UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Proactive Resilience

Paulo Jorge Paiva de Sousa

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE CIÊNCIA DA COMPUTAÇÃO

2007

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Proactive Resilience

Paulo Jorge Paiva de Sousa

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE CIÊNCIA DA COMPUTAÇÃO

2007

Tese orientada pelo Prof. Doutor Paulo Jorge Esteves Veríssimo
e pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves

Abstract

This thesis introduces a new dimension over which systems dependability may be evaluated, *exhaustion-safety*. Exhaustion-safety means safety against resource exhaustion, and its concrete semantics in a given system depends on the type of resource being considered. The thesis focuses on the nodes of a fault-tolerant distributed system as crucial resources and on understanding the conditions in which the typical assumption on the maximum number of node failures may or may not be violated.

An interesting first finding was that it is impossible to build a *node-exhaustion-safe* intrusion-tolerant distributed system under the asynchronous model. This result motivated the research on developing the right model and architecture to guarantee node-exhaustion-safety. The main outcome of this research was *proactive resilience*, a new paradigm to build intrusion-tolerant distributed systems. Proactive resilience is based on architectural hybridization and hybrid distributed system modeling: the system is asynchronous in its most part and it resorts to a synchronous subsystem to periodically recover the nodes and remove the effects of faults/attacks. The Proactive Resilience Model (PRM) is presented and shown to be a way of building node-exhaustion-safe intrusion-tolerant distributed systems.

Finally, the thesis presents two application scenarios of proactive resilience. First, a proof-of-concept prototype of a *secret sharing* system built according to the PRM is described and shown to be highly resilient under different attack scenarios. Then, a novel intrusion-tolerant *state machine replication* architecture (based on the PRM) is presented and a new result established, that a minimum of $3f +$

$2k + 1$ replicas are required to ensure availability, on a system where f arbitrary faults may happen between recoveries, with at most k replicas recovering simultaneously.

Keywords: Distributed systems, dependability, security, intrusion-tolerance, availability.

Resumo

Esta tese introduz uma nova dimensão segundo a qual a confiabilidade dos sistemas pode ser avaliada, *segurança-contra-exaustão*. Segurança-contra-exaustão significa segurança contra exaustão de recursos e o seu significado concreto no contexto de um determinado sistema depende do tipo de recurso que se considere. A tese foca os nós de um sistema distribuído tolerante a faltas e estuda as condições em que o pressuposto típico sobre o número máximo de falhas de nós pode ou não ser violado.

Um primeiro resultado interessante foi a conclusão de que é impossível, sob o modelo assíncrono, construir-se um sistema distribuído tolerante a intrusões *seguro-contra-exaustão de nós*. Este resultado motivou a investigação para desenvolver o modelo e arquitectura adequados para garantir segurança-contra-exaustão de nós. O maior fruto desta investigação foi a *resiliência proactiva*, um novo paradigma para a construção de sistemas distribuídos tolerantes a intrusões. A resiliência proactiva é baseada em hibridização da arquitectura e modelação híbrida de sistemas distribuídos: o sistema é maioritariamente assíncrono e faz uso de um subsistema síncrono para recuperar periodicamente os nós e remover os efeitos das faltas/ataques. O Modelo de Resiliência Proactiva (PRM) é apresentado e é demonstrado que permite construir sistemas distribuídos tolerantes a intrusões seguros-contra-exaustão de nós.

Por fim, a tese apresenta dois cenários de aplicação da resiliência proactiva. Em primeiro lugar, é descrito um protótipo prova-de-conceito de um sistema de *partilha de segredos* construído de acordo com o PRM, e uma avaliação deste sistema permite concluir que é bastante resistente em diversos cenários de ataque. Em segundo

lugar, é apresentada um nova arquitectura (baseada no PRM) para *replicação de máquina de estados* tolerante a intrusões. É estabelecido um novo resultado de que um mínimo de $3f + 2k + 1$ réplicas são necessárias para garantir disponibilidade, num sistema onde f faltas arbitrárias possam acontecer entre recuperações, com um máximo de k réplicas a recuperar ao mesmo tempo.

Palavras Chave: Sistemas distribuídos, confiabilidade, segurança, tolerância a intrusões, disponibilidade.

Resumo Alargado

Hoje em dia, e cada vez mais, a confiabilidade dos sistemas informáticos é um assunto importante, dado que os computadores estão a invadir as nossas vidas, criando uma dependência cada vez maior no seu correcto funcionamento.

Informalmente, diz-se que um sistema é *confiável* se existe uma probabilidade alta de se comportar de acordo com a sua especificação. Quando o comportamento de um sistema viola a sua especificação, dizemos que ocorre uma *falha*. Construir sistemas confiáveis é construir sistemas onde se evita que as falhas aconteçam. Para se ter sucesso, é necessário perceber primeiro o processo que leva à falha, e que tipicamente é desencadeado por uma causa interna ou externa, denominada *falta*. As faltas são assim os alvos naturais de muitos dos mecanismos existentes para se obter confiabilidade, tais como: previsão de faltas, remoção de faltas, prevenção de faltas.

Claro que não é possível garantir que nenhuma falta vá ocorrer durante a operação do sistema. Logo, é necessário criar mecanismos complementares que bloqueiem o efeito da falta antes desta gerar uma falha. Quando este tipo de mecanismos está presente, diz-se que o sistema é capaz de oferecer um serviço correcto apesar da ocorrência de uma ou mais faltas, ou, em outras palavras, diz-se que o sistema é *tolerante a faltas*.

Durante a concepção de um sistema tolerante a faltas, são feitos pressupostos sobre o ambiente em que o sistema irá executar. Nomeadamente, o arquitecto de sistema faz pressupostos sobre o tipo e número de faltas que podem acontecer e sobre o comportamento temporal dos vários componentes do sistema. Neste contexto, um objectivo fundamental para se ter confiabilidade é garantir-se que durante a execução do sistema, o número *real* de faltas nunca exceda o número máximo de faltas f que se decide tolerar em tempo de projecto. Em termos práticos, o arquitecto de sistema deveria prever o número máximo de faltas N_f possíveis de acontecer durante a execução do sistema, de forma a que o mesmo fosse concebido para tolerar pelo menos $f \geq N_f$ faltas. O presente trabalho mostra que a dificuldade de se atingir este objectivo

varia não só com o tipo de faltas considerado, mas também com os pressupostos temporais. Para além disso, os modelos de sistema usados actualmente escondem parte destas dificuldades, uma vez que não são suficientemente expressivos. A tese propõe um novo modelo teórico de sistemas (REX), suficientemente expressivo para representar esses problemas. O modelo REX introduz uma nova dimensão segundo a qual a confiabilidade dos sistemas pode ser avaliada, *segurança-contra-exaustão*. Segurança-contra-exaustão significa segurança contra exaustão de recursos e o seu significado concreto no contexto de um determinado sistema depende do tipo de recurso que se considere. A tese foca os nós de um sistema distribuído tolerante a faltas e estuda as condições em que o pressuposto típico sobre o número máximo de falhas de nós pode ou não ser violado. Um sistema distribuído tolerante a intrusões *seguro-contra-exaustão de nós* é um sistema que garantidamente não sofre mais do que o número máximo assumido de falhas de nós.

Um primeiro resultado interessante deste trabalho foi a conclusão de que é impossível, sob o modelo assíncrono (isto é, não fazendo qualquer pressuposto temporal sobre o tempo de processamento local e o tempo de entrega de mensagens pela rede), construir-se um sistema distribuído tolerante a intrusões seguro-contra-exaustão de nós. Este resultado motivou a investigação no desenvolvimento do modelo e arquitectura adequados para garantir segurança-contra-exaustão de nós. No contexto desta investigação, percebeu-se que trabalhos anteriores já tinham proposto uma abordagem, denominada *recuperação proactiva*, com objectivos similares. A recuperação proactiva, que pode ser vista como uma forma de redundância dinâmica, tem o potencial de permitir a construção de sistemas tolerantes a intrusões seguros-contra-exaustão de nós. O objectivo da recuperação proactiva é conceptualmente simples: os nós são rejuvenescidos periodicamente para que sejam removidos os efeitos de faltas/ataques que tenham entretanto ocorrido. Se os rejuvenescimentos ocorrerem a um ritmo adequado, então um adversário é incapaz de corromper nós suficientes (isto é, mais do que aqueles que o arquitecto do sistema assumiu poderem ser corrompidos) para comprometer o sistema distribuído. No entanto, para dar estas garantias, a recuperação proactiva precisa de ser concebida segundo um modelo suficientemente forte que lhe permita atingir

o seu objectivo: o rejuvenescimento regular do sistema. Nenhum dos trabalhos existentes consegue garantir este objectivo. A tese analisa os problemas particulares dos diferentes trabalhos existentes e apresenta uma generalização destes problemas. Mais concretamente, os problemas dos sistemas distribuídos tolerantes a intrusões que usam recuperação proactiva são categorizados em quatro classes:

1. Um adversário malicioso pode ser mais forte (ter mais potência) do que o originalmente assumido e corromper os nós a um ritmo mais rápido do que as recuperações.

2. Um adversário malicioso pode tentar atrasar o ritmo das recuperações, de forma a aumentar as hipóteses de comprometer o sistema com a potência disponível.

3. Um adversário malicioso pode fazer ataques camuflados às referências temporais do sistema, o que em sistemas assíncronos ou parcialmente síncronos pode até nem ser detectado pela lógica essencialmente atemporal do sistema, deixando-o indefeso.

4. Os procedimentos de recuperação podem fazer com que os nós fiquem num estado temporariamente inactivo, baixando o quorum de redundância e a resiliência do sistema.

O primeiro problema (violação dos pressupostos sobre a potência do adversário) está fora do âmbito da tese, e é um problema irresolúvel, uma vez que envolve pressupostos fundamentais na área de investigação da tolerância a intrusões. Este problema pode no entanto ser combatido com técnicas que mitiguem as possibilidades de o adversário ganhar vantagem de uma forma não prevista, tais como, diversidade, mutação, ofuscação, ou componentes seguros. Todos os restantes problemas são resolvidos pela abordagem proposta nesta tese, *resiliência proactiva*, um novo paradigma para a construção de sistemas distribuídos tolerantes a intrusões, que permite usufruir de todas as potencialidades da recuperação proactiva. A resiliência proactiva é baseada num

modelo e arquitectura híbridos: o sistema (distribuído) é maioritariamente assíncrono e faz uso de um subsistema síncrono para recuperar periodicamente os nós e remover os efeitos das faltas/ataques. A tese descreve e avalia o modelo genérico de resiliência proactiva (PRM), que por sua vez modela o subsistema de recuperação como um componente abstracto denominado PRW. O componente PRW pode ter várias instâncias, dependendo dos requisitos do protocolo de recuperação que for adequado em cada caso (por exemplo, rejuvenescimento de chaves criptográficas, restauro do código do sistema operativo e/ou das aplicações). A tese demonstra que o PRM permite construir sistemas distribuídos tolerantes a intrusões seguros-contra-exaustão de nós.

Por fim, a tese apresenta dois cenários de aplicação da resiliência proactiva. Em primeiro lugar, é descrito um protótipo prova-de-conceito de um sistema de *partilha de segredos* construído de acordo com o PRM e fazendo uso de uma instância específica do componente PRW. Uma avaliação deste sistema permite concluir que é bastante resistente em diversos cenários de ataque. Nomeadamente, o protótipo desenvolvido é tolerante a intrusões e pode ser configurado para tolerar qualquer número de intrusões desde que a taxa de intrusões não seja maior do que uma intrusão por segundo[1]. Em segundo lugar, é apresentada um nova arquitectura (baseada no PRM) para *replicação de máquina de estados* tolerante a intrusões. Esta arquitectura usa outra instância do componente PRW para remover periodicamente os efeitos das faltas/ataques das réplicas. É feito um estudo quantitativo do nível de redundância necessário para se conseguir ter replicação de máquina de estados resiliente e disponível, e neste contexto é estabelecido um novo resultado: um mínimo de $3f + 2k + 1$ réplicas são necessárias para tolerar f faltas arbitrárias entre recuperações, com um máximo de k réplicas a recuperar ao mesmo tempo. Para comprovar a importância de assegurar a existência do número mínimo de réplicas, ditado por este resultado, foi feita uma avaliação por simulação. A avaliação incidiu sobre a resiliência e disponibilidade reais de um sistema genérico com replicação de máquina de estados construído de acordo com a arquitectura proposta e com um mínimo de $3f + 2k + 1$ réplicas. Paralelamente, efectuou-se o mesmo

[1]Note-se que não estamos a falar de ataques, mas sim de ataques bem sucedidos, ou intrusões.

tipo de avaliação a sistemas de replicação de máquina de estados propostos em trabalhos anteriores e que usam apenas $3f + 1$ réplicas. Uma das principais conclusões é que, ao invés da arquitectura proposta nesta tese, as propostas anteriores não conseguem garantir disponibilidade em vários cenários realistas, apresentando elevados tempos de indisponibilidade, especialmente na presença de adversários com uma potência próxima do originalmente assumido e/ou quando as recuperações têm tempos de execução elevados.

Acknowledgements

Many were those who contributed to strengthen the quality of the PhD work reported in this thesis. I would like to thank:

- My advisors Prof. Paulo Veríssimo and Prof. Nuno Neves for being precious guides and friends. I learned a lot with them. A special word for Prof. Paulo Veríssimo, who motivated (and continues to motivate) me to pursue a research career.
- Prof. António Casimiro and Prof. Antónia Lopes for being always ready to help me.
- My current and past colleagues at Navigators and LaSIGE for making research so enjoyable.
- My friends Bruno and Nando for being with me since our first breaths at the university.
- Maria Helena for the constant support.
- M. João for having motivated me to start the PhD.
- My son José for his genuine love for me. He constantly remembers me that there is life beyond work.
- My parents Isabel and António, my brothers Pedro and Tiago, and my sister Vanessa, for being my everyday companions. It would have been impossible to do the PhD journey without their support. Um beijo especial para a minha mãe, que nunca deixou de lutar e sempre conseguiu dar as melhores condições do mundo aos seus filhos. Por esta razão e por muitas mais, esta tese é-lhe dedicada.
- Mónica for her love and for being a fundamental inspiration in the last part of the PhD work.

À melhor mãe do mundo

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Contributions and Thesis Statement

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.

