Proactive Resilience through Architectural Hybridization

Paulo Sousa
Nuno Ferreira Neves
Paulo Veríssimo

DI–FCUL TR–05–8

May 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.

Proactive Resilience through Architectural Hybridization*

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. Recently, we presented a theoretical Physical System Model (PSM), which introduced a new dimension over which distributed systems resilience may be evaluated – exhaustion-safety. We showed that it is theoretically impossible to have an exhaustion-safe f fault/intrusion-tolerant asynchronous system, even when enhanced with asynchronous proactive recovery. This paper proposes *proactive resilience* as a new and more resilient approach to proactive recovery based on architectural hybridization. We present the Proactive Resilience Model (PRM) and describe a design methodology under the PRM. This design methodology is formally proved to be a way of building exhaustion-safe systems and we use it to derive an exhaustion-safe distributed f fault/intrusion-tolerant secret sharing system.

1 Introduction

Recently, we presented a theoretical Physical System Model (PSM), that takes into account the environmental resources and their evolution along the timeline of system execution [15]. The model builds on the concept of *resource exhaustion* – the situation when a system no longer has the necessary resources to execute correctly (e.g., computing power, bandwidth, replicas). PSM allowed us to introduce the predicate *exhaustion-safe*, meaning freedom from *exhaustion-failures* – failures that result from accidental or provoked resource exhaustion.

Using PSM, it was possible to predict the extent to which fault/intrusion-tolerant distributed systems (synchronous [8, 18] and asynchronous [6, 11, 5]) can be made to work correctly. Namely, we showed that it is theoretically impossible to have exhaustion-safe f fault/intrusion-tolerant asynchronous systems, i.e., asynchronous systems which can only tolerate a bounded number of faults.

Having proven this impossibility result, the next step was trying to discover how to build exhaustion-safe systems. We found proactive recovery as being a very interesting approach [13] to avoid resource exhaustion. The aim of proactive recovery is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/failures. If the rejuvenation is performed

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

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 [10, 9, 7, 20, 2, 19, 12]. It may also be utilized to restore the system code from a secure source to eliminate potential transformations carried out by an adversary [13, 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 errors exploitable by outside attackers).

Therefore, proactive recovery has the potential to allow the construction of exhaustion-safe fault/intrusion-tolerant distributed systems. But, in order to allow this, proactive recovery needs to be architected under a model sufficiently strong to allow proactive recovery to achieve its goal: regular rejuvenation of the system. In this context, it was also shown in [15] that proactive recovery has some limitations when used under the asynchronous model [3, 19, 2, 12]. More concretely, we showed that it is theoretically impossible to have exhaustion-safe f fault/intrusion-tolerant asynchronous systems, even when they are enhanced with asynchronous proactive recovery. This impossibility result was illustrated in practice, through the identification of some shortcomings on existent asynchronous proactive recovery work [12]. In fact, proactive recovery protocols typically require stronger environment assumptions (e.g., synchrony, security) than the rest of the system (i.e., the part that is proactively recovered).

This paper proposes proactive resilience – a new and more resilient approach to proactive recovery based on architectural hybridization [16]. We argue that the architecture of a system enhanced with proactive recovery should be hybrid, i.e., divided in two parts: the "normal" payload system and the proactive recovery subsystem, the former being proactively recovered by the latter. Each of these two parts should be built under different timing and fault assumptions: the payload system may be asynchronous and vulnerable to arbitrary faults, and the proactive recovery subsystem should be constructed in order to assure a more synchronous and secure behavior. We present the generic Proactive Resilience Model (PRM), which proposes to model the proactive recovery subsystem as an abstract component – the Proactive Recovery Wormhole (PRW). The PRW may have many instantiations depending on the application/protocol proactive recovery needs (e.g., rejuvenation of cryptographic keys, restoration of system code). Then, a design methodology under the PRM is described and formally proved to be a way of building exhaustion-safe systems. Finally, we use this methodology to build an exhaustion-safe distributed f fault/intrusion-tolerant secret sharing system, which makes use of a specific instantiation of the PRW targetting the secret sharing scenario [1, 14].

The paper is organized as follows. Section 2 revisits and complements the PSM model proposed in [15] and summarizes the theoretical results obtained by applying it to asynchronous systems, when using and not using asynchronous proactive recovery. Then, Section 3 presents the Proactive Resilience Model and shows how to build exhaustion-safe systems based on a novel methodology. Finally, our conclusions and future work are presented in Section 4.

2 *PSM* Revisited

Distributed systems usually use in their implementation 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;
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.

PSM 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, through *PSM*, 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.

We start by defining the concept of resource exhaustion.

Definition 1. *Resource exhaustion occurs when any of the system resource assumptions is violated.*

A resource exhausted system is prone to what we designate as exhaustion-failures.

Definition 2. *An exhaustion-failure is a failure that results from accidental or provoked resource exhaustion.*

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

Definition 3. *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 4. *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.

2.1 The Model

The main goal of PSM is to allow a formal reasoning about how exhaustion-safety may be affected by different combinations of timing and fault assumptions. So, it takes in account the relevant system resources and their evolution with time. PSM 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: one 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.

