows to assess A correctness, according to the exhaustion-safety property, is defined by a triple:*

$\mathcal{A} = \langle A_{t_{start}}, A_{t_{end}}, A_{t_{corrupt}} \rangle$*, where*

- $A_{t_{start}} \in \Re_0^+$
  represents system A real time start instant.
- $A_{t_{end}} \in [A_{t_{start}}, +\infty[$
  represents system A real time termination instant.
- $A_{t_{exhaust}} \in [A_{t_{start}}, +\infty[$
  if $A_{t_{exhaust}} \leq A_{t_{end}}$, it represents the real time instant when resource exhaustion occurs. After this instant, system correctness may be corrupted. Otherwise, if $A_{t_{exhaust}} > A_{t_{end}}$, it represents the real time instant when resource exhaustion would occur.

We can now prove a necessary and sufficient condition for system exhaustion-safety under the $\mathcal{A}$ model.

Proposition 1. *A system A is exhaustion-safe under the $\mathcal{A}$ model if and only if $A_{t_{end}} < A_{t_{exhaust}}$.*

Proof. First we will prove the necessary condition and then the sufficient condition.

- A is exhaustion-safe $\Rightarrow A_{t_{end}} < A_{t_{exhaust}}$.
  If A is exhaustion-safe, it means that it is guaranteed that A does not fail due to resource exhaustion. This guarantee can only be given if resource exhaustion does not occur, which implies $A_{t_{end}} < A_{t_{exhaust}}$.
- $A_{t_{end}} < A_{t_{exhaust}} \Rightarrow A$ is exhaustion-safe.
  If $A_{t_{end}} < A_{t_{exhaust}}$, it means that resource exhaustion does not occur, and thus it is guaranteed that the system does not fail due to resource exhaustion. Therefore, A is exhaustion-safe.

Depending on the relation of $A_{t_{exhaust}}$ with $A_{t_{start}}$ and $A_{t_{end}}$, it is possible to distinguish two classes of correctness:

Permanently Correct Systems that satisfy the condition of Proposition 1. That is, a system A is *permanently correct* ($\mathcal{PC}$) if and only if $A_{t_{exhaust}} > A_{t_{end}}$.

Temporarily Correct Systems that execute for at least one instant before resource exhaustion occurs. That is, a system A is *temporarily correct* ($\mathcal{TC}$) if and only if $A_{t_{exhaust}} > A_{t_{start}}$.

In terms of resource exhaustion, $\mathcal{PC}$ systems are more dependable than $\mathcal{TC}$ systems. Notice also that we could have enumerated a third class of "always incorrect" systems (where $A_{t_{exhaust}} = A_{t_{start}}$), but this class would be empty because no practical system should have its resources permanently exhausted. So, under the $\mathcal{A}$ model, all systems belong to the $\mathcal{TC}$ class and a subset of them, those that are always correct w.r.t. resource exhaustion, belong to the $\mathcal{PC}$ class.

3 Dependability Under The New Classification

In the next sections we analyze and evaluate both worlds of synchronous and asynchronous systems, according to the new classification. We will also consider that systems either suffer 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 under the $\mathcal{A}$ model described in the previous section, then we can use the worst-case bounds defined during the design phase to evaluate the correctness.

Theorem 1. *Consider a synchronous system A under the $\mathcal{A}$ model. Let T_{bound} designate a known bound on A execution time, such that $A_{t_{end}} \leq A_{t_{start}} + T_{bound}$. If $A_{t_{exhaust}} > A_{t_{start}} + T_{bound}$, then A is $\mathcal{PC}$.*

Proof. If $A_{t_{end}} \leq A_{t_{start}} + T_{bound}$ and $A_{t_{exhaust}} > A_{t_{start}} + T_{bound}$, then $A_{t_{end}} < A_{t_{exhaust}}$.

Therefore, if a designer wants to build a $\mathcal{PC}$ synchronous system, then she or he will have to guarantee that no resource exhaustion occurs during the limited period of time T_{bound}. If, for some reason, the assumed bound T_{bound} is not guaranteed during system operation, then, albeit $\mathcal{PC}$ in theory, system correctness may be compromised.

In fact, synchronous systems may be subject to accidental or malicious faults. These faults may have two bad effects: provoke timing failures that increase $A_{t_{end}}$; cause resource degradation which decreases $A_{t_{exhaust}}$. Therefore, dependable synchronous (or real-time) systems address this issue by including enough redundancy in the implementation. If the system can experience malicious faults the solution (achieving intrusion tolerance) becomes much harder. 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.