A system is *dependable* if it exhibits a high probability of behaving according to its specification. When the system behavior violates its service specification we say that a *failure* occurs. Building dependable systems is about preventing failures from occurring. However, to be successful, we must understand the process that leads to failures, which starts with an internal or external cause, called *fault*. The fault may remain dormant for a while, until it is activated. For example, a defect in a file system disk record is a fault that may remain unnoticed until the record is read. The corrupted record is an *error* in the state of the system that will lead to a failure of the file service when the disk is read. The failure is thus the externally observable effect of the error. The fault, error and failure definitions can be applied recursively when a system is decomposed into several components. That is, an error inside the system is often caused by the failure of one of its components, which at the system level should be seen as a fault. Figure 1.1 illustrates this recursion. Although we should avoid ambiguity when addressing the mechanisms of failure, it is thus possible to address

the same phenomenon as a (component) failure or a (system) fault, depending on the viewpoint. Likewise, interactions between components may fail in several ways (e.g., timing failure) constituting system-level faults that lead to an erroneous state (e.g., timing error). Faults may cause other faults. An error may give rise to other errors, by propagation, and in fact, a failure may be at the end of a chain of propagated errors (Avizienis *et al.*, 2004; Veríssimo & Rodrigues, 2001).

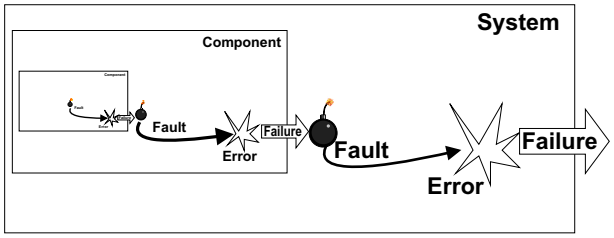

Figure 1.1: Fault, error and failure.

So, there is usually a cause-effect chain from a fault to a failure. To achieve dependability one should break this chain by applying methods that act at any point in the chain to prevent the failure from occurring. The source of the chain, the faults, are thus the natural targets of several means to achieve dependability. These means can be used in isolation or, preferably, in combination.

One of the approaches is called *fault removal*. It consists of detecting faults and removing them before they have the chance of causing an error. Targets of fault removal include software bugs, defective hardware, and so forth. *Fault forecasting* is the set of methods aiming at the estimation of the probability of faults occurring, or remaining in the system. Some classes of faults are easier to detect and remove than others. In consequence, fault forecasting can be seen as complementing fault removal, by predicting the amount of residual faults in the system. *Fault prevention*, as its name implies, consists of preventing the cause of errors by eliminating the conditions that make fault occurrence probable during system operation. For instance, using high quality components, components with internal redundancy, rigorous design techniques, etc. The

combination of fault prevention and removal is sometimes called *fault avoidance*, i.e., aiming at a fault-free system.

Of course, not all faults can be prevented from occurring during system operation, whereas other faults may even be present from the beginning of operation, having eluded fault removal. In consequence, one must create complementary mechanisms that block the effect of the fault before it generates a failure. In such case, we say that the system is capable of providing correct service despite the occurrence of one or more faults or, in other words, that the system is *fault-tolerant*.

During the design of a fault-tolerant system, assumptions are made about the environment where the system will execute. Namely, the system architect makes assumptions about the timing behavior of the system components and about the types of faults that can happen.

All else being equal, the dependability or trustworthiness of a system is inversely proportional to the number and strength of the assumptions made about the environment where the former executes. This applies to any type of assumptions, namely timing and fault assumptions. Moreover, the same assumption may have different strengths in different environments. For instance, consider the assumption "hackers will not try to compromise system correctness". This fault assumption is admittedly stronger if made in a system connected to the Internet than in a system operating only on a local network. In the same way, consider the assumption "message delivery delays never exceed 1 ms". This timing assumption is weaker when system nodes are all connected by a local network, than when the connection is over the Internet.

There are several flavors of synchrony models (Veríssimo & Rodrigues, 2001). Synchronous systems (Hadzilacos & Toueg, 1994) make timing assumptions, whereas asynchronous ones (Fischer *et al.*, 1985; Lynch, 1996) 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 (a few examples are (Ben-Or, 1983; Cachin *et al.*, 2002; Castro & Liskov, 2002; Chandra & Toueg, 1996; Chor & Dwork, 1989; Marsh & Schneider, 2004; Zhou *et al.*, 2002)).

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 (Schlichting & Schneider, 1983), omission (Dolev *et al.*, 1996, 1997), Byzantine (Lamport *et al.*, 1982)), or to the way a fault is produced (e.g., accidental or malicious, like vulnerability, attack, intrusion). The number establishes, in abstract, a notion of resilience (to f faults occurring). As such, current fault-tolerant system models feature a set of synchrony assumptions (or the absence thereof), and pairs $\langle type, number \rangle$ of fault assumptions (e.g., f omission faults; f intruded 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.

We now give the intuition of the problem. Consider a system where only accidental faults are assumed to exist. If it is synchronous, and its execution time is bounded, 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 does not have 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 predicting 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 (Cachin *et al.*, 2002; Castro & Liskov, 2002; Marsh & Schneider, 2004; Zhou *et al.*, 2002) 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 assuring that the actual number of faults in a system never exceeds the maximum num-

ber 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?

The findings presented in this thesis 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.

This being known, the thesis claim is the following:

To design a system architecture and devise the algorithmic underpinnings of protocols that can tolerate, through proactive resilience, any number of arbitrary faults over the lifetime of the system, as long as the fault rate does not exceed a predefined threshold, that depends on the specific system.

Therefore, our goal is to show that one can design dependable protocols — and thus, dependable systems — that are capable of tolerating any number of arbitrary faults over the lifetime of the system. The only constraint is on the rate of the faults: as expected, it cannot exceed a threshold defined at design time, and this threshold depends on the specific system being designed. We think that the existence of this constraint is obvious. Without it, the system would have to tolerate an unbounded number of arbitrary faults at each execution time instant, which is infeasible.

1.2 Research Methodology

Given the intuition about the dependability problems described in the previous section, and in order to prove the thesis claim, the research methodology that gave origin to the present thesis was composed of the following four tasks:

- Task T1: Definition of a more expressive model able to formally identify the problems described in the previous section;

- Task T2: Design and evaluation of a system model and architecture with the characteristics specified in the thesis claim;
- Task T3: Description of application scenarios where the proposed system model and architecture can be applied;
- Task T4: Implementation and evaluation of a proof-of-concept prototype for one of the described application scenarios.

The work produced as a result of the above tasks was validated through the publication of papers on international conferences of this area. The complete list of publications related to this research is presented next:

. **A New Approach to Proactive Recovery**
Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo
In Fifth European Dependable Computing Conference (EDCC-5) Supplementary Volume. Budapest, Hungary, pages 35-40, April 2005.

. **How Resilient are Distributed f Fault/Intrusion-Tolerant Systems?**
Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo
In Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05). Yokohama, Japan, pages 98-107, June 2005.

. **Resilient State Machine Replication**
Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo
In Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC). Changsha, China, pages 305-309, December 2005.

. **Proactive Resilience through Architectural Hybridization**
Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo
In Proceedings of the 2006 ACM Symposium on Applied Computing (SAC) - Volume 1. Dijon, France, pages 686-690, April 2006.

. **Proactive Resilience Revisited: the Delicate Balance Between Resisting Intrusions and Remaining Available**
Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo, William H. Sanders
In Proceedings of the 25th Symposium on Reliable Distributed Systems (SRDS'06). Leeds, United Kingdom, pages 71-80, October 2006.

. **Proactive Resilience**
Paulo Sousa
In Sixth European Dependable Computing Conference (EDCC-6) Supplementary Volume. Coimbra, Portugal, pages 27-32, October 2006.

. **On the Resilience of Intrusion-Tolerant Distributed Systems**
Paulo Sousa, Nuno Ferreira Neves, Antónia Lopes, Paulo Veríssimo
Submitted to IEEE Transactions on Dependable and Secure Computing.

. **Resilient and Available State Machine Replication**
Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo
Submitted to ACM Transactions on Computer Systems.

. **Protecting CRUTIAL Things**
Alysson Neves Bessani, Paulo Sousa, Miguel Correia, Nuno Ferreira Neves, Paulo Veríssimo
Submitted to DSN'07.

. **Hidden Problems of Asynchronous Proactive Recovery**
Paulo Sousa, Nuno Ferreira Neves, Paulo Veríssimo
Submitted to HotDep'07.

1.3 Overview

This section provides a synopsis for each of the remaining chapters of this thesis.

Chapter 2 describes the related work.

Chapter 3 introduces *exhaustion-safety* – a new dimension over which distributed systems resilience may be evaluated. A node-exhaustion-safe intrusion-tolerant distributed system is a system that assuredly does not suffer more than the assumed number of node failures: depending on the system, nodes may fail by crash, or start behaving in a Byzantine way, or disclose some secret information, and all these types of failures may be caused by accidents (e.g., a system bug), or may be provoked by malicious actions (e.g., an intrusion carried out by an hacker). We show that it is not possible to build *any* type of node-exhaustion-safe distributed f intrusion-tolerant system under the asynchronous model. In fact, we achieve a more general result, and show that it is impossible, under the asynchronous model, to avoid the exhaustion of any resource with bounded exhaustion time. Despite this general result, the focus on this work is on fault/intrusion-tolerant distributed systems and on node-exhaustion-safety.

Chapter 4 proposes and evaluates *proactive resilience* – a new and more resilient approach to proactive recovery based on hybrid distributed system modelling (Veríssimo, 2006) and architectural hybridization (Veríssimo, 2003). It is argued 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 describe and evaluate a generic Proactive Resilience Model (PRM), which proposes to model the proactive recovery subsystem as an abstract compo-

nent – the Proactive Recovery Wormhole (PRW). The PRW may have many instantiations according to the application/protocol proactive recovery needs (e.g., rejuvenation of cryptographic keys, restoration of system code). Then, it is shown that the *PRM* can be used to build node-exhaustion-safe intrusion-tolerant distributed systems.

Chapter 5 describes two examples of application scenarios where proactive resilience can be applied. Section 5.1 describes the design of a distributed f fault/intrusion-tolerant *secret sharing system*, which makes use of a specific instantiation of the PRW targeting the secret sharing scenario (Shamir, 1979). This system is shown to be node-exhaustion-safe and we built a proof-of-concept prototype that is highly resilient under different attack scenarios. Then, Section 5.2 describes a resilient f fault/intrusion-tolerant *state machine replication architecture*, which guarantees that no more than f faults ever occur. The architecture makes use of another instantiation of the PRW – the State Machine Proactive Recovery Wormhole – to periodically remove the effects of faults from the replicas. A quantitative assessment is performed of the level of redundancy required to achieve resilient and available state machine replication. In this context a new result is established, that a minimum of $3f + 2k + 1$ replicas are required for maintaining availability, on a system tolerating f Byzantine faults, with at most k replicas recovering simultaneously.

Chapter 6 presents the conclusions and future work.

Chapter 2

Related Work

2.1 The FLP Impossibility Result and Beyond

A distributed system built under the asynchronous model makes no timing assumptions about the operating environment: local processing and message deliveries may suffer arbitrary delays, and local clocks may present unbounded drift rates (Fischer *et al.*, 1985; Lynch, 1996). In other words, in a (purely) asynchronous system it is not possible to guarantee that something will happen before a certain time. Therefore, if the goal is to build dependable systems, the asynchronous model should only be used when system correctness does not depend on timing assumptions. At first sight, this conclusion only impacts the way algorithms are specified, by disallowing their dependence on time (e.g., timeouts) in asynchronous environments. However, as it will be showed in this thesis, other types of timing dependencies exist in a broader context orthogonal to algorithm specification. In brief, every system depends on a set of resource assumptions (e.g., a minimum number of correct replicas), which must be met in order to guarantee correct behavior. If a resource degrades more than assumed at system runtime, i.e., if the time until the violation of a

resource assumption is bounded, then it is not safe to use the asynchronous model because one cannot ensure that the system will terminate before the assumption is violated.

Consider now that we want to build a dependable intrusion-tolerant distributed system, i.e., a distributed system able to tolerate arbitrary faults, including malicious ones. In what conditions can one build such a system? Is it possible to build it under the asynchronous model?

This question was partially answered, twenty years ago, by Fischer, Lynch and Paterson (Fischer *et al.*, 1985), who proved that there is no deterministic protocol that solves the consensus problem is an asynchronous distributed system prone to even a single crash failure. This impossibility result (commonly known as FLP) has been extremely important, given that consensus lies at the heart of many practical problems, including membership, atomic commitment, leader election, and atomic broadcast. Considerable amount of research addressed solutions to this problem, trying to preserve the desirable independence from timing assumptions at algorithmic level. One could say that the question asked above was well on its way of being answered. However, the results presented in this thesis show that other (unexpected) dependencies on timing may exist.

Our results are orthogonal to FLP. Whereas FLP shows that a class of problems has no deterministic solution in asynchronous systems subject to failures, our results apply to any kind of problem in asynchronous distributed systems, independently of the specific goal/protocols of the system.

2.2 Proactive Recovery

Section 1.1 enumerated a set of works about protocols and systems that use proactive recovery with the goal of tolerating any number of faults over the lifetime of the system. In this section each of these works is analyzed in detail and the problems they face to achieve their goal are explained, with the intention of motivating our work. As the reader will note, we will not deal with classical vulnerabilities or flaws in the cited works, which are in essence excellent technical pieces. The problems are subtle and derive from the model used by these works. Afterwards, we summarize the four problems that may affect systems employing proactive recovery.

2.2.1 CODEX

CODEX (COrnell Data EXchange) is a recent distributed service for storage and dissemination of secrets (Marsh & Schneider, 2004). 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, fair links are assumed, which means that if a message is transmitted sufficiently often from one node to another, then it will eventually be delivered.

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.

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 in 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 Marsh & Schneider (2004). 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.

2.2.1.1 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

[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 $f + 1$.

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* (Ostrovsky & Yung, 1991), CODEX employs the APSS proactive secret sharing protocol (Zhou *et al.*, 2005). "This protocol is periodically executed, each time generating a new sharing of the private key but without ever materializing the private key at any server". Because older secret shares cannot be combined with new shares, the CODEX paper concludes that "a mobile virus attack would succeed only if it is completed in the interval between successive executions of APSS". This scenario can be prevented from occurring by running APSS regularly, in intervals that "can be as short as a few minutes".