Now we review the formal definition of PSM presented in [15].

Definition 5. *Let A be a system. An A execution is defined by a triple:* $\mathcal{A} = \langle \mathcal{A}_{t_{start}}, \mathcal{A}_{t_{end}}, \mathcal{A}_{t_{exhaust}} \rangle$, *where*

- $\mathcal{A}_{t_{start}} \in \Re_0^+$ *represents the real time start instant.*
- $\mathcal{A}_{t_{end}} \in [\mathcal{A}_{t_{start}}, +\infty[$ *represents the real time termination instant.*
- $\mathcal{A}_{t_{exhaust}} \in [\mathcal{A}_{t_{start}}, +\infty[$ *represents the real time instant when resource exhaustion occurs. If $\mathcal{A}_{t_{exhaust}} \leq \mathcal{A}_{t_{end}}$, system correctness may be corrupted through exhaustion-failures.*
- *Additionally, let $\mathcal{A}_{T_{end}} = \mathcal{A}_{t_{end}} - \mathcal{A}_{t_{start}}$, and $\mathcal{A}_{T_{exhaust}} = \mathcal{A}_{t_{exhaust}} - \mathcal{A}_{t_{start}}$.*

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

Definition 6 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.

The main advantage of this more expressive model is that condition $\mathcal{A}_{t_{end}} < \mathcal{A}_{t_{exhaust}}$ can be evaluated, that is, one 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 $\mathcal{A}_{t_{start}}, \mathcal{A}_{t_{end}}$ and $\mathcal{A}_{t_{exhaust}}$, but rather to reason about constraints imposed on them derived from environment assumptions. With this goal in mind, one crucial property of PSM follows immediately from the above definition.

Property 1. If $\mathcal{A}_{T_{exhaust}}$ has an upper bound $T_{exhaust_{max}}$ (i.e., $\mathcal{A}_{T_{exhaust}} \leq T_{exhaust_{max}}$, $\forall \mathcal{A}$), then A is exhaustion-safe only if $\mathcal{A}_{T_{end}} < T_{exhaust_{max}}, \forall \mathcal{A}$.

Property 1 states a necessary condition for exhaustion-safety, which allows to prove the following corollary regarding the exhaustion-safety of asynchronous systems.

Corollary 1. *If A is an asynchronous system (i.e., $\nexists T_{end} : \mathcal{A}_{T_{end}} \leq T_{end}, \forall \mathcal{A}$), and $\mathcal{A}_{T_{exhaust}}$ has an upper bound $T_{exhaust_{max}}$ (i.e., $\mathcal{A}_{T_{exhaust}} \leq T_{exhaust_{max}}, \forall \mathcal{A}$), then A is not exhaustion-safe.*

Proof. If $\mathcal{A}_{T_{end}}$ does not have a known bound, it is impossible to guarantee that $\mathcal{A}_{T_{end}} < T_{exhaust_{max}}, \forall \mathcal{A}$, and therefore, by Property 1, A is not exhaustion-safe.

This corollary shows that any asynchronous system with an upper bound on $\mathcal{A}_{T_{exhaust}}$ is not exhaustion-safe. Thus, it is impossible to have an exhaustion-safe f fault-tolerant asynchronous system, given that $\mathcal{A}_{T_{exhaust}}$ is upper bounded by a constant that depends on the time needed to occur $f+1$ faults.

This impossibility result induced us to research on how to build exhaustion-safe systems which could somehow maintain the nice characteristics of the asynchronous model. Proactive recovery was found as being a very interesting approach [13] to avoid resource exhaustion and therefore to enable exhaustion-safe operation. In the next section, we describe this approach and evaluate it under the light of PSM.

2.2 Proactive Recovery under PSM

Proactive recovery is based on periodic rejuvenation of components. The goal of this periodic rejuvenation is to remove the effects caused by accidental and/or malicious faults. In the case of malicious attacks, if the rejuvenation is performed frequently often, then an adversary is unable to corrupt enough resources to break the system. Proactive recovery can be used, for instance, to periodic refresh (all or part of) the state of an application, such as cryptographic keys [10, 9, 7, 20, 2, 19, 12]. Another application of proactive recovery can be the periodic restoration of system code from a secure source to eliminate potential transformations carried out by an adversary [13, 3].

A proactive recovery subsystem can be formally defined in the following way.

Definition 7. *Let R be a proactive recovery subsystem. An R execution is defined by the sequence of rejuvenations $\mathcal{R} = \langle \mathcal{R}^1, \mathcal{R}^2, ..., \mathcal{R}^n \rangle$. Each rejuvenation $\mathcal{R}^i, i \in \{1, ..., n\}$, is defined by the pair $\langle \mathcal{R}^i_{t_{start}}, \mathcal{R}^i_{t_{end}} \rangle$, where $\mathcal{R}^i_{t_{start}}$ represents the rejuvenation real time start instant and $\mathcal{R}^i_{t_{end}}$ represents the rejuvenation real time termination instant, such that, $\mathcal{R}^i_{t_{start}} \leq \mathcal{R}^i_{t_{end}}, \forall i$, and $\mathcal{R}^i_{t_{end}} \leq \mathcal{R}^{i+1}_{t_{start}}, \forall i < n$. $\#\mathcal{R}$ denotes the number n of rejuvenations performed during an execution $\mathcal{R}$.*

Under PSM, a proactively recovered system is defined in the following manner.