3.2 Asynchronous Systems

The distinguishing feature of an asynchronous model 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, 13, 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. Moreover, these programs and components are more tolerant to variable or unexpected delays, caused for instance by temporary system overloads.

If one considers an asynchronous system A under the $\mathcal{A}$ model presented in Section 2, then A can be built in such a way that completion 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 not known. Therefore, it is necessary to analyze the relation between $A_{t_{end}}$ and $A_{t_{exhaust}}$, in order to classify system A according to the $\mathcal{PC}$ and $\mathcal{TC}$ dependability classes.

Can an asynchronous system A be $\mathcal{PC}$? Despite the arbitrariness of $A_{t_{end}}$, $A_{t_{exhaust}}$ must always be maintained above its value to ensure permanent correctness. This can only be guaranteed in two situations: if $A_{t_{exhaust}}$ has an infinite value or if $A_{t_{exhaust}}$ is correlated with $A_{t_{end}}$. 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 one can only be achieved 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]. That is, faults are assumed to be stationary.

On 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 $A_{t_{exhaust}}$ – that results from the time necessary to crash/corrupt $f+1$ components. Notice that this sort of "doom timer" is started

at system boot and tends to decrease as the system evolves. Many protocols naively assume that all components are correct when the protocol is initiated. Unless a protocol begins to run at system boot, or the system is completely reconstructed whenever the protocol starts, this assumption is not plausible and can result in a failure. Therefore, although asynchronous systems are designed without timing considerations, they have an indirect relation with time through $A_{t_{exhaust}}$, and their correctness is time dependent. Given that $A_{t_{end}}$ does not have a known bound on these systems, one can prove the following theorem:

Theorem 2. *Consider an asynchronous system A under the $\mathcal{A}$ model. If $A_{t_{exhaust}}$ has a bounded value, then A is not $\mathcal{PC}$.*

Proof. In order to prove by contradiction, assume that system A is $\mathcal{PC}$ and that $A_{t_{exhaust}}$ is bounded. Let t_k be the real time instant corresponding to the bound of $A_{t_{exhaust}}$, that is, $A_{t_{exhaust}} \leq t_k$. If a system is $\mathcal{PC}$ then it is true that $A_{t_{end}} < A_{t_{exhaust}}$. Therefore, $A_{t_{end}} < A_{t_{exhaust}} \leq t_k$. But, if A is asynchronous, $A_{t_{end}}$ does not have a defined bound, which means that there is a contradiction.

Even though practical asynchronous systems 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, occurs 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 permanently correct w.r.t. resource exhaustion, 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 'practical intrusion-tolerant asynchronous systems'.

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 $A_{t_{end}}$.

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 [15] (which can be seen as a form of dynamic replication [16]). The aim of this mechanism is conceptually simple – components are periodically rejuvenated to remove the effects

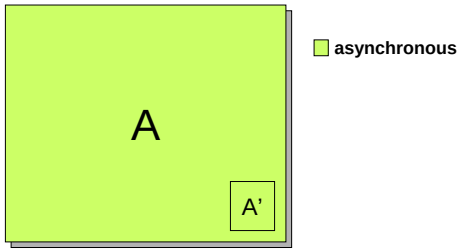

Fig. 1. A system A enhanced with a proactive recovery subsystem A'. Both A and A' run asynchronously.

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]. It may also be utilized to restore the system code from a secure source to eliminate potential transformation carried out by the adversary [15, 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). Therefore, proactive recovery allows a constant increase in $A_{t_{exhaust}}$, in particular in the parts of the system that are most vulnerable to attacks.

For simplicity let us imagine that whenever rejuvenation occurs the system is completely regenerated. A system with proactive recovery is represented as in Figure 1. 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. Subsystem A' periodically rejuvenates system A and, by doing so, indirectly increases $A_{t_{exhaust}}$. Therefore, apparently, by using a well planned strategy of proactive recovery, $A_{t_{exhaust}}$ can be constantly increased in order that resource exhaustion never happens before $A_{t_{end}}$.

Now let us consider subsystem A'. A' correctness is a necessary condition for A correctness, which means that it is mandatory that A' is also $\mathcal{PC}$. Since A' is asynchronous it is bound by Theorem 2. Consequently, A' must have an unbounded $A'_{t_{exhaust}}$, otherwise, it will not be $\mathcal{PC}$.