2.2.1.2 An Example Attack

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

The intrusion campaign is carried out by two adversaries, ADV1 and ADV2. ADV1's attack takes the system into a state where the final attack can be performed by the second adversary. As expected, both adversaries will execute the attacks without violating any of the assumptions presented in the CODEX paper. ADV1 basically delays some parts of the system – it slows down some nodes and postpones the delivery of messages between two parts of the system (or temporally partitions the network). The reader should notice that after

this first attack the system will exhibit a behavior that could have occurred in any fault-free asynchronous system. Therefore, this attack simply forces the system to act in a manner convenient for ADV2, instead of having her or him wait for the system to naturally behave in such way.

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

APSS execution is triggered either by a local timer at each node or by a notification received from another node[1]. 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

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

execution in all of them.

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

Without any time constraint, it suffices to implement the mobile virus attack suggested in the CODEX paper, learning, one by one, $f + 1$ CODEX private key shares. The CODEX private key is disclosed using these shares. Using this key, ADV2 can decrypt the secrets stored in the compromised nodes. Moreover, she or he can get all new secrets submitted by clients through *write* operations.

The described attack explores one pitfall in the reasoning behind 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 (Fischer *et al.*, 1985; Lynch, 1996). 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.

2.2.2 COCA

COCA (Cornell Online Certification Authority) (Zhou *et al.*, 2002) is the predecessor of CODEX. COCA makes the same type of assumptions, namely it builds on the asynchronous model and uses APSS. However, there are two main differences between COCA and CODEX:

window of vulnerability assumption COCA assumes a bound on the maximum number of nodes (f) that can be compromised between consecutive rejuvenations;

server code and state recovery in COCA, not only the private key shares are refreshed through APSS, but also the server code and state are periodically rejuvenated.

Despite these two differences, the attack to CODEX described in the previous section may work unchanged against COCA. It all boils down to consider or not to consider each local clock as being part of the COCA servers state and specification. Let us look at this in more detail.

We start by analyzing the COCA definition of window of vulnerability. After it is introduced, the authors qualify it by saying that "Each window of vulnerability at a COCA server begins when that server starts executing the proactive recovery protocols and terminates when that server has again started and finished those protocols." So, according to this definition, a window of vulnerability is defined by local events. This introduces some ambiguity in the following: "at most t of the n COCA servers are ever compromised during each window of vulnerability, where $3t + 1 \leq n$ holds".

Independently from considering a local or distributed definition of window of vulnerability, our attack does not violate the assumption of COCA. This happens because during the attacks of adversary ADV1, various instances of

the proactive recovery protocols may run. So, we change the clock rate of $f+1$ servers, but never more than f servers between consecutive rejuvenations. The reader may ask how we manage to maintain servers compromised after various executions of proactive recovery protocols, specially if these protocols also rejuvenate the server state. The answer is simple: we do not really compromise any server. This will be more clear after a short explanation of what is considered to be a correct and compromised COCA server.

In COCA, "servers are either correct or compromised, where a compromised server might":

1. "stop executing";
2. "deviate arbitrarily from its specified protocols (i.e., Byzantine failure)";
3. "disclose information stored locally".

Our attack changes servers local clock rate, so it is clear that it is not included in 1 and 3. Regarding 2, the attack would be compromising if a clock with bounded drift was part of COCA specification. Authors do not talk in fact about clocks in COCA system model. However, let us assume this could be corrected and that it was assumed the existence of local clocks. Then the reader will note that, by necessity of the asynchronous model under which COCA is built, clocks will have unbounded drift rates (Fischer *et al.*, 1985; Lynch, 1996). Even if the model's asynchrony was relaxed to allow for actual physical clocks with bounded drift rate, the processes asynchrony would allow readings to be arbitrarily delayed, creating an effect analogous to unbounded drift rates, from the clocks readers' perspective (Veríssimo, 2006). So, local clocks may exist, but its *correctness* will not be compromised if one changes its rate: the specification of the clock allows unbounded drift rates, as seen by clock readers.

In consequence, a mobile clock-rate changing attack leaves the clocks correct, i.e., in accordance with COCA's assumptions, and as such *does not* count for the number of compromised servers. To make it clearer, "very slow" servers would still be correct according to COCA's model, be it because they are too loaded or because their clocks tick slowly. So, such an attack could legally slow down the clocks of $f + 1$ servers during a window of vulnerability, i.e., between consecutive rejuvenations, realizing the conditions for the final attack described earlier for CODEX.

COCA and CODEX authors argue that their proactive recovery protocols can be executed under the same asynchronous assumptions of the rest of the system. However, we showed above how this approach makes the COCA and CODEX systems vulnerable to clock attacks. The reasons for this will be clarified later in the thesis.

2.2.3 BFT and BFT-PR

Castro & Liskov (2002) proposed the first Byzantine-fault-tolerant (BFT) state machine replication algorithm not relying on any synchrony assumption to provide safety. Actually, BFT is proposed in two flavors. Firstly, authors present non-proactive BFT, which works under the asynchronous model and assumes that the number f of faulty replicas is bounded by $\lfloor \frac{n-1}{3} \rfloor$ during the entire lifetime of the system. Then, authors propose BFT with proactive recovery (BFT-PR), which can tolerate any number of faults provided fewer than 1/3 of the replicas become faulty within a supposedly small window of vulnerability corresponding to the interval between two consecutive recoveries. However, contrarily to COCA and CODEX, the BFT proactive recovery mechanism needs extra assumptions: secure cryptography, read-only memory and watchdog timers. Secure cryptography means that a replica can sign and de-

crypt messages without exposing its private key. Read-only memory is used both to store the public keys for other replicas and to store the code of the recovery monitor that executes the (proactive) recovery procedure. Watchdog timers are used to periodically interrupt processing and hand control to the recovery monitor. Therefore, each replica is equipped with a secure cryptographic coprocessor, a watchdog timer and a recovery monitor. It is also assumed that there is some unknown point in the execution after which either all messages are delivered within some constant time Δ or all non-faulty clients have received replies to their requests.

If these assumptions are satisfied, then BFT-PR works correctly. Namely, authors point out that an appropriate choice of Δ allows recoveries at a fixed rate. This suggests that the length T_v of the window of vulnerability can have a known bounded value in normal conditions.

So, contrarily to COCA and CODEX, BFT-PR explicitly makes an assumption about the need for a more synchronous component – the watchdog timer – that is able to guarantee the timely triggering of proactive recovery protocols. Thus, BFT-PR is immune to the attack that was described in Section 2.2.1. However, we argue that these assumptions are still not sufficient to guarantee the correct execution of the proactive recovery mechanism. Namely, we are concerned with the timeliness guarantees of the latter in a malicious environment.

The watchdog timer guarantees a timely periodic interrupt and can therefore be used to timely *trigger* periodic activities. However, the watchdog timer is a mere 'click': at best it only guarantees that the recovery monitor is timely started. The recovery monitor responsible for the recovery procedure, albeit stored in read-only memory and thus immune to modification faults, is executed as a normal task of the asynchronous system. So, in theory, the recovery

monitor, even if timely started, may take an unknown and very long interval to finish its execution. This can also easily happen in practice if BFT-PR is deployed in a malicious environment where its execution may be delayed indefinitely by an adversary. For instance, it is straightforward for a malicious adversary to increase the delivery time of recovery messages by the network. This can be done by corrupting the OS code that handles the reception of network messages. In the worst case scenario, the recovery mechanism executes during the entire lifetime of the system, never terminating and therefore never rejuvenating the system. Therefore, it is subject to the rules of the asynchronous environment, one of them being the non-existence of a known bound for processing. So, in theory, the recovery monitor, even if timely started, may take an unknown and very long interval to finish its execution.

BFT-PR authors admit the impossibility of ensuring a bound on T_v, but they only consider the scenario of a denial-of-service attack. In this scenario, replicas would be able to time recoveries and alert an administrator. However, an adversary able to compromise the clock rate, may simultaneously increase the value of T_v and prevent detection. Moreover, it is not explained how the interface between the synchronous watchdog timer and the asynchronous recovery monitor is secured. This is a very important part of the proactive recovery mechanism, given that a malicious adversary can neutralize this mechanism by simply breaking the synchronous-asynchronous interface. Such an attack would be equivalent to the time attack that allows to compromise COCA and CODEX.

These synchrony-related problems are better understood by observing Figure 2.1, which depicts the BFT-PR proactive recovery architecture, in terms of the synchronous guarantees each component requires. Note that the watchdog timer does not necessarily fit in the BFT asynchronous system model. The ba-

sic BFT algorithm is designed according to the asynchronous model and then, to allow proactive recovery, the watchdog timer is engineered as an auxiliary component, which provides some predefined synchronous actions. We believe that the watchdog timer, and more broadly the proactive recovery mechanism, should be considered at the system model level, and not as an engineering solution. Architecting BFT-PR under the same asynchronous system model of BFT plus some additional assumptions, renders BFT-PR with the same practical resilience of BFT: in a malicious environment, such as the Internet, the BFT-PR "small window of vulnerability" may be made as large as the lifetime of the system.

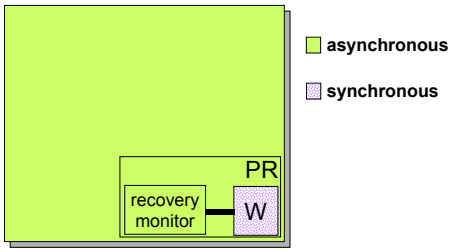

Figure 2.1: BFT-PR architecture: the proactive recovery mechanism (PR) is composed by a synchronous watchdog timer (W) that triggers an asynchronous recovery monitor.

Therefore, we hope to have shown that in order for BFT-PR to have a greater resilience factor than BFT, it must work under a different and adequate system model. BFT resists at most $\lfloor \frac{n-1}{3} \rfloor$ replica failures during the entire lifetime of the system, under an asynchronous system model. On the other hand, BFT-PR can only tolerate any number of faults provided fewer than $1/3$ of the replicas become faulty within a window of vulnerability. This is only possible if the window has a guaranteed time length, which in turn can only be achieved under a partially synchronous system model such as the one proposed in Chapter 4.

In summary, albeit periodically triggered, the BFT-PR proactive recovery mechanism can have an unbounded execution time, aggravated by the fact that an adversary may render impossible the detection of this anomalous behavior.

2.2.4 Asynchronous Proactive Cryptosystems

The process of building a fault-tolerant cryptosystem typically includes the use of threshold cryptography (Desmedt, 1998). The idea is that a cryptographic operation is performed by a group of n servers, such that an adversary who corrupts up to f ($f < n$) servers and observes their secret key shares can neither break the cryptosystem nor prevent the system as a whole from correctly performing the operation.

However, when a threshold cryptosystem operates over a longer time period, it may not be realistic (or safe) to assume that an adversary corrupts only f servers during the entire lifetime of the system. Proactive cryptosystems address this problem by operating in phases; they can tolerate the corruption of up to f different servers during every phase (Herzberg *et al.*, 1995). They rely on the assumption that servers may have a special reboot procedure that erases data and removes the adversary from a corrupted server. The idea is to proactively reboot all servers at the beginning of every phase, and to subsequently refresh the secret key shares such that in any phase, knowledge of shares from previous phases does not give the adversary any type of advantage. Therefore, proactive cryptosystems tolerate a mobile adversary (Ostrovsky & Yung, 1991), which may move from server to server and eventually corrupt every server in the system.

Cachin *et al.* (2002) introduce the idea of asynchronous proactive cryptosystems, i.e., proactive cryptosystems in asynchronous networks. Namely, they

make two different types of contributions:

1. On a conceptual level, they propose a formal model for cryptosystems in asynchronous proactive networks.

2. On a technical level, they present an efficient protocol for proactively refreshing discrete logarithm-based shares of a secret key.

In this section we focus on the first contribution. The formal model proposed for asynchronous proactive networks extends an asynchronous network by an *abstract timer* that is accessible to every server. The timer is scheduled by the adversary and defines the phase of a server locally. It is assumed that the adversary corrupts at most f servers who are in the same local phase.

When the adversary corrupts a server, she or he may provoke arbitrary changes to the server state. However, when a local phase begins, the server is rebooted from a correct state.

The *abstract timer* is formally modelled by a *trivial protocol timer* that works as follows: "Every honest server continuously runs one instance of this protocol, which starts when the server is initialized. Upon initialization, the protocol sends a timer message called a clock tick to itself. Whenever the server receives a clock tick, the server resends the message to itself over the network. The local phase of an uncorrupted server P_i is defined as the number of clock ticks that it has received so far." Therefore, it is not possible to upper-bound the maximum duration of a local phase, given that network messages do not have bounded delivery time in an asynchronous environment, namely if a malicious adversary is present.

However, in the implementation section it is assumed that neither asynchrony nor malicious actions may affect phase changes. Authors say that, in practice, the start of every local phase may be done through an impulse from

an external clock, and that an intruder must not be able to influence it. In other words, implicitly they make the same type of assumption done in BFT-PR: the existence of a watchdog that periodically triggers phase changes.

This assumption is not minor, since it is not compatible to the authors' claim of asynchrony. The formal model allows an adversary to control when local phases start and terminate, but this power is weakened in the discussion where they say that local phases may start when wanted, triggered by an *external impulse*.

Would this problem be solved by assuming that the system is in fact a partially synchronous system with watchdogs? As discussed in Section 2.2.3, even assuming that local phases are triggered by a watchdog, an adversary may still provoke problems. For instance, the adversary may attack the interface between the external clock and the reboot code such that all servers are always maintained in the same phase. In this case, the proactive cryptosystem becomes a normal cryptosystem living under the classic assumption that at most f servers may be corrupted during the entire lifetime of the system.

2.2.5 Problem Categorization

The previous sections showed, by example, the problems faced by a set of existing approaches to proactive recovery, despite being carefully conceived and designed. This points to the concept of proactive recovery having some compatibility problems with the asynchronous model, a fact that deserves a deeper study. Zhou *et al.* (2005) briefly discuss some of these problems, and conclude that the definition of the window of vulnerability in terms of events rather than the passage of time, can potentially afford attackers leverage. In fact, asynchronous systems evolve at an arbitrary pace, while proactive recovery has natural timeliness requirements: proactive recovery leverages the defenses of a sys-

tem by *periodically* "removing" the work of an attacker. This reasoning led us to think in what ways failures may happen in intrusion-tolerant systems employing proactive recovery. We found that the problems of intrusion-tolerant systems employing proactive recovery may be categorized in the following four classes:

1. A malicious adversary may deploy more power than originally assumed, and corrupt nodes at a pace faster than recovery;

2. It may attempt to slow down the pace of recovery, in order to leverage the chances of intruding the system with the available power;

3. It may perform stealth attacks on the system timing, which in asynchronous[1] or partially synchronous systems may not even be perceived by the essentially time-free logic of the system, leaving it defenseless;

4. Recovery procedures may make it necessary to bring individual nodes to a temporarily inactive state, lowering the redundancy quorum and thus system resilience.