Definition 8. *Let A^* be a system A enhanced with a proactive recovery subsystem R. Let $\mathcal{A}^*$ represent an A^* execution, where $\mathcal{A}^*_{t_{start}} = \mathcal{A}_{t_{start}}$, $\mathcal{A}^*_{t_{end}} = \mathcal{A}_{t_{end}}$, and $\mathcal{A}^*_{t_{exhaust}} = \mathcal{A}_{T_{exhaust}} + D$, where the value D may have different values depending on the following conditions:*

- $D = \mathcal{A}^*_{t_{start}}$, *if* $\mathcal{R}^1_{t_{end}} - \mathcal{A}^*_{t_{start}} \geq \mathcal{A}_{T_{exhaust}}$;
- $D = \mathcal{R}^j_{t_{end}}$, *if*
  - $((j = \#\mathcal{R}) \vee (\mathcal{R}^{j+1}_{t_{end}} - \mathcal{R}^j_{t_{end}} \geq \mathcal{A}_{T_{exhaust}}))\wedge$
  - $((j = 1) \vee (\forall 1 < i \leq j, \mathcal{R}^i_{t_{end}} - \mathcal{R}^{i-1}_{t_{end}} < \mathcal{A}_{T_{exhaust}}))$.

Additionally, let $T_{rejuvenation}$ be a constant such that:

- $\mathcal{R}^i_{t_{end}} - \mathcal{R}^{i-1}_{t_{end}} \leq T_{rejuvenation}, \forall \mathcal{R}, \forall i < \#\mathcal{R}$;
- $\mathcal{R}^1_{t_{end}} - \mathcal{A}^*_{t_{start}} \leq T_{rejuvenation}, \forall \mathcal{R}, \forall \mathcal{A}^*$;
- $\mathcal{A}^*_{t_{end}} - \mathcal{R}^n_{t_{end}} \leq T_{rejuvenation}, \forall \mathcal{R}, \forall \mathcal{A}^*$.

Using this definition, one can prove sufficient and necessary conditions for the exhaustion-safety of a system enhanced with proactive recovery. We start with a sufficient condition.

Theorem 1. *Consider A^* as a system A enhanced with a proactive recovery subsystem R. Consider also that $\mathcal{A}_{T_{exhaust}}$ has a lower bound $T_{exhaust_{min}}$ (i.e., $\mathcal{A}_{T_{exhaust}} \geq T_{exhaust_{min}}, \forall \mathcal{A}$). Then, A^* is exhaustion-safe if $T_{rejuvenation} < T_{exhaust_{min}}$.*

Proof. By definition, $\mathcal{A}^*_{t_{exhaust}} = \mathcal{A}_{T_{exhaust}} + D$. $D \neq \mathcal{A}^*_{t_{start}}$ given that $\mathcal{R}^1_{t_{end}} - \mathcal{A}^*_{t_{start}} \leq T_{rejuvenation} < T_{exhaust_{min}} \leq \mathcal{A}_{t_{exhaust}}$. $D = \mathcal{R}^n_{t_{end}}$ given that $n = \#\mathcal{R}$ and $\mathcal{R}^i_{t_{end}} - \mathcal{R}^{i-1}_{t_{end}} \leq T_{rejuvenation} < T_{exhaust_{min}} \leq \mathcal{A}_{t_{exhaust}}, \forall 1 < i \leq n$. Thus, $\mathcal{A}^*_{t_{exhaust}} = \mathcal{A}_{T_{exhaust}} + \mathcal{R}^n_{t_{end}}$. By hypothesis, $\mathcal{A}^*_{t_{exhaust}} \geq T_{exhaust_{min}} + \mathcal{R}^n_{t_{end}} > T_{rejuvenation} + \mathcal{R}^n_{t_{end}} \geq \mathcal{A}^*_{t_{end}}$. Therefore, A^* is exhaustion-safe.

This sufficient condition will be used in Section 3 to show that our proactive resilience approach can be used to design an exhaustion-safe system.

A necessary condition for the exhaustion-safety of a system enhanced with proactive recovery is presented next.

Theorem 2. *Consider A^* as a system A enhanced with a proactive recovery subsystem R. Consider also that $\mathcal{A}_{T_{exhaust}}$ has an upper bound $T_{exhaust_{max}}$ (i.e., $\mathcal{A}_{T_{exhaust}} \leq T_{exhaust_{max}}, \forall \mathcal{A}$). Then, A^* is exhaustion-safe only if $min(\mathcal{A}_{T_{end}}, T_{rejuvenation}) < T_{exhaust_{max}}, \forall \mathcal{A}$.*

Proof. In order to prove by contradiction, let us assume that $\exists \mathcal{A} : T_{exhaust_{max}} \leq min(\mathcal{A}_{T_{end}}, T_{rejuvenation})$ and A is exhaustion-safe.