Several proactive recovery protocols for asynchronous systems proposed in the literature [11, 10, 8, 21, 3], despite having different goals, are all based on the same assumptions: periodic and timely execution. They assume that the proactive subsystem is regularly executed, and that the rejuvenation operation does not take a very long period to complete. In other words, these proactive recovery works make timing assumptions about the environment, which by definition, can be violated in an asynchronous setting. In this context, $A'_{t_{exhaust}}$ can be defined as the instant after which the assumptions of subsystem A' may be broken.

As explained in the previous section, if one considers malicious failures, we cannot assume that we can construct A' in a way that guarantees that $A'_{t_{exhaust}} \leq$

$A'_{t_{end}}$ never happens. This is because the adversary will attack the weakest point in the system, which in this case is A'.

Consequently, the asynchronous proactive recovery subsystem A' is only $\mathcal{TC}$, and for that reason it cannot permanently guarantee the correctness of the asynchronous system A. Thus:

- Asynchronous systems with a bounded $A_{t_{exhaust}}$ can only be $\mathcal{TC}$ even when enhanced with (asynchronous) proactive recovery subsystems.

To illustrate these conclusions in a real system, we will describe in the next section a possible attack to CODEX [14] that is based on the time-related vulnerability of the proactive recovery protocols it uses, predicted by our results under the $\mathcal{A}$ model 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 [14]. 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[1], which 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.

[1] 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 $f + 1$.

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 [14]. 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* [15], 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 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 attack is carried out by two adversaries, ADV1 and ADV2. ADV1 takes the system into a state where the actual attack can be performed by the second adversary. 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. As expected, both adversaries will execute the attacks without violating any of the assumptions presented in the CODEX paper.

Attack by adversary ADV1: ADV1 performs a mobile virus attack against $f+1$ servers. However, instead of retrieving the CODEX private key share of each node, it adjusts, one after the other, the drift 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[2]. 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, since clocks are slow, the actual APSS interval is much larger than expected. For all practical considerations, there is no time constraint since the clocks were delayed by a helping accomplice – ADV1.

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 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, 13, 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. These systems evolve at an arbitrary

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

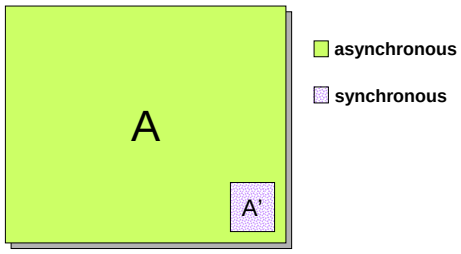

Fig. 2. A system A enhanced with a proactive recovery subsystem A'. A runs asynchronously, but A' runs synchronously in the context of a local time wormhole.

pace, while proactive recovery has natural timeliness requirements: proactive recovery leverages the defenses of a system by *periodically* "removing" the work of an attacker. 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 (certainly not the only) solution to the problem of ensuring permanently correct ($\mathcal{PC}$) 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 [17]. 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 area [5] and another for the time domain [18].

Remember that as explained in Sections 3.2 and 3.3, it is impossible to guarantee the permanent correctness of an asynchronous system A when $A_{t_{exhaust}}$ has a bounded value, even with an asynchronous proactive recovery scheme. 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.

The Trusted Timely Computing Base (TTCB) wormhole [5] deals with the problem of handling application timeliness requirements in insecure environments with loose real-time guarantees. A representation of a system with a TTCB wormhole is depicted in Figure 2. This wormhole offers, among others, the following *Timely Execution* service [18]:

Timely Execution Service *Given any function func with an execution time bounded by T, the TTCB is able to execute func within T from the execution start instant.*

This service could be clearly used to timely execute proactive recovery protocols. The feasibility of building such a service in a real system is confirmed by the already available implementation[3] of the TTCB for the RTAI [4] operating system.

5 Conclusions and Future Work

This paper has made a discussion about the actual dependability of synchronous and asynchronous systems. We showed that it is impossible to have permanently correct 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 introduced a system model expressive enough to represent these problems, and a classification of system correctness based on the predicate *exhaustion-safe*, meaning freedom from resource exhaustion.

Based on it, we predicted the extent to which fault/intrusion-tolerant distributed systems (synchronous and asynchronous) can be made to work correctly. Namely, we explained why proactive recovery has limitations when used in the context of asynchronous systems 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 this 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.

References

1. G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 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, November 2002.
4. P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper. In *Real-Time Linux Workshop*, November 2000.

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

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, October 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, April 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. Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. 1990.
13. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
14. 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.
15. 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.
16. D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation (2nd Edition)*. Digital Press, 1992.
17. 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.
18. P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, August 2002.
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, November 2002.
21. L. Zhou, F. B. Schneider, and R. van Renesse. Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Cornell University, Ithaca, New York, October 2002.