The first problem (violation of attacker power assumptions) is outside the scope of this thesis, and it is fundamentally an unsolvable problem, since those assumptions are at the heart of the intrusion-tolerance body of research. It must be addressed with techniques that mitigate any leverage an attacker may unexpectedly try to get, such as diversity (Chen & Avizienis, 1978; Joseph & Avizienis, 1988; Littlewood & Strigini, 2004), obfuscation (Bhatkar *et al.*, 2003; Pucella & Schneider, 2006), or trusted components (Meyer, 2003), which

[1]Notice that it is possible to do attacks on the timing of asynchronous systems, that is, on the way they make progress according to real time. The difference between synchronous and asynchronous systems is that the safety of protocols running in the former may depend on timing guarantees, whereas it should not for protocols running in the latter.

can complement the approach we describe here. In this thesis, it is shown how an architecture and generic algorithmic approach to proactive recovery, globally designated *proactive resilience*, addresses each of the remaining problems. In certain conditions, our design methodology allows us to build resilient intrusion-tolerant services that never suffer more than the assumed number of faults and are always available. To our knowledge, this is the first time that solutions to these problems have been presented and analyzed, and the elimination of these problems drastically reduces the state-space of the attacker, as compared with previous work. In consequence, our results may have importance in generic on-line services, and even more in critical infrastructure settings.

Before delving into the description of our proactive resilience approach, in the next chapter we present a theoretical model that formally expresses the limitations of current approaches to proactive recovery.

Chapter 3

Exhaustion-Safety

3.1 Formalization

Typically, the correctness of a protocol depends on a set of assumptions regarding aspects like the type and number 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 in order 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 of bandwidth and latency). The violation of these resource assumptions may affect the safety and/or liveness of the protocol. If the protocol is vital for the operation of some system, then the system liveness and/or safety may also be affected.

To formally define and reason about exhaustion-safety of systems with regard to a given resource assumption, it is necessary to adopt a suitable model. Let φ_r be a resource assumption on a resource r. We consider models that define: (i) for every system S, the set of its executions $[\![S]\!] = \{\mathcal{E} : \mathcal{E} \text{ is an } S-\text{execution}\}$, which is a subset of the set $EXEC$ that contains all possible executions of any system (i.e., $[\![S]\!] \subseteq EXEC$); and (ii) a set $\models \varphi_r$, s.t. $\models \varphi_r \subseteq EXEC$

is the subset of all possible executions that satisfy the assumption φ_r. We shall use $\mathcal{E} \models \varphi_r$ to represent that the assumption φ_r is not violated during the execution $\mathcal{E}$.

In the context of such models, exhaustion-safety is defined straightforwardly.

Definition 3.1.1. *A system S is r-exhaustion-safe wrt φ_r if and only if $\forall \mathcal{E} \in [\![S]\!] : \mathcal{E} \models \varphi_r$.*

Notice that this formulation allows one to study the exhaustion-safety of a system for different types of assumptions φ_r on a given resource r.

3.1.1 The Resource Exhaustion Model

Our main goal is to formally reason about how exhaustion-safety may be affected by any different combinations of timing and fault assumptions. So, we need to conceive a model in which the impact of those assumptions can be analyzed. We call this model the Resource EXhaustion model (REX).

Our model considers systems that have a certain mission. Thus, the execution of this type of systems is composed of various processing steps needed for fulfilling the system mission (e.g., protocol executions). We define two intervals regarding the system execution and the time necessary to exhaust a resource, defined by: *execution time* and *exhaustion time*. The exhaustion time concerns a specific resource assumption φ_r on a specific resource r. Therefore, in what follows, $[\![S]\!]$ denotes the set of executions of a system S under REX for a fixed assumption φ_r on a specific resource r.

Definition 3.1.2. *A system execution $\mathcal{E}$ is a pair $\langle T^{\mathcal{E}}_{exec}, T^{\mathcal{E}}_{exh} \rangle$, where*

- $T^{\mathcal{E}}_{exec} \in \Re^{+}_{0}$ *and represents the total execution time;*

- $T^{\mathcal{E}}_{exh} \in \Re^{+}_{0}$ *and represents the time necessary, since the beginning of the execution, for assumption* φ_r *to be violated.*

The proposed notion of system execution captures the execution time of the system and the time necessary for assumption φ_r to be violated in a specific run. Notice that, in this way, one captures the fact that the time needed to violate a resource assumption may vary from execution to execution. For instance, if a system suffers upgrades between executions, its exhaustion time may be, consequently, increased or decreased.

Definition 3.1.3. *The assumption* φ_r *is not violated during a system execution* $\mathcal{E}$*, which we denote by* $\mathcal{E} \models \varphi_r$*, if and only if* $T^{\mathcal{E}}_{exec} < T^{\mathcal{E}}_{exh}$.

By combining Definitions 3.1.1 and 3.1.3, we can derive the definition of an r-exhaustion-safe system under REX.

Proposition 3.1.4. *A system* S *is* r*-exhaustion-safe wrt a given assumption* φ_r *if and only if* $\forall \mathcal{E} \in [\![S]\!] : T^{\mathcal{E}}_{exec} < T^{\mathcal{E}}_{exh}$.

This proposition states that a system is r-exhaustion-safe if and only if resource exhaustion (i.e., the violation of φ_r) does not occur during any execution. Notice that, even if the system is not exhaustion-safe, it 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 (i.e., a failure caused by resource exhaustion) will not happen. Figure 3.1 illustrates the differences between an execution of a (potentially) exhaustion-safe system and a "bad" execution of a non exhaustion-safe system. An exhaustion-safe system is immune to exhaustion-failures. A

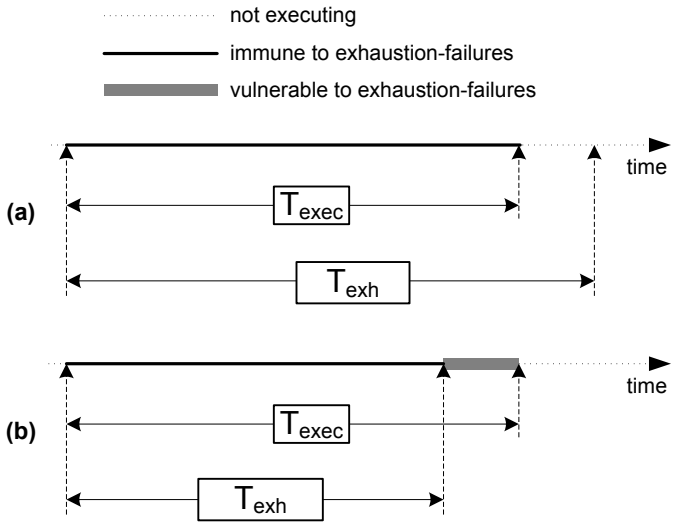

Figure 3.1: (a) An execution not violating φ_r; (b) An execution violating φ_r.

non exhaustion-safe system has at least one execution (such as the one depicted in Figure 3.1b) with a period of vulnerability to exhaustion-failures (the shaded part of the timeline) where the resource is exhausted and thus correctness may be compromised.

In a distributed fault-tolerant system, nodes are important resources, so important that one typically makes the assumption that a maximum number f of nodes can fail during its execution, and the system is designed in order to resist up to f node failures. This type of systems can be analyzed under the REX model, nodes being the resources considered, and the assumption φ_{node} being equal to $n_{fail} \leq f$, where n_{fail} represents the number of nodes that, during an execution, are failed at any time. In other words, this assumption means that no more than f nodes can be failed simultaneously.

Notice that in a system in which failed nodes do not recover, this assumption is equivalent to assuming that no more than f node failures can occur during the system execution. According to Proposition 3.1.4, a distributed

fault-tolerant system whose failed nodes do not recover is *node-exhaustion-safe* if and only if every execution terminates before the time needed for $f + 1$ node failures to be produced. In order to build a node-exhaustion-safe fault-tolerant system, one would like to forecast the maximum number of failures bound to occur during any execution, call it N_{fail}, so that the system is designed to handle at least $f = N_{fail}$ failures.

As Section 3.2 will show, the key aspect of the study of this model is that condition $T^{\mathcal{E}}_{exec} < T^{\mathcal{E}}_{exh}$ can be evaluated, that is, that we can determine whether it is maintained or not, depending on the type of system assumptions. Note that the idea is not to know the exact values of $T^{\mathcal{E}}_{exec}$ and $T^{\mathcal{E}}_{exh}$, but rather to reason about constraints that may be imposed on them, derived from environment and/or algorithmic assumptions. This way, one could predict, at system or even at algorithm design time, if the system can be exhaustion-safe according to the environment assumptions. This would allow us, as we shall show, to make propositions about exhaustion-safety for categories of algorithms and system fault and synchrony models, i.e., propositions about the potential resilience of algorithms long before systems are built. With this goal in mind, we start by defining two crucial properties of the model, which follow immediately from the previous definitions.

Property 3.1.5. *A* sufficient condition *for S to be r-exhaustion-safe wrt* φ_r *is*
$\exists T_{exec_{max}} \in \Re_0^+ (\forall \mathcal{E} \in [\![S]\!] : T^{\mathcal{E}}_{exec} \leq T_{exec_{max}}) \wedge (\forall \mathcal{E} \in [\![S]\!] : T^{\mathcal{E}}_{exh} > T_{exec_{max}})$

Property 3.1.6. *A* necessary condition *for S to be r-exhaustion-safe wrt* φ_r *is*
$\exists T_{exh_{max}} \in \Re_0^+ (\forall \mathcal{E} \in [\![S]\!] : T^{\mathcal{E}}_{exh} \leq T_{exh_{max}}) \Rightarrow (\forall \mathcal{E} \in [\![S]\!] : T^{\mathcal{E}}_{exec} < T_{exh_{max}})$

Property 3.1.5 states that a system S is r-exhaustion-safe wrt φ_r if there exists an upper-bound $T_{exec_{max}}$ on the system execution time, and if the exhaustion time of every execution is greater than $T_{exec_{max}}$.

Property 3.1.6 states that a system S can only be r-exhaustion-safe wrt φ_r if, given an upper-bound $T_{exh_{max}}$ on the system exhaustion time, the execution time of every execution is lower than $T_{exh_{max}}$.

3.2 Exhaustion-Safety vs Synchrony Assumptions

This section analyzes the impact of synchrony assumptions on the design of exhaustion-safe systems.

3.2.1 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 maximum delay, and components have access to local clocks with a known bounded drift rate with respect to real time (Hadzilacos & Toueg, 1994; Veríssimo & Rodrigues, 2001).

If one considers a synchronous system S with a bounded lifetime under REX, then it is possible to use the worst-case bounds defined during the design phase to assess the conditions of r-exhaustion-safety, for given r and φ_r.

Corollary 3.2.1. *If S is a synchronous system with a bounded lifetime $T_{exec_{max}}$ (i.e., $\forall \mathcal{E} \in [\![S]\!] : T^{\mathcal{E}}_{exec} \leq T_{exec_{max}}$) and $\forall \mathcal{E} \in [\![S]\!] : T^{\mathcal{E}}_{exh} > T_{exec_{max}}$, then S is r-exhaustion-safe wrt φ_r.*

Proof: Follows trivially from Property 3.1.5. ■

Therefore, if one wants to design an exhaustion-safe synchronous system with a bounded lifetime, then one has to guarantee that no exhaustion is pos-

sible during the limited period of time delimited by $T_{exec_{max}}$. For instance, and getting back to our previous example, in a distributed f fault-tolerant system this would mean that no more than f node failures should occur within $T_{exec_{max}}$.

Note that Corollary 3.2.1 only applies to synchronous systems with a bounded lifetime. A synchronous system may however have an unbounded lifetime. This seems contradictory at first sight and thus deserves a more detailed explanation. A synchronous system is typically composed by a set of (synchronous) rounds with bounded execution time (e.g., a synchronous server replying to requests from clients, each pair request-reply being a round). However, the number of rounds is not necessarily bounded. We consider that a synchronous system has a bounded lifetime if the number of rounds is bounded. Otherwise, the system has unbounded lifetime. If the system lifespan is unbounded, and T_{exh} is bounded, then we can prove the following.

Corollary 3.2.2. *If S is a synchronous system with an unbounded lifetime (i.e., $\nexists T_{exec_{max}} \in \Re_0^+, \forall \mathcal{E} \in [\![S]\!] : T_{exec}^{\mathcal{E}} \leq T_{exec_{max}}$) and $\exists T_{exh_{max}} \in \Re_0^+, \forall \mathcal{E} \in [\![S]\!] : T_{exh}^{\mathcal{E}} \leq T_{exh_{max}}$, then S is not r-exhaustion-safe wrt φ_r.*

Proof: If the set $\{T_{exec}^{\mathcal{E}} : \mathcal{E} \in [\![S]\!]\}$ does not have a bound, it is impossible to guarantee that $T_{exec}^{\mathcal{E}} < T_{exh_{max}}$, for every $\mathcal{E} \in [\![S]\!]$ and, therefore, by Property 3.1.6, S is not r-exhaustion-safe. ■

In fact, synchronous systems may suffer accidental or malicious faults. Such faults may have two bad effects: provoking timing failures that increase the expected execution time; causing resource degradation, e.g., node failures, which decrease T_{exh}. Notice that both these effects force the conditions of Corollary 3.2.2. Thus, in a synchronous system, an adversary can not only perform attacks to exhaust resources, but also violate the timing assumptions, even if during a limited interval, buying time for resources to be exhausted. In consequence, Corollary 3.2.2 formalizes and explains the current belief among the

research community that synchronous systems are fragile, and that secure systems should be built under the asynchronous model.

3.2.2 Asynchronous Systems

The distinguishing feature of an asynchronous system is the absence of timing assumptions, which means arbitrary delays for the execution of operations and message deliveries, and unbounded drift rates for the local clocks (Fischer *et al.*, 1985; Lynch, 1996). This model is quite attractive because it leads to the design of programs and components that are easier to port or include in different environments.