If $min(\mathcal{A}_{T_{end}}, T_{rejuvenation}) = \mathcal{A}_{T_{end}}$, then A is not exhaustion-safe by Property 1, which contradicts the hypothesis.

If $min(\mathcal{A}_{T_{end}}, T_{rejuvenation}) = T_{rejuvenation}$, then $T_{exhaust_{max}} \leq T_{rejuvenation} \Rightarrow \mathcal{A}_{T_{exhaust}} \leq T_{rejuvenation}$. Given that $T_{rejuvenation} \leq \mathcal{A}_{T_{end}}$, A is not exhaustion-safe, which also contradicts the hypothesis.

From this theorem it immediately follows that asynchronous proactive recovery does not guarantee the exhaustion-safety of a f fault-tolerant asynchronous system. More generally, the following corollary shows that asynchronous proactive recovery does not guarantee the exhaustion-safety of any asynchronous system with an upper bound on $\mathcal{A}_{t_{exhaust}}$.

Corollary 2. *Consider A^* as an asynchronous system A (i.e., $\nexists T_{end} : \mathcal{A}_{T_{end}} \leq T_{end}, \forall \mathcal{A}$) enhanced with an asynchronous proactive recovery subsystem R. Consider also that $\mathcal{A}_{T_{exhaust}}$ has an upper bound $T_{exhaust_{max}}$ (i.e., $\mathcal{A}_{T_{exhaust}} \leq T_{exhaust_{max}}, \forall \mathcal{A}$). Then, A^* is not exhaustion-safe.*

Proof. Theorem 2 states that A^* is exhaustion-safe only if $min(\mathcal{A}_{T_{end}}, T_{rejuvenation}) < T_{exhaust_{max}}, \forall \mathcal{A}$. Given that $\mathcal{A}_{T_{end}}$ is unbounded, $min(\mathcal{A}_{T_{end}}, T_{rejuvenation}) = T_{rejuvenation}$. If R is asynchronous, then it has unbounded processing time and thus it is impossible to guarantee that $T_{rejuvenation} < T_{exhaust_{max}}$. Therefore, A^* is not exhaustion-safe.

3 An Architectural Hybrid Model for Proactive Recovery

The main difficulty with proactive recovery is not the concept but its implementation – this mechanism is useful to periodically rejuvenate components and remove the effects of malicious attacks/failures, as long as it has timeliness guarantees. In fact, the rest of the system may even be completely asynchronous – only the proactive recovery mechanism needs synchronous execution.

This type of requirement make us believe that one of the possible approaches to dependably use proactive recovery, is to execute it in the context of a wormhole: a subsystem capable of providing a small set of services, with good properties that are otherwise not available in the rest of the system [16]. Notice that these "good properties" may be of many different types. Until now we have only tackled certain aspects of the time and security scenarios, but it should be possible to apply the concept of wormholes to other domains. So, in the past, two incarnations of distributed wormholes have already been created, one for the security area [4] and another for the time domain [17]. 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 planned to guarantee that they have access to all required resources when needed, avoiding in this way resource exhaustion.

We propose the Proactive Resilience Model (PRM) – a more resilient approach to proactive recovery that is based on a wormhole-enhanced architecturally hybrid distributed system model. The PRM defines that the architecture of a system enhanced with proactive recovery should be hybrid, i.e., divided in two parts: the proactive recovery subsystem and the payload system, the latter being proactively recovered by the former. Each of these two parts should be built under different timing assumptions and with a different fault model.

The payload system executes the "normal" applications/protocols. Thus, the payload synchrony and fault model entirely depends on the applications/proto-

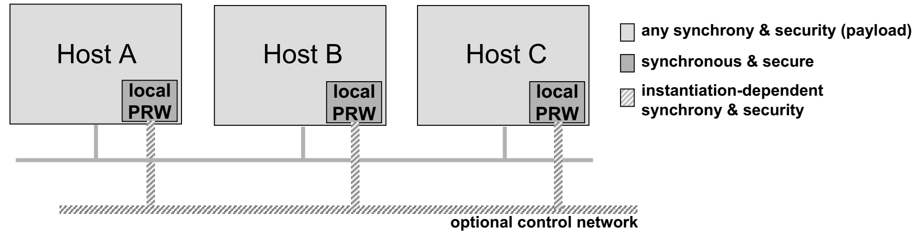

Fig. 1. The architecture of a system with a PRW

cols executing in this part of the system. For instance, the payload may operate under an asynchronous Byzantine environment.

The proactive recovery subsystem executes the proactive recovery protocols that rejuvenate the applications/protocols running in the payload part. This subsystem is more demanding, by definition, in terms of timing and fault assumptions, but some of these assumptions depend on the specific proactive recovery protocol, which can be of many types. Thus, we chose to model the proactive recovery subsystem as an abstract component which allows many instantiations. This abstract component is described in the next section. Then, in Section 3.2, we propose a design methodology to build exhaustion-safe distributed f fault/intrusion-tolerant systems, under the PRM. Finally, in Section 3.3, we conclude by instantiating both the abstract component and the methodology for a concrete application scenario.

3.1 The Proactive Recovery Wormhole

The Proactive Recovery Wormhole (PRW) is an abstract secure real-time distributed component that aims to execute proactive recovery procedures. By abstract we mean that the PRW allows many instantiations. Typically, an instantiation is chosen according to the concrete application/protocol that needs to be proactively recovered.

The architecture of a system with a PRW is suggested in Figure 1. An architecture with a PRW has a local module in some hosts, called the *local PRW*. Depending on the instantiation, these modules may or may not be interconnected by a *control network*. This set up of local PRWs optionally interconnected by the control network is collectively called *the* PRW. The PRW is used to execute proactive recovery procedures of protocols/applications running between participants in the hosts concerned, on any usual distributed system architecture (e.g., the Internet). We call the latter the *payload system and network*, to differentiate from the PRW part.

Conceptually, a local PRW should be considered to be a module inside a host, and separated from the OS. In practice, this conceptual separation between the local PRW and the OS can be achieved in several ways: (1) the local PRW can be implemented in a separate, tamper-proof hardware module (e.g., PC board) and so the separation is physical; (2) the local PRW can be implemented on

AN1 Broadcast – The AN has an unreliable packet broadcast primitive
AN2 Integrity – Nodes can detect if packets were corrupted in the network. Corruptions are converted to omission failures
AN3 Omission degree – No more than Od omissions may occur in a given interval of time
AN4 Bounded delay – Any correct packet is received within a maximum delay T_{send} from the send request
AN5 Partition free – The network does not get partitioned
AN6 Broadcast Degree – If a broadcast is received by any local PRW other than the sender, then it is received by at least Bd local PRWs
AN7 Confidentiality – The content of network traffic cannot be read by unauthorized users
AN8 Authenticity – Nodes can detect if a packet was sent by a correct node

Table 1. An example of a set of Abstract Network (AN) properties.

the native hardware, with a virtual separation and shielding implemented in software, between the former and the OS processes.

The local PRWs are assumed to be fail-silent (they fail by crashing). Every local PRW preserves, by construction, the following properties:

P1 There exists a known upper bound $T_{proc_{max}}$ on the processing delays;
P2 There exists a known upper bound $T_{drift_{max}}$ on the drift rate of local clocks.

As mentioned, a PRW instantiation may or may not have a control network. For instance, if a proactive recovery procedure only requires local information, then the control network is expendable. Even when the control network is required, its characteristics will depend on the specific requirements of the proactive recovery procedure. We specify the characteristics of the control network through abstract network properties. A specific control network will be characterized by the set of abstract network properties that it fulfils. Table 1 presents an example of a set of abstract network properties which can be used to characterize a PRW control network.

The PRW offers a single service: *periodic timely execution*. This service can be defined as follows:

Definition 9. *Given any function F, with a calculated worst case execution time of T_{Xmax}, an execution interval T_D, and a time interval (period) T_P, satisfying $T_{Xmax} < T_D < T_P$, then F is triggered by the PRW* ***periodic timely execution service*** *at real time instants t_i (the i-th triggering occurs at instant t_i), with $T_D < t_i - t_{i-1} \leq T_P$, and F terminates within T_D from $t_i, \forall i$.*

In short, the PRW has the ability to periodically execute well-defined functions in known bounded time. Moreover, the PRW allows the definition of a set of fail-safe measures to be triggered in certain situations. For instance, these fail-safe measures may shutdown the system if the *periodic timely execution* service fails to satisfy its specification. These self-checking mechanisms can then be used to prevent the occurrence of resource exhaustion even if the proactive recovery procedure fails to achieve its goal.

A PRW instantiation is defined by a triple $\langle D, \langle F, T_P, T_D\rangle, S\rangle$, such that:

- D represents the set of *data* which is proactively recovered, in all nodes (e.g., private key shares as in CODEX [12], system state and code as in BFT [3]);
- $\langle F, T_P, T_D\rangle$ represents the *function* F which is periodically triggered with period T_P and timely executed within T_D of each triggering, through the *periodic timely execution* service, in all nodes. F makes operations over the data defined in D;
- S represents the set of *self-checking* mechanisms, which have the goal of guaranteeing a fail-safe behaviour of all the nodes.

The feasibility of a specific PRW instantiation is assessed at design time. The system architect defines, at design time, the function F corresponding to the proactive recovery procedure to be executed, as long as, its required periodicity T_P and execution interval T_D. A PRW instantiation is feasible if T_P is greater than T_D and T_D is greater than the worst-case execution time of F.