If one considers a distributed asynchronous system S under REX, then S can be built in such a way that termination is eventually guaranteed (sometimes only if certain conditions become true). However, it is impossible to determine exactly when termination will occur. In other words, the execution time is unbounded. Therefore, all we are left with is the relation between T_{exec} and T_{exh}, in order to assess whether or not S is r-exhaustion-safe, for given r and φ_r.

Can a distributed asynchronous system S be r-exhaustion-safe? Despite the arbitrariness of T_{exec}, the condition $T^{\mathcal{E}}_{exec} < T^{\mathcal{E}}_{exh}$ must always be maintained. Given that $T^{\mathcal{E}}_{exec}$ may have an arbitrary value, impossible to know through aprioristic calculations, the system should be constructed in order to ensure that, in all executions, $T^{\mathcal{E}}_{exh}$ is greater than $T^{\mathcal{E}}_{exec}$. This is very hard to achieve for some types of r and φ_r. An example is assuring that no more than f nodes ever fail. We provide a solution to this particular case in Chapter 4 based on a hybrid system architecture that guarantees exhaustion-safety through a partially synchronous subsystem that executes periodic rejuvenations.

If one assumes that the system is homogeneously asynchronous, and that

the set $\{T_{exh}^{\mathcal{E}} : \mathcal{E} \in [\![S]\!]\}$ is bounded, one can prove the following corollary of Property 3.1.6, similar to Corollary 3.2.2:

Corollary 3.2.3. *If S is an asynchronous system (and, hence, $\nexists T_{exec_{max}} \in \Re_0^+, \forall \mathcal{E} \in [\![S]\!] : T_{exec}^{\mathcal{E}} \leq T_{exec_{max}}$) and $\exists T_{exh_{max}} \in \Re_0^+, \forall \mathcal{E} \in [\![S]\!] : T_{exh}^{\mathcal{E}} \leq T_{exh_{max}}$, then S is not r-exhaustion-safe wrt φ_r.*

Proof: Follows trivially from Corollary 3.2.2. ■

This corollary is generic, in the sense that it applies to any type of system with a bounded T_{exh} for some assumption φ_r. However, its implications on distributed f fault-tolerant systems deserve a special look, given that in the remaining of the thesis, we concentrate on the exhaustion-safety of such systems.

Even though real distributed systems working under the asynchronous model have a bounded T_{exh} in terms of node failures, they have been used with success for many years. This happens because, until recently, only accidental faults (e.g., crash or omission) were a threat to systems. This type of faults, being accidental by nature, occur in a random manner. Therefore, by studying the environment in detail and by appropriately conceiving the system (e.g., estimate a conservative upper bound on T_{exec} i.e., one that applies to a large number of executions), one can achieve an asynchronous system that behaves as if it were exhaustion-safe, with as high a probability as we wish. That is, despite having the above-mentioned failure syndrome, it would be very difficult to observe it in practice.

However, when one starts to consider 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 (e.g., force the system to execute during more time than any estimated upper bound on

T_{exec}). In these conditions, having a upper-bounded T_{exh} (which we get when using a stationary maximum bound for node failures) most probably implies the actual failure of the system due to exhaustion failure.

Consequently, T_{exh} should not have an upper-bound in an asynchronous distributed fault-tolerant system operating in a environment prone to anything more severe than accidental faults. The goal should then be to maintain T_{exh} above T_{exec}, in all executions.

The findings in this chapter prompt us to two conclusions. Firstly, the theoretical impact of these findings remains across the fault spectrum. That is, such failure syndromes were previously unknown and even with accidental faults they can cause the inadvertent failure of asynchronous or synchronous distributed systems. These systems are to our findings fairly the same as "apparently working" pre-FLP asynchronous consensus systems were to FLP. In consequence, our results may alert researchers and help conceive better distributed systems.

Secondly, the practical impact of the same findings can in our opinion become higher, commensurate to the measure in which systems, critical or generic, are becoming prey to hacker attacks (malicious faults). This means that, with increasing probability, systems having the failure syndrome discovered in this thesis not only can but *will* be attacked and made to fail.

Finally, given that proactive recovery systems are a special case of systems aiming at achieving perpetual execution in face of the continuous production of faults, they deserve a special attention in this thesis.

Chapter 4

Proactive Resilience

One of the most interesting approaches to avoid resource exhaustion due to accidental or malicious corruption of components is through proactive recovery (Ostrovsky & Yung, 1991), which can be seen as a form of dynamic redundancy (Siewiorek & Swarz, 1992). The aim of this mechanism is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/faults. If the rejuvenation is 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 (Cachin *et al.*, 2002; Garay *et al.*, 2000; Herzberg *et al.*, 1995, 1997; Marsh & Schneider, 2004; Zhou *et al.*, 2002, 2005). It may also be utilized to restore the system code from a secure source to eliminate potential transformations carried out by an adversary (Castro & Liskov, 2002; Ostrovsky & Yung, 1991). Moreover, it may encompass the substitution of software components to remove vulnerabilities existent in previous versions (e.g., software bugs that could crash the system or errors exploitable by outside attackers). Vulnerability removal can also be done through address space randomization (Bhatkar

et al., 2003, 2005; Forrest *et al.*, 1997; PaX; Xu *et al.*, 2003), which could be used to periodically randomize the memory location of all code and data objects.

Thus, intuitively, by using a well-planned strategy of proactive recovery, T_{exh} can be recurrently increased in order that it is always greater than T_{exec} in all executions. However, this intuition is rather difficult to substantiate if the system is asynchronous. As it was explained in Section 2.2, the simple task of timely triggering a periodic recovery procedure is impossible to attain under the pure asynchronous model, namely if it is subject to malicious faults. From this reasoning, and according to Corollary 3.2.3, one can conclude that it is not possible to ensure the exhaustion-safety of an asynchronous system with bounded exhaustion time through asynchronous proactive recovery.

The impossibility of building an exhaustion-safe f fault/intrusion-tolerant distributed asynchronous system, namely in the presence of malicious faults, and even if enhanced with asynchronous proactive recovery, led us to investigate hybrid models for proactive recovery.

4.1 The Proactive Resilience Model

Proactive recovery 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 indicates that one of the possible approaches to use proactive recovery in a effective way, is to model and architect it under a hybrid distributed system model or *Wormholes model* (Veríssimo, 2006).

In this context, we propose the Proactive Resilience Model (PRM), a more resilient approach to proactive recovery. The PRM defines a system enhanced

with proactive recovery through a model composed of two parts: the proactive recovery subsystem and the payload system, the latter being proactively recovered by the former. Each of these two parts obeys different timing assumptions and different fault models, and should be designed accordingly to the guidelines of hybrid distributed system models (Veríssimo, 2006).

The payload system executes the "normal" applications and protocols. In this way, the payload synchrony and fault model entirely depend on the applications/protocols executing in this part of the system. For instance, the payload may operate in 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, albeit simple in functionality, is more demanding in terms of timing and fault assumptions, and it is modeled as an abstract distributed component called *Proactive Recovery Wormhole* (PRW). By abstract we mean that this component admits different instantiations. Typically, a specific 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 4.1. Each host contains a local module, called the *local PRW*. These modules are organized in clusters, called *PRW clusters*, and the local PRWs in each cluster are interconnected by a synchronous and secure *control network*. The set of all PRW clusters is what 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).

Conceptually, a local PRW is a module separated from the OS. In practice, this separation between the local PRW and the OS can be achieved in either

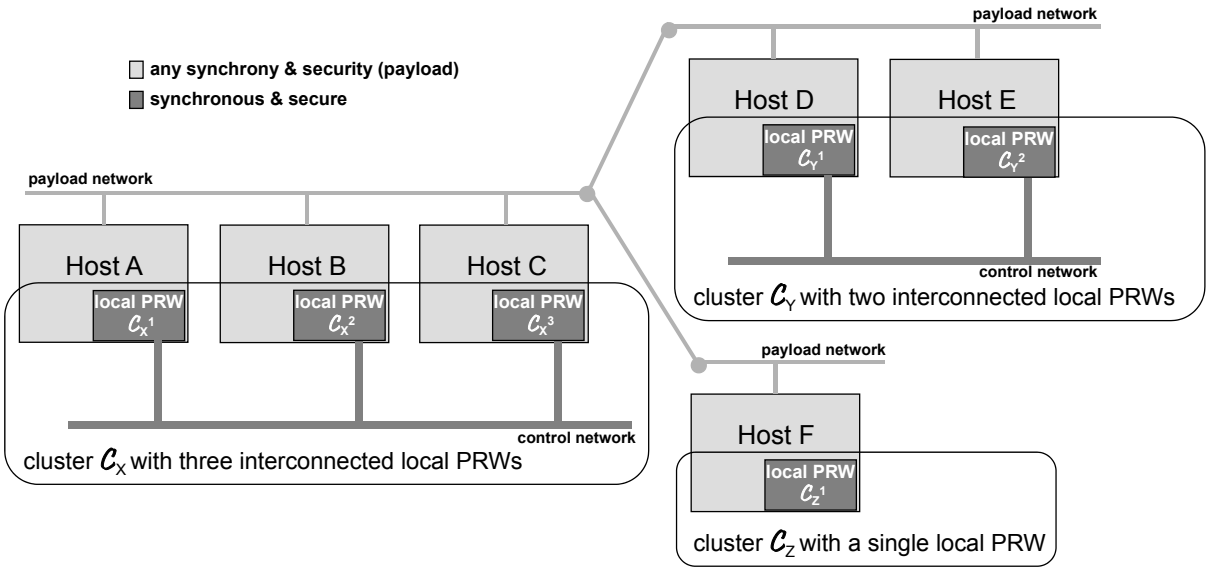

Figure 4.1: The architecture of a system with a PRW.

of two ways: (1) the local PRW is implemented in a separate tamper-proof hardware module (e.g., smartcard, or PC appliance board (Kent, 1980; Trusted Computing Group, 2004)) and so the separation is physical; (2) the local PRW is implemented on the native hardware, with a virtual separation and shielding (e.g., using software virtualization (Barham *et al.*, 2003)) between the local PRW and the OS processes.

The way clusters are organized is dependent on the rejuvenation requirements. Typically, a cluster is composed of nodes that are somehow interdependent w.r.t. rejuvenating (e.g., need to exchange information during recovery). We focus two specific cluster configurations:

PRW^l is composed of n clusters, each one including a single local PRW. Therefore, every PRW^l cluster is exactly like cluster $\mathcal{C}_Z$ depicted in Figure 4.1, and, consequently, no control network exists in any cluster;

PRW^d is composed of a single cluster including all local PRWs. For instance, if

the system is composed of 3 [resp. 2] nodes, then the (single) PRW^d cluster would be like cluster $\mathcal{C}_X$ [resp. $\mathcal{C}_Y$] depicted in Figure 4.1. In this case every local PRW is interconnected through the same control network.

PRW^l should be used in scenarios where the recovery procedure only requires local information, and therefore there is no need for distributed execution (e.g., rebooting a replica from clean media as described in Section 5.2). PRW^d should be used when the recovery is done through a fully distributed recovery procedure in which every local PRW should participate (e.g., proactive secret sharing as explained in Section 5.1). Many more configurations are possible, namely configurations composed of heterogeneous clusters (i.e., clusters with different sizes). We leave the discussion of such configurations and their usefulness as future work.

4.1.1 Periodic Timely Rejuvenation

The PRW executes periodic rejuvenations through a periodic timely execution service. This section defines the periodic timely execution service, proposes an algorithm to implement it, and specifies the real-time guarantees required of the PRW. Then, assuming that the local PRWs do not fail, Section 4.1.2 proves that systems enhanced with a PRW executing an appropriate periodic timely rejuvenation service are node-exhaustion-safe. Section 4.1.2 also discusses how this result can be generalized in order to take into account potential crashes of local PRWs.

Each PRW cluster runs its own instance of the periodic timely execution service, and there are no constraints in terms of the coordination of the different instances. Albeit running independently, each cluster offers the same set of properties dictated by four global parameters: F, T_P, T_D and T_π. Namely,

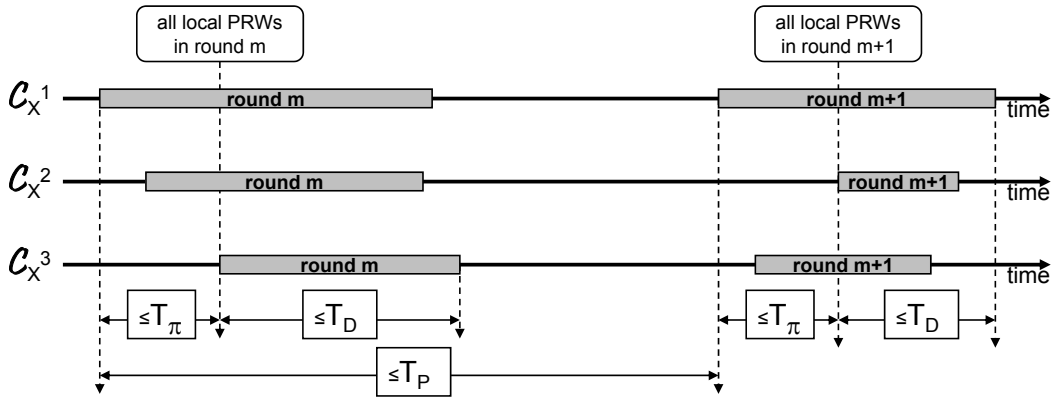

Figure 4.2: Relationship between T_P,T_D and T_π in a cluster $\mathcal{C}_X$ with three local PRWs.

each cluster executes a rejuvenation procedure F in rounds, and each round is triggered within T_P from the last triggering. This triggering is done by at least one local PRW (in each cluster), and all other local PRWs (of the same cluster) start executing the same round within T_π of each other. In this way, T_π corresponds to the maximum time interval between a local PRW entering a certain round, and all other local PRWs entering the same round. Moreover, each cluster guarantees that, once all local PRWs are in the same round, the execution time of F is bounded by T_D. Therefore, the worst case execution time of each round of F is given by $T_\pi + T_D$. Figure 4.2 illustrates the relationship between T_P, T_D, and T_π, in a cluster with three local PRWs. A formal definition of the periodic timely execution service is presented next.

Definition 4.1.1. *Let F be a procedure and $T_D, T_P, T_\pi \in \Re_0^+$, s.t. $T_D + T_\pi < T_P$. A set of components $\mathcal{C}$, organized in s disjoint and non-empty clusters $\mathcal{C}_1, ..., \mathcal{C}_s$, offers a* periodic timely execution service $\langle F, T_D, T_P, T_\pi \rangle$, *if and only if:*

1. *the components of the same cluster $\mathcal{C}_i$ execute F in rounds, and therefore F is a distributed procedure within a cluster;*

2. *for every real time instant t of $\mathcal{C}$ execution time, there exists a round of F trig-*

gered in each cluster $\mathcal{C}_i$ within T_P from t, i.e., one component C in each cluster $\mathcal{C}_i$ triggers the execution of a round of F within T_P from t;

3. *every component in a cluster $\mathcal{C}_i$ triggers the execution of the same round of F within T_π of each other component in the same cluster;*

4. *each cluster $\mathcal{C}_i$ ensures that, once all components are in the same round of F, the execution time of F is bounded by T_D, i.e., the difference between the real time instant when the last component in a cluster $\mathcal{C}_i$ starts executing F and the real time instant when the last component of the same cluster finishes executing is not greater than T_D (both executions refer to the same round).*

Corollary 4.1.2. *If $\mathcal{C}$ is a set of components, organized in s clusters $\mathcal{C}_1, ..., \mathcal{C}_s$, that offers a periodic timely execution service $\langle F, T_D, T_P, T_\pi \rangle$ then, for every real time instant t of $\mathcal{C}$ execution time, there exists a round of F triggered in each cluster $\mathcal{C}_i$ within T_P from t that is terminated within $T_P + T_D + T_\pi$ from t.*

Definition 4.1.3. *A system enhanced with a $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ has a local PRW in every host. Moreover these are organized in clusters and in conjunction offer the periodic timely execution service $\langle F, T_D, T_P, T_\pi \rangle$.*

As mentioned before, the PRW admits two particular cluster configurations — PRW^l and PRW^d. These are defined as follows.

Definition 4.1.4. *A system enhanced with a $PRW^l(\langle F, T_D, T_P, T_\pi \rangle)$ is a system enhanced with a $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ s.t. there exist n clusters $\mathcal{C}_1, ..., \mathcal{C}_n$, and each cluster $\mathcal{C}_i$ is composed of a single local PRW.*

Definition 4.1.5. *A system enhanced with a $PRW^d(\langle F, T_D, T_P, T_\pi \rangle)$ is a system enhanced with a $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ s.t. there exist a single cluster $\mathcal{C}_1$ comprising all local PRWs.*

A periodic timely execution service can be built using, for instance, Algorithm 1, on an environment that ensures the following properties:

P1 There exists a known upper bound on the processing delays of every local PRW.

P2 There exists a known upper bound on the clock drift rate of every local PRW.

P3 There exists a known upper bound on the message delivery delays of every control network interconnecting the local PRWs of a same cluster.

Suppose that each local PRW executes Algorithm 1, where function *clock* returns the current value of the clock of the local PRW, F is the recovery procedure that should be periodically timely executed, and T_P is the desired recovery periodicity. Value δ defines a safety time interval used to guarantee that consecutive recoveries are triggered within T_P from each other in the presence of the assumed upper bounds on the processing delays (P1) and the clock drift rate (P2). Notice that between the *wait* instruction in line 2 and the triggering of F in line 7, there is a set of instructions that take (bounded) time to execute. δ should guarantee that consecutive recoveries are always triggered within T_P of each other independently of the actual execution time of those instructions, and taking into account the maximum possible clock drift rate. However, δ should also guarantee that every local PRW triggers F within T_π of each other. So, δ should not be greater than $T_P - (T_D + T_\pi)$ in order to ensure that the local PRW ${\mathcal{C}_i}^1$ in each cluster $\mathcal{C}_i$ does not start to execute F too early (i.e., when other local PRWs may still be executing the previous round). In these conditions, the algorithm guarantees that F is always triggered, in each cluster $\mathcal{C}_i$, by local PRWs ${\mathcal{C}_i}^1$ within T_P from the last triggering. Moreover, given that it is assured that different rounds do not overlap, the triggering instant in the

Algorithm 1: Periodic timely execution service run by each local PRW $\mathcal{C}_i{}^j$ in cluster $\mathcal{C}_i$

```
   initialization: t_last ← clock
   begin
       while true do
           /* local PRWs C_i^j with j = 1 in each cluster C_i coordinate the
              recovering process                                          */
1          if j = 1 then
2              wait until clock = t_last + T_P − δ
3              t_last ← clock
4              multicast(trigger, C_i)
5          else
6              receive(trigger)
7          execute F
   end
```

local PRWs of the same cluster differs in at most the maximum message delivery delay (P3) plus the maximum processing delay, i.e., the time necessary for message *trigger* to be delivered and processed in all local PRWs. Thus, the value of T_π is defined by this sum. In this situation, each local PRW offers a periodic timely execution service $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ provided it ensures that, once all local PRWs are in the same round of F, its execution time is bounded by T_D.

4.1.2 Building Node-Exhaustion-Safe Systems

A distributed system enhanced with a $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ can be made node-exhaustion-safe under certain conditions, as it will be shown in Theorem 4.1.6. This theorem states that if it is possible to lower-bound the exhaustion time (i.e., the time needed to produce $f + 1$ node failures) of every system execution by a known constant $T_{exh_{min}}$, then node-exhaustion-safety is achieved by assuring that $T_P + T_D + T_\pi < T_{exh_{min}}$.

In what follows, let $[\![S]\!]$ denote the set of executions of an f fault-tolerant distributed system S under the REX model for the condition $\varphi_{node} = n_{fail} \leq f$, where n_{fail} represents the number of nodes which, during an execution, are failed simultaneously. Notice that the type of failure is not specified, but only that nodes may fail in some way and that this failure can be recovered through the execution of a rejuvenation procedure. A node failure may be for instance the disclosure of some secret information (the type of failures considered in Section 5.1), or a hacker intrusion that compromises the behavior of some parts of the system. Notice also that the rejuvenation procedure will depend on the specific type of failure one wishes to recover from. For instance, whereas a direct intrusion may require the reboot of the system and the reloading of code and state from some trusted source, the disclosure of secret information may be solved by simply turning that information obsolete.

Theorem 4.1.6. *Suppose that:*

1. *S is a system composed of a total of n nodes which, once failed, do not recover, and let* $T_{exh_{min}} = \inf(\{T^{\mathcal{E}}_{exh} : \mathcal{E} \in [\![S]\!]\})$[1]*;*

2. *The time needed to produce* $f + 1$ $(\leq n)$ *node failures at any instant is independent of the number of nodes that are currently failed;*

3. *F is a distributed procedure that upon termination ensures that all nodes involved in its execution are not failed.*

Then, system S enhanced with a $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ *s.t.* $T_P + T_D + T_\pi < T_{exh_{min}}$ *is node-exhaustion-safe w.r.t.* φ_{node}*.*

Proof: Assumption (1) entails that, in every execution of S, from a state with 0 failed nodes, it takes at least $T_{exh_{min}}$ for $f + 1$ node failures to be produced. Let

[1] $\inf()$ denotes the infimum of a set of real numbers, i.e., the greatest lower bound for the set.

m be a natural number such that $m + f + 1 \leq n$. Then, using assumption (2), we may conclude that, in every execution of S, it takes at least $T_{exh_{min}}$ to reach a state with $m + f + 1$ failed nodes from a state with m failed nodes[1]. This also means that :

4. in every execution of S, the number of node failures during a time interval $]t, t + T_{exh_{min}}[$ is at most f.

By contradiction, assume that there exists an execution of the system S, enhanced with a $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ s.t. $T_P + T_D + T_\pi < T_{exh_{min}}$, which violates φ_{node}. This means that there is a time instant t_C when there are more than f failed nodes. Notice that t_C cannot occur in less than $T_{exh_{min}}$ from the system initial start instant, because this would mean that more than $f + 1$ node failures were produced in less than $T_{exh_{min}}$ from a state with *0* failed nodes, which is contradictory with assumption (1). Hence, t_C occurs in at least $T_{exh_{min}}$ from the system initial start instant.

Then, by (4), in $t_I = t_C - T_{exh_{min}}$ there is at least one failed node, because in less than $T_{exh_{min}}$ is not possible that more than f nodes become failed. Given that we assumed that the PRW never fails and given that the nature of F is to recover the nodes of the cluster where F is executed (assumption (3)), the execution of S under $PRW(\langle F, T_D, T_P, T_\pi \rangle)$ with $T_P + T_D + T_\pi < T_{exh_{min}}$ ensures that any node that is failed at t_I is recovered no later than $t_I + T_P + T_D + T_\pi$ and, hence, is recovered earlier than $t_C = t_I + T_{exh_{min}}$. If one of the nodes that are failed in t_I becomes recovered before t_C and there are more than f failed nodes in $t_C = t_I + T_{exh_{min}}$, then more than f nodes become failed in the interval $]t_I, t_I + T_{exh_{min}}[$. But this is contradictory with (4) above. ■

[1]Notice that a node may fail, be recovered, fail again, and so on. Therefore, the total number of node failures does not correspond necessarily to the number of currently failed nodes.

From Theorem 4.1.6 it follows that, in order to build a node-exhaustion-safe intrusion-tolerant system, the system architect should choose an appropriate degree of fault-tolerance f, such that $T_P + T_D + T_\pi < T^{\mathcal{E}}_{exh}$, for every system execution $\mathcal{E}$. In other words, any interval with length $T_P + T_D + T_\pi$ should not be sufficient for $f + 1$ node failures to be produced, throughout the lifetime of the system.

As mentioned before, the results presented in this section depend on the assumption that local PRWs never fail. This assumption allows to abstract from PRWs crashes and, in this way, allows to focus on what is really important. However, Theorem 4.1.6 could be extended to the case where the number of crashes is upper-bounded by some known constant f_c. The difference would be that one would need to add sufficient redundancy to the system in order to resist the f_c possible crashes, and the protocol(s) executed by the PRW would also have to take this into account. Section 5.1.2 explains how this could be done in a concrete scenario. In order to minimize the probability of crashing more than f_c local PRWs, and in this way guarantee the exhaustion-safety of the overall system, the system architect would need to estimate the probability of crash according to environment conditions and/or apply techniques of dynamic redundancy, where crashed PRWs could be repaired or replaced before more than f_c become crashed.

4.2 Evaluation

This section presents the results of evaluating, through simulation, whether or not our intrusion-tolerance approach, *proactive resilience*, is sufficient to achieve node-exhaustion-safety and availability for an assumed fault rate. It also shows how previous approaches fail to guarantee such behavior. It starts by present-

ing the modeling formalism and describing both the models developed to represent an abstract distributed system that periodically recovers, and the adversary that tries to break the system. Then, it presents the results of simulating such an environment when using and not using proactive resilience.

The simulation model incorporates a technique that allows the system to live under two different timebases: one representing the pace of generation of faults and/or attacks, and another representing the internal execution, e.g., of recoveries. The former largely depends on physical events happening in real time (such as accidental fault generation and hacker attacks) and thus should be modeled as having a synchronous behavior, whereas the latter depends on the internal system synchrony assumptions. The innovation in this separation is more important than meets the eye, for at the root of our initial findings was the discovery that the asynchronous system models used so far did not depict accurately enough the subtle timing relations between the pace at which faults occur and the pace at which a system executes, leading to unexpected failures by exhaustion of system resources. For instance, as described in Section 2.2.1, a simple attack on a node's clock drift rate may slow down the rate of recoveries, and thus increase the probability that another type of faults (potentially more dangerous) will be successful. As already mentioned before, such a (timing) attack is undetectable under asynchronous assumptions.

4.2.1 Node-Exhaustion-Safety and Availability

In order to perform an evaluation through simulation, we start by defining a more specialized model for evaluating exhaustion-safety, which can help to assess if a fault-tolerant distributed system using proactive recovery is node-exhaustion-safe.

As pointed out in Section 2.2.5, recoveries may introduce availability prob-

lems given that the recovering nodes may be unavailable during some time. Therefore, the system architect should take this into account if availability is a requirement. The model presented here specifies what are the exact conditions to guarantee node-exhaustion-safety and availability. Then, Section 5.2.4 discusses this same issue in the context of state machine replication.

We model a distributed system adversary through a parameter:

- $mift$, the minimum inter-failure time, which measures the speed at which the adversary is capable of attacking and causing individual node failures.

We want to keep the model simple enough, so for the purpose of this thesis we assume that whenever there are dependencies between node failures creating common failure modes (e.g., two nodes using the same operating system and thus being vulnerable to the same type of attacks on the OS vulnerabilities), this can be absorbed by a smaller $mift$. Despite this simplification, the model is sufficiently representative to discuss the problems presented in Section 2.2.5.

We model the distributed system itself through three parameters:

- met, the maximum execution time, represents the maximum duration of a meaningful execution (e.g., protocol run, transaction, server action, or sequences thereof);

- $mirt$, the maximum inter-recovery time, specifies the maximum interval between the triggering of a recovery procedure and the termination of the next consecutive one (i.e., the maximum interval between the termination of two consecutive recoveries in any node, given that nodes may recover by a different order in different recoveries);

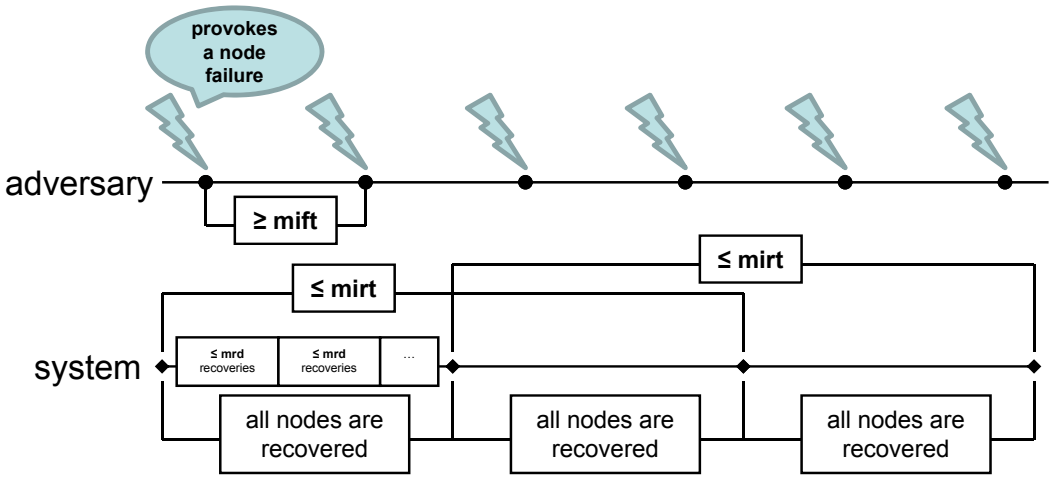

Figure 4.3: Relationship between $mift$,$mirt$ and mrd.