3.2 Building Exhaustion-Safe Distributed f Fault/Intrusion-Tolerant Systems

In order to build an exhaustion-safe distributed f fault/intrusion-tolerant system, one has to guarantee that no more than f (accidental or malicious) faults occur during system execution. If the system maximum execution time is known, then one may choose a sufficient high f – by endowing the system with sufficient replicas – so that resource exhaustion never occurs. However, if the system has an unbounded execution time, we have a problem – it is not possible to estimate how many replicas will be needed to avoid resource exhaustion. One possible approach to solve this problem is to use the Proactive Resilience Model – enhance the system with a PRW in order that replicas are periodically and timely rejuvenated. Notice that this approach may even be applied in systems with a known bound on execution time when there is the need of minimizing the number of used replicas.

We propose a design methodology to build exhaustion-safe distributed f fault/intrusion-tolerant systems, under the Proactive Resilience Model. The methodology has 3 steps.

Definition 10. *The design methodology $\mathcal{M}_{exhaustion-safe}$ is defined by the following steps:*

1. *Define the data D to rejuvenate, the rejuvenation procedure F, the required periodicity T_P, the execution interval T_D, and the actions S to be performed if F is not executed with the required periodicity and execution time.*
2. *Build a PRW instantiation $\langle D, \langle F, T_P, T_D\rangle, S\rangle$.*
   - *If not feasible, increase the values of T_P and/or T_D.*
3. *Define the degree f_{safe} of fault-tolerance, such that, the minimum time necessary – $T_{exhaust_{min}}$ – for $f_{safe}+1$ faults to be produced satisfies the condition $T_{exhaust_{min}} > T_P + T_D$.*

The following theorem shows that a system built using this methodology is exhaustion-safe.

Theorem 3. *Consider A^* as a distributed f_{safe} fault/intrusion-tolerant system A enhanced with a proactive recovery subsystem R, both built according to the design methodology $\mathcal{M}_{exhaustion-safe}$. Consider also that $\mathcal{A}_{T_{exhaust}}$ has a lower bound $T_{exhaust_{min}}$ (i.e., $\mathcal{A}_{T_{exhaust}} \geq T_{exhaust_{min}}, \forall \mathcal{A}$), with $T_{exhaust_{min}}$ corresponding to the minimum time necessary for $f_{safe} + 1$ faults to be produced. Then, A^* is exhaustion-safe.*

Proof. Theorem 1 states that A^* is exhaustion-safe if $T_{rejuvenation} < T_{exhaust_{min}}$. Given that A^* is built using the design methodology $\mathcal{M}_{exhaustion-safe}$, then the PRW periodic timely execution service guarantees that $T_{rejuvenation} = T_P + T_D$ and step 3 of $\mathcal{M}_{exhaustion-safe}$ guarantees that $T_P + T_D < T_{exhaust_{min}}$. Therefore, A^* is exhaustion-safe.

3.3 The Proactive Secret Sharing Wormhole

Secret sharing schemes protect the secrecy and integrity of secrets by distributing them over different locations. A secret sharing scheme transforms a secret s into n shares $s_1, s_2, ..., s_n$ which are distributed to n share-holders. In this way, the adversary has to attack multiple share-holders in order to learn or to destroy the secret. For instance, in a $(k+1, n)$-threshold scheme, an adversary needs to compromise more than k share-holders in order to learn the secret, and corrupt at least $n-k$ shares in order to destroy the same secret.

Various secret sharing schemes have been developed to satisfy different requirements. In this paper we use Shamir's scheme [14] to implement a $(k+1, n)$-threshold scheme: given an integer valued secret s, pick a prime q which is bigger than both s and n. Randomly choose $a_1, a_2, ..., a_k$ from $[0, q[$ and set polynomial $f(x) = (s + a_1 x + a_2 x^2 + ... + a_k x^k) \ mod \ q$. For $i = 1, 2, ..., n$, set the share $s_i = f(i)$. The reconstruction of the secret can be done by having $k+1$ participants providing their shares and using polynomial interpolation to compute s.

In many applications, a secret s may be required to be held in a secret-sharing manner by n share-holders for a long time. If at most k share-holders are corrupted throughout the entire lifetime of the secret, any $(k+1, n)$-threshold scheme can be used. In certain environments, however, gradual break-ins into a subset of locations over a long period of time may be feasible for the adversary. If more than k share-holders are corrupted, s may be stolen. An obvious defense is to periodically refresh s, but this is not possible when s corresponds to inherently long-lived information (e.g., cryptographic root keys, legal documents).

Thus, what is actually required to protect the secrecy of the information is to be able to periodically renew the shares without changing the secret. Proactive secret sharing (PSS) was introduced in [10] in this context. In PSS, the lifetime of a secret is divided into multiple periods and shares are renewed periodically. In this way, corrupted shares will not accumulate over the entire lifetime of the

secret since they are checked and corrected at the end of the period during which they have occurred. A $(k+1, n)$ proactive threshold scheme guarantees that the secret is not disclosed and can be recovered as long as at most k share-holders are corrupted during each period, while every share-holder may be corrupted multiple times in some periods.

The Proactive Secret Sharing Wormhole (PSSW) is an instantiation of the PRW presented in Section 3.1. The PSSW targets distributed systems which are based on secret sharing and the goal of the PSSW is to periodically rejuvenate the secret shares of each system node.