- mrd, the maximum recovery degree, specifies the maximum number of nodes that are recovered simultaneously at any recovery step (i.e., a recovery procedure is composed of at least $\lceil \frac{n}{mrd} \rceil$ recovery steps, where n is the total number of nodes).

The last two parameters typify system recoveries. A recovery execution can take several steps and recovers all the nodes of the system; hence, regardless of whether they failed before recovery, all the nodes become correct after a recovery. Figure 4.3 illustrates the relationship between $mift$, $mirt$ and mrd.

Notice that during a recovery step, the nodes in question may be unavailable. Therefore, the system must have extra redundancy if it is to continue operating uninterrupted through recoveries. The discussion of how the state of the nodes is affected by a recovery is out of the scope of this work. In fact, this is an advantage: our model is sufficiently generic that it can be applied to either stateful or stateless recovery.

If the system maximum execution time met is known, then recoveries are not necessary if the system avoids exhaustion during execution. It is trivial

to see that the following condition should be satisfied: the system should be resourced so as to tolerate $f \geq \lceil \frac{met}{mift} \rceil$ faults. However, if the system has an unbounded execution time, or if there is a bound on the amount of redundancy available to deploy the system, recoveries are mandatory.

Let us start by defining node-exhaustion-safety and availability under the described model.

Definition 4.2.1. *Let S=$(n, f_a, f_c, met, mift, mirt, mrd)$ be a distributed system with n nodes, able to resist a maximum number $f_a \leq n$ of arbitrary failures, and a maximum number $f_c \leq n$ of crash failures, and let, for all nodes, met represent maximum execution time; mift represent minimum inter-failure time; mirt represent maximum inter-recovery time; and mrd represent maximum recovery degree.*

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

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

Proposition 4.2.2. *Let S be a fault-tolerant distributed system under the asynchronous model, and able to resist a known maximum number f_a of arbitrary node failures. Then, S is not node-exhaustion-safe.*

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

This result is not surprising given that it was showed in Corollary 3.2.3 that any asynchronous system with a finite exhaustion time is not exhaustion-safe.

Notice that Proposition 4.2.2 applies to any type of fault-tolerant asynchronous distributed system, including ones using asynchronous proactive recovery (Cachin *et al.*, 2002; Castro & Liskov, 2002; Marsh & Schneider, 2004; Zhou *et al.*, 2002, 2005). The ultimate goal of asynchronous proactive recovery is to guarantee that the value of $mirt$ is such that more than f_a node failures never occur. However, as shown above, this is theoretically impossible by definition of asynchrony. In practical terms, it is also readily observable, since the asynchrony pattern leading to exhaustion can be induced by a malicious adversary, as it was discussed in Section 2.2.

This problem can be better understood if we consider the following. Variables met, $mift$, and $mirt$ measure real-time intervals from an omniscient observer's perspective. $mift$ measures the activity leading to the production of node failures due to faults or intrusions. $mift$ largely depends on physical events that happen in real time (e.g., accidental fault generation or external hacker attacks), and in the worst case it is independent of the system's speed of execution. Also, $mift$ needs to be lower-bounded, say by $Mift$: this is an assumption of the type made in several similar systems (Cachin *et al.*, 2002; Castro & Liskov, 2002; Zhou *et al.*, 2002). If the minimum inter-failure time ($mift$) were not lower-bounded with $Mift$, the adversary would have infinite power (e.g., $mift$=0), and it would be impossible to derive an exhaustion-safe design. Notice that this timing assumption is about fault production and thus is represented in the external timebase, not compromising at all the asynchrony of

the system itself, which runs according to the internal timebase. *met* measures the longest duration of an execution, which is non-definable if the system is asynchronous or if the system executes forever.

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

Consider now an asynchronous fault-tolerant distributed system with a $mift$=20 time units, expected to rejuvenate periodically in intervals equal to or shorter than *mirt*=30 time units. Assume also that *mrd*=1. From Definition 4.2.1, this system is node-exhaustion-safe iff it is able to resist $f_a \geq 2$. Assume that one deploys such a system with f_a=2, programmed internally to rejuvenate in order that *Mirt*=30. The system should be node-exhaustion-safe in theory, but this is not necessarily true. Suppose the system's execution is slowed down by an internal time factor itf=3: that is, all system actions run three times slower than expected. This is normal behavior for an asynchronous system, by definition. Then, whatever triggers and executes rejuvenation

is also affected by this delay: $mirt{=}Mirt \times itf = 30 \times 3 = 90$ time units. So in reality, the interval between the start and termination of two consecutive rejuvenation periods is 90 time units, instead of 30 time units. However, this interval is long enough for more than two arbitrary faults to occur, inducing a potential system failure.

A distributed intrusion-tolerant system built according to the proactive resilience model is node-exhaustion-safe and ensures availability in the following conditions.

Theorem 4.2.3. *Let $S = (n, f_a, f_c, met, mift, mirt, mrd)$ be a distributed system able to resist at most f_a arbitrary node failures, and at most f_c crash node failures. If S is periodically recovered through a PRW instantiation with parameters T_P, T_D, and T_π, then S is node-exhaustion-safe iff $f_a \geq \lceil \frac{\min(met, T_P+T_D+T_\pi)}{mift} \rceil$. Moreover, S ensures availability iff $f_c \geq mrd$.*

Proof: By Definition 4.2.1, S is node-exhaustion-safe iff $f_a \geq \lceil \frac{\min(met, mirt)}{mift} \rceil$ and it ensures availability iff $f_c \geq mrd$. Thus, it suffices to show that $T_P + T_D + T_\pi = mirt$. From Corollary 4.1.2 it follows that the interval between the triggering of a recovery and the termination of the next consecutive one is upper-bounded by $T_P + T_D + T_\pi$, and therefore $T_P + T_D + T_\pi = mirt$. ■

4.2.2 SAN Models

We use Stochastic Activity Networks (SANs) (Sanders & Meyer, 2000) as the modeling formalism. SANs are a convenient, graphical, high-level language for describing system behavior. SANs are useful in capturing the stochastic (or random) behavior of a system.

Stochastic Petri nets are a subset of SANs. A stochastic Petri net has the following components: *places* (denoted by circles), which contain tokens and

are like variables; *tokens*, which indicate the "value" or "state" of a place; *transitions* (denoted by ovals), which change the number of tokens in places; *input arcs*, which connect places to transitions; and *output arcs*, which connect transitions to places. A transition is enabled if for each place connected by input arcs, the number of tokens in the place are greater than or equal to the number of input arcs connecting the place and the transition. When a transition is enabled, it may fire, removing a token from the corresponding place for each input arc and adding a token to the corresponding place for each output arc. An exponentially distributed time is assigned to each transition. The term *marking* of a place is used to indicate the number of tokens in the place.

Stochastic Petri nets, while being easier to read, write, modify, and debug than Markov chains, are still limited in their expressive power, since they may perform only $+$, $-$, $>$, and test-for-zero operations. That makes it very difficult to model complex interactions, and more general and flexible formalisms are needed to represent real systems. Stochastic activity networks are one such extension. They have many properties, which include a general way to specify that an activity (transition) is enabled, a general way to specify a completion (firing) rule, a way to represent zero-timed events, a way to represent probabilistic choices upon completion, state-dependent parameter values, and general delay distribution on activities.

SANs have all the components of Stochastic Petri nets plus four more: *input gates* (denoted by triangles pointing left), which are used to define complex enabling predicates and completion functions; *output gates* (denoted by triangles pointing to the right), which are used to define complex completion functions; *cases* (denoted by small circles on activities), which are used to specify probabilistic choices; and *instantaneous activities* (denoted by vertical lines), which are used to specify zero-timed events.

An input gate has two components: an enabling predicate and an input function. An activity is enabled if for every connected input gate, the enabling predicate is true, and for each input arc, the number of tokens in the connected place is greater than or equal to the number of arcs. Each case has a probability associated with it and represents a probabilistic choice of the action to take when an activity completes. When an activity completes, an output gate allows for a more general change in the state of the system than an output arc does. The output gate function is usually expressed using pseudo-C code. The input functions of all input gates connected to an activity are also executed when the activity completes. Those functions are also expressed in pseudo-C code. The times between enabling and firing of activities can be distributed according to a variety of probability distributions, and the parameters of the distribution can be a function of the state.

Composed models consist of SANs that have been replicated and joined multiple times. Replicated and joined models can interact with each other through a set of places (called common places) that are common to multiple submodels. A comprehensive description of SANs can be found in Sanders & Meyer (2000).

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

In the remainder of the section, Figures 4.4, 4.5, 4.6, 4.7, and 4.8 present the SAN models. It is not necessary to understand them in order to follow the explanations in the text. To understand fully the graphical representation of the models and the models themselves, the interested reader can find detailed explanations of the generic SAN's formalism in Sanders & Meyer (2000), and a detailed documentation of the models is presented in Appendix A.

4.2.2.1 SAN Model for the External/Internal Timebases

The external/internal timebases SAN in Figure 4.4 models the rate at which the external and internal timebases progress. The external timebase advances at the same rate as the simulator clock (represented by the activity `clock_tick`), while the internal timebase advances at a rate specified by a node parameter, the `internal time rate`. In fact, we model two different internal timebases: the payload one used by normal applications, and the proactive recovery one used by the recovery mechanism. Although these two internal timebases are coincident in a typical homogenous system, we need to separate them in order to model our hybrid architecture. Notice that both internal time (variables `payload_internal_time_vector` and `pr_internal_time_vector`) and internal time rate (variables `payload_internal_time_rate_vector` and `pr_internal_time_rate_vector`) are specified as vectors. This happens because different nodes may have different internal time behaviors.

This SAN model is the basis of our novel approach to intrusion-tolerant modeling and evaluation. The different timebases will be used by the subsequent submodels according to the type of dependencies they have or do not have on the system timing assumptions.

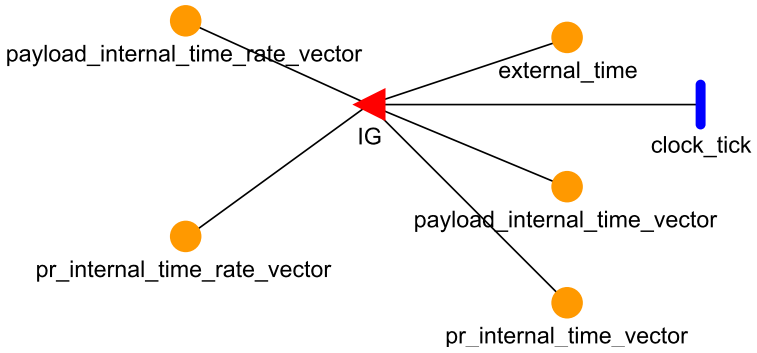

Figure 4.4: SAN model for the external/internal timebases.

4.2.2.2 SAN Model for the Stealth Time Adversary

The stealth time adversary SAN in Figure 4.5 models a special kind of adversary. This adversary does not behave like the classic one (described in Section 4.2.2.4) that simply tries to compromise system nodes, but instead has a more specific goal: to detach a node's internal timebase from the external one. Such an adversary makes it possible to model both random and intentional (i.e., maliciously triggered) asynchrony. The stealth time adversary behavior is specified through two parameters:

- `stealth time attack period` specifies the minimum *external time* interval between stealth time attacks. In each attack (represented by the activity `stealth_time_intrusion`), the adversary randomly chooses and compromises (a previously correct) victim.

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

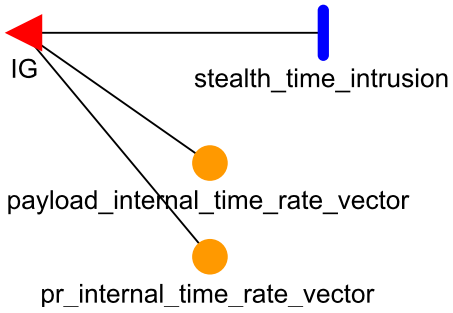

Figure 4.5: SAN model for the stealth time adversary.

4.2.2.3 SAN Model for the Conspicuous Time Adversary

The conspicuous time adversary SAN in Figure 4.6 models a different type of time adversary that simply delays system actions. This adversary may be seen as a particular case of the generic classic adversary (described in Section 4.2.2.4) that develops, for instance, a DoS attack. However, we have decided to model it separately in order to discuss in Section 4.2.3 the theoretical difference between a conspicuous and a stealth time adversary, and how malicious behavior may be detected in one case but not in the other. The conspicuous time adversary behavior is specified through two parameters:

- `conspicuous time attack period` specifies the minimum *external time* interval between conspicuous time attacks. In each attack (represented by the activity `conspicuous_time_intrusion`), the adversary randomly chooses and compromises (a previously correct) victim.

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

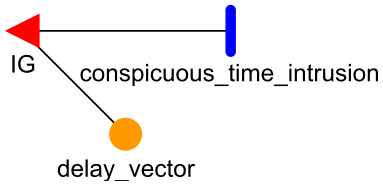

Figure 4.6: SAN model for the conspicuous time adversary.

4.2.2.4 SAN Model for the Classic Adversary

The classic adversary SAN in Figure 4.7 models the typical adversary that is constantly trying to corrupt system nodes, and that ultimately exhausts the distributed system when more than the assumed number of nodes are compromised. The classic adversary behavior is specified through one parameter:

- `minimum inter-failure time` specifies the minimum *external time* interval between attacks. In each attack (represented by the activity `intrusion`), the adversary randomly compromises one node.

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

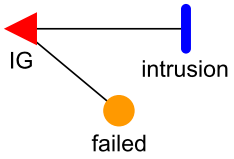

Figure 4.7: SAN model for the classic adversary.