The PSSW is defined by the triple $\langle D_{PSSW}, F_{PSSW}, S_{PSSW} \rangle$, such that:

- D_{PSSW} = { `share` }, where `share` is the secret share to be periodically refreshed.
- F_{PSSW} = ⟨ `refresh_share`, T_P, T_D⟩, where the concrete values of T_P and T_D depend on the requirements of the application/protocol, and `refresh_share` function is presented as Algorithm 1, and based on the share renewal scheme of [10]. Each process $P_i, i \in \{1..n\}$, executes Algorithm 1 at the beginning of the time periods defined by T_P. In lines 1–2, P_i picks k random numbers $\{\delta_{im}\}_{m \in \{1...k\}}$ in $[0, q[$. These numbers define the polynomial $\delta_i(z) = \delta_{i1}z^1 + \delta_{i2}z^2 + ... + \delta_{ik}z^k$. In lines 3–7, P_i sends the value $u_{ij} = \delta_i(j) \bmod q$ to all other correct servers P_j. Then, in lines 8–10, P_i receives the values u_{ji} from all other correct servers. These values are used to calculate, in line 11, the new share. Finally, in line 12, the old share is erased, namely any copy which could have been stored in the payload side. This action is vital in order to guarantee the effectiveness of proactive secret sharing. Reliable erasure could be implemented, for instance, through a reboot of the payload system.
- S_{PSSW} = {shutdown if share is not periodically and timely refreshed, as specified by T_P and T_D}.

The following assumptions are needed in order to prove the properties of Algorithm 1, which are presented more ahead in Theorem 4.

A1 There exists a known upper bound $T_{proc_{max}}$ on local processing delays;
A2 Nodes can detect if packets were corrupted in the network. Corruptions are converted to omission failures;
A3 No more than Od omissions may occur per sending node, in each execution of the algorithm;
A4 Any correct packet is received within a maximum delay $T_{send_{max}}$ from the send request;
A5 The content of network traffic cannot be read by unauthorized users;
A6 Nodes can detect if a packet was sent by a correct node;
A7 Nodes are fail-silent (they fail by crashing). At most $n - (k + 1)$ nodes are crashed in any time period. A crashed node does not recover. Nodes do not crash during the execution of the algorithm;
A8 Nodes have access to a perfect failure detector pfd, such that, $pfd(P_i) =$ *correct* iff P_i is not crashed, $\forall P_i$;

A9 There exists a known upper bound $T_{trigger_{max}}$ on the difference between the real time instants when the algorithm is triggered in all the nodes.

Algorithm 1 refresh_share() for each node $P_i, i \in \{1...n\}$

1: **for** $m = 1$ to k **do**
2: $\quad\delta_{im} \leftarrow$ generate_random_number($[0, q[$)
$\quad\{\{\delta_{im}\}_{m\in\{1...k\}}$ defines the polynomial $\delta_i(z) = \delta_{i1}z^1 + \delta_{i2}z^2 + ... + \delta_{ik}z^k\}$

$\quad\{$send $\delta_i(j)$ to each correct server $P_j\}$
3: **for** $j = 1$ to n **do**
4: $\quad$ **if** $j \neq i$ and $pfd(P_j) = correct$ **then**
5: $\quad\quad u_{ij} \leftarrow \delta_i(j) \bmod q$
6: $\quad\quad$ **for** $l = 1$ to $Od + 1$ **do**
7: $\quad\quad\quad$ send u_{ij} to P_j

$\quad\{$receive $\delta_j(i)$ from each correct server $P_j\}$
8: **for** $j = 1$ to n **do**
9: $\quad$ **if** $j \neq i$ and $pfd(P_j) = correct$ **then**
10: $\quad\quad$ receive u_{ji} from P_j

$\quad\{$calculate the new share and erase the old one$\}$
11: $share \leftarrow share + (u_{1i} + u_{2i} + ... + u_{ni}) (\bmod q)$
12: erase_old_share()

Assumptions A1–A6 are guaranteed by construction, given that Algorithm 1 is executed in the context of the PSSW. Assumption A1 is guaranteed by the property P1 of the PSSW. Assumptions A2–A6 are guaranteed by deploying a control network with the abstract network properties AN2, AN3, AN4, AN7 and AN8 presented in Table 1.

Assumption A7 is deliberately strong in terms of the type of faults considered. We could make a weaker assumption allowing Byzantine behaviour, recovery, and faults during execution, but this would result in a more complex and time-consuming algorithm (see [10]). Moreover, PSSW is secure by construction and therefore immune to Byzantine faults. Regarding the number of faults tolerated, the bound $n-(k+1)$ is simply to assure the liveness of the secret sharing scheme – the secret cannot be reconstructed with less than than $k + 1$ correct nodes.

The perfect failure detector assumed in A8 can be trivially implemented under assumptions A1–A4, which, as explained, are guaranteed by construction.

Assumption A9 requires the existence of a global clock, or some sort of triggering synchronization. These mechanisms can also be easily implemented under assumptions A1–A4.

Theorem 4. *If all correct nodes follow Algorithm 1 under assumptions A1–A9, then:*

Bounded execution time *There is an upper bound* $T_{exec_{max}}$ *on the difference between the real time instant when the first node starts executing the algorithm and the real time instant when the last node finishes executing it.*

Robustness *After all nodes finishing algorithm execution, the new shares computed correspond to the initial secret (i.e., any subset of* $k+1$ *of the new shares interpolate to the initial secret).*

Secrecy *An adversary that at any time knows no more than* k *shares learns nothing about the secret.*

Proof. Robustness and Secrecy are proved in [10].

Bounded execution time: Let I be the set of all instructions used in the algorithm. Let T_{exec_i} be a bound on the execution time of instruction $i, \forall i \in I$. The execution time of any instruction, with the exception of *receive*, depends only on the local processing delays, such that, $T_{exec_i} = c_i T_{proc_{max}}, \forall i \in I \setminus \{receive\}, c_i$ constants. The execution time of *receive* depends on the local processing delays, on $T_{trigger_{max}}$, and on $T_{send_{max}}$, such that, $T_{exec_{receive}} = c_{receive}(T_{proc_{max}} + T_{trigger_{max}} + T_{send_{max}}), c_{receive}$ constant. Thus, we can upper bound the local execution time of the algorithm by $T^{local}_{exec_{max}} = \sum_{i \in I} c_i T_{proc_{max}} + c_{receive} T_{trigger_{max}} + c_{receive} T_{send_{max}}$. Finally, $T_{exec_{max}} = T^{local}_{exec_{max}} + T_{trigger_{max}}$.

We now apply the methodology $\mathcal{M}_{exhaustion-safe}$, presented in the previous section, to build an exhaustion-safe distributed f fault/intrusion-tolerant secret sharing system:

1. $D = \{share\}, F = refresh_share(), T_P = c_p T_{exec_{max}}, T_D = c_d T_{exec_{max}}, c_p, c_d$ constants with $c_p > c_d > 1$, and $S = S_{PSSW}$.
2. Build the PSSW with the parameters defined in step 1. This PSSW is feasible given that $T_P > T_D$ and $T_D > T^{local}_{exec_{max}}$.
3. f_{safe} is chosen in order that $(c_p + c_d) T_{exec_{max}} < T_{exhaust_{min}}$.

Corollary 3. *Consider* A^* *as a distributed* f_{safe} *fault/intrusion-tolerant secret sharing system A enhanced with a proactive recovery subsystem R, executed under a PSSW defined by* $\langle D_{PSSW}, \langle$ `refresh_share`, $c_p T_{exec_{max}}, c_d T_{exec_{max}} \rangle, S_{PSSW} \rangle$. *Consider also that* $\mathcal{A}_{T_{exhaust}}$ *has a lower bound* $T_{exhaust_{min}}$ *(i.e.,* $\mathcal{A}_{T_{exhaust}} \geq T_{exhaust_{min}}, \forall \mathcal{A}$*), with* $T_{exhaust_{min}}$ *corresponding to the minimum time necessary for* $f_{safe}+1$ *faults to be produced. Then,* A^* *is exhaustion-safe.*

Proof. Given that f_{safe} and the PSSW are derived using methodology $\mathcal{M}_{exhaustion-safe}$, then, by Theorem 3, A^* is exhaustion-safe.

In each node, applications/protocols running in the payload get the current valid share through some interface function. We let the discussion of the appropriated interface as future work.

4 Conclusions and Future Work

In [15], we showed that it is theoretically impossible to have exhaustion-safe f fault/intrusion-tolerant asynchronous systems, i.e., asynchronous systems which

can only tolerate a bounded number of faults. These include systems that use asynchronous proactive recovery.

Based on this finding and on the fact that proactive recovery protocols typically require stronger environment assumptions (e.g., synchrony, security) than the rest of the system, in this paper, proactive resilience was proposed as a novel approach to proactive recovery that is based on an architectural hybrid distributed system model: the proactive recovery protocols are executed through a subsystem with "good" properties that are not available in the rest of the system.

The Proactive Resilience Model (PRM) was presented and we formally proved that there exists a design methodology under the PRM allowing to build exhaustion-safe systems. This methodology was applied to the secret sharing scenario in order to derive an exhaustion-safe distributed f fault/intrusion-tolerant secret sharing system.

As future work, we intend to implement an experimental prototype of the proposed secret sharing system, and to evaluate and compare, in practice, its actual resilience with the resilience of previous approaches. We also plan to apply the PRM and the methodology $\mathcal{M}_{exhaustion-safe}$ to different application scenarios.

References

1. G. R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the National Computer Conference*, volume 48 of *AFIPS*, pages 313–317. 1979.
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. 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.
5. 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.
6. 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.
7. J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theor. Comput. Sci.*, 243(1-2):363–389, 2000.
8. 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.
9. 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.

10. 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.
11. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
12. 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.
13. 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.
14. A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
15. P. Sousa, N. F. Neves, and P. Veríssimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, June 2005. to appear, http://www.navigators.di.fc.ul.pt/archive/sousa05how.pdf.
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, August 2002.
18. P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.
19. 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.
20. 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.