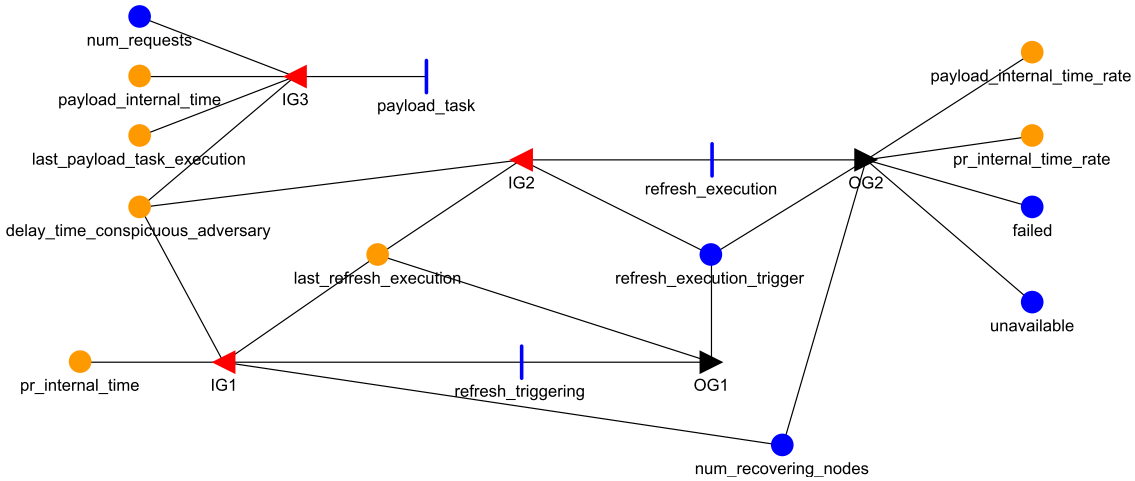

Figure 4.8: SAN model for a node.

4.2.2.5 SAN Model for a Node

The last submodel is the node SAN presented in Figure 4.8, which models the periodic recoveries done at each node. This submodel is more complex than the other ones, mainly because activities, such as `refresh_triggering` and `refresh_execution`, are scheduled according to the *internal timebase*. After a node recovery, the node becomes correct, its internal timebase is re-synchronized with the external one, and any delay injected by the conspicuous adversary is removed. However, during node recovery, the node is unavailable. The node behavior is specified through three parameters:

- T_P specifies the maximum *internal time* interval between two consecutive recoveries.
- T_D specifies the maximum *internal time* interval between the start and termination of a recovery.
- `maximum recovery degree` specifies the maximum number of nodes recovered simultaneously.

In order to simplify the model it is assumed that recovering nodes start the recovery exactly at the same time instant, and thus $T_\pi = 0$.

The most relevant variables of the node SAN are described next:

- `failed` stores a boolean value indicating if the node is failed. This variable can be updated by the classic adversary as a result of an attack, or by the `refresh_execution` activity as a result of a recovery.
- `unavailable` stores a boolean value indicating if the node is recovering.
- `num_recovering_nodes` stores the number of nodes that are recovering at each instant. It serves only statistical purposes.

4.2.2.6 Composed Model

The composed model for the simulation environment is presented in Figure 4.9. It consists of the five atomic SAN submodels presented in the previous sections, organized in the following way: one node SAN per system node (a maximum of 7 nodes is used), one classic adversary SAN, one stealth time adversary SAN, one conspicuous time adversary SAN, and one external/internal timebases SAN. The composed model also includes a monitor submodel, which serves only statistical purposes, by collecting statistics about the progress of other SANs.

The overall model behavior is specified by the parameters defined for the five submodels, plus the following: n specifies the total number of system nodes, f represents the assumed maximum number of node failures, and *met* specifies the maximum execution time of a simulation. When more than f failures happen at the same time, the system is considered exhausted.

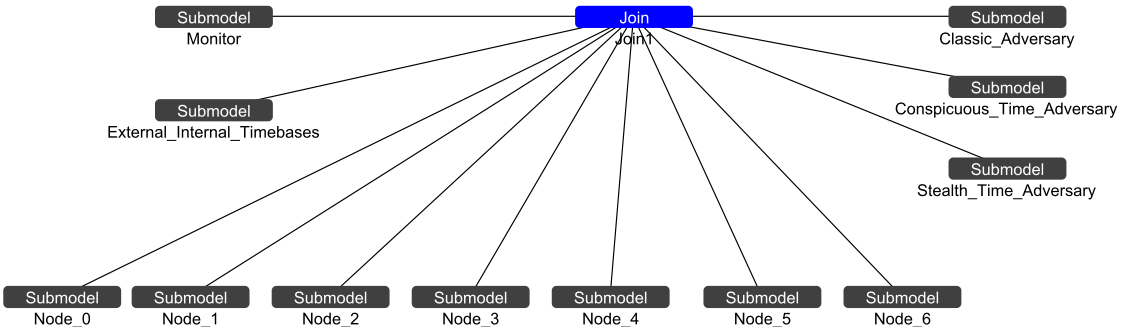

Figure 4.9: SAN model for the composed model.

4.2.3 Simulation Results

We used the Möbius (Deavours *et al.*, 2002) tool to build the SANs, define the availability and the intrusion tolerance measures, design studies on the model, simulate the model, and obtain values for the measures defined on various studies. The goal of the simulations was to put in evidence the four problems of proactively recovered systems that were introduced in Section 2.2.5:

1. A malicious adversary may deploy more power than originally assumed and corrupt nodes at a pace faster than recovery;

2. she or he may attempt to slow down the pace of recovery in order to leverage the chances of intruding the system with the available power;

3. she or he may perform stealth attacks on the system timing, which in asynchronous or partially synchronous systems may not even be perceived by the essentially time-free logic of the system, leaving it defenseless;

4. recovery procedures often involve bringing individual nodes to a temporarily inactive state, lowering the redundancy quorum and thus system resilience.

In the next subsections, it is shown that previous approaches to proactive recovery can be affected by all these problems, and that proactive resilience can be used to address all of them with the exception of problem 1, which is a fundamental one.

We used two metrics in our simulations: *percentage of exhausted time* and *percentage of unavailability time*. The former shows the amount of time the system has more than f nodes compromised and is thus very vulnerable to failures, especially those maliciously provoked (e.g., in a $3f + 1$ Byzantine-resilient system for f=1, it shows the percentage of time the system runs with at most 2 nodes). The latter shows the amount of time the system is unavailable due to recoveries, i.e., when the system cannot make progress due to an insufficient number of correct replicas, because some of them are recovering. Unless specified otherwise, the simulations were done with parameters n=4, f=1, mrd=1, met=10000, T_P=35, and T_D=4, with times in abstract units.

Notice that, according to Theorem 4.2.3, and in these conditions, the sheer limit of the resilience of the system to attacks lies around $mift$=$\frac{T_P+T_D}{f}$=39, after which the attack is so powerful that the system starts to give in. This, in fact, is the previously mentioned fundamental limitation (problem 1) for any design.

4.2.3.1 Impact of Time Adversaries on Exhaustion

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

Figures 4.10 and 4.11 illustrate how exhaustion time changes as a function of the combined strength of the classic and the conspicuous time adversaries, when using, respectively, asynchronous recovery and PRW recovery. With asynchronous recovery, the system starts to exhaust with much higher val-

ues of $mift$ than the baseline resilience (i.e., with $mift > 39$) in the presence of time attacks of decreasing period (Figure 4.10), whereas the PRW renders the system immune to those timing attacks (Figure 4.11).

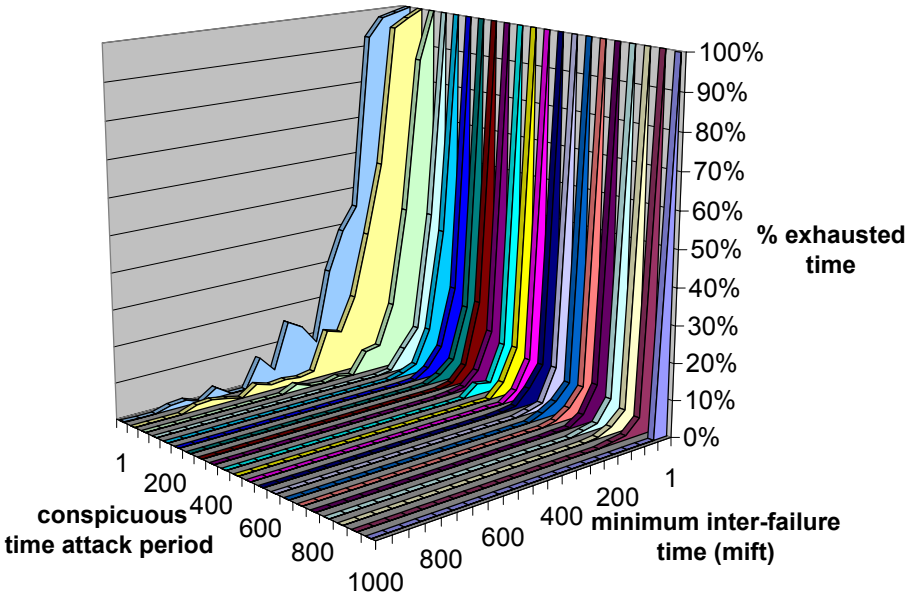

Figure 4.10: Exhausted time per conspicuous time attack period and minimum inter-failure time (Asynchronous recovery).

Figures 4.12 and 4.13 illustrate, respectively, for asynchronous recovery and PRW recovery, the impact of a stealth time adversary that, every 100 time units, slows the internal timebase of a different node down. The graphs depict increasing amounts of speed-down (time attack factor). As in the previous scenario, the system exhausts much faster when the time attack factor increases (Figure 4.12), whereas the PRW also renders the system immune to this second type of attacks (Figure 4.13). However, note that these attacks are more efficient, since with less power (stealth attacks on, e.g., timers, or interrupt routines, vs. conspicuous direct attacks, e.g., of the denial-of-service type), they achieve a more dire effect, as shown in Figure 4.12: for attack factors of 200 and up, the system becomes almost permanently exhausted; for an attack fac-

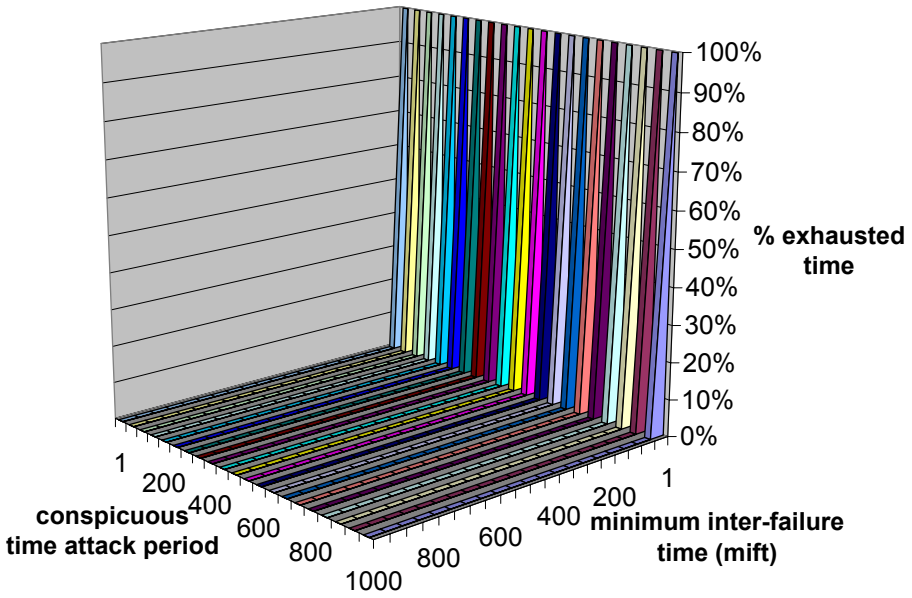

Figure 4.11: Exhausted time per conspicuous time attack period and minimum inter-failure time (PRW recovery).

tor of 1000, the system is exhausted 80% of the time.

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

The attack on the internal timebase is completely different in the sense that the adversary is attacking the time references of the system and thus programmatic detections are not reliable, because they use these same time references. Moreover, the attack will not necessarily affect the entire system. For instance,

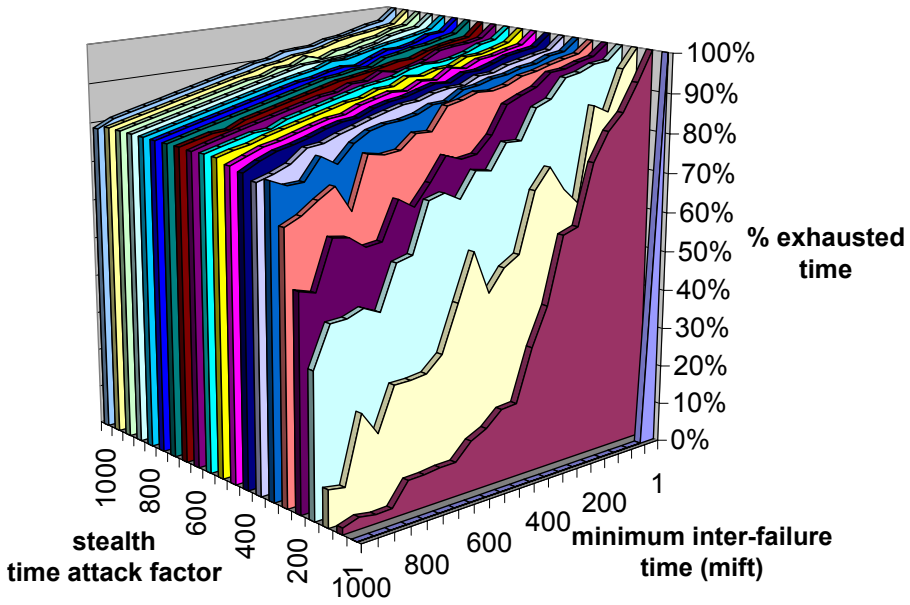

Figure 4.12: Exhausted time per stealth time attack factor and minimum inter-failure time (Asynchronous recovery).

the adversary may tamper with the kernel function that returns the current value of the local clock, and make it return different values to different applications. Or, alternatively, the attack may be applied to kernel scheduler/dispatcher code, only lengthening the execution of the functions used by some processes. Therefore, a stealth time adversary may be very difficult to defend against in classical asynchronous or even partially synchronous systems. The neutralization of this kind of attacks is one of the main results of this thesis.

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

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

Let us focus on a typical Byzantine fault-tolerant scenario in which n=4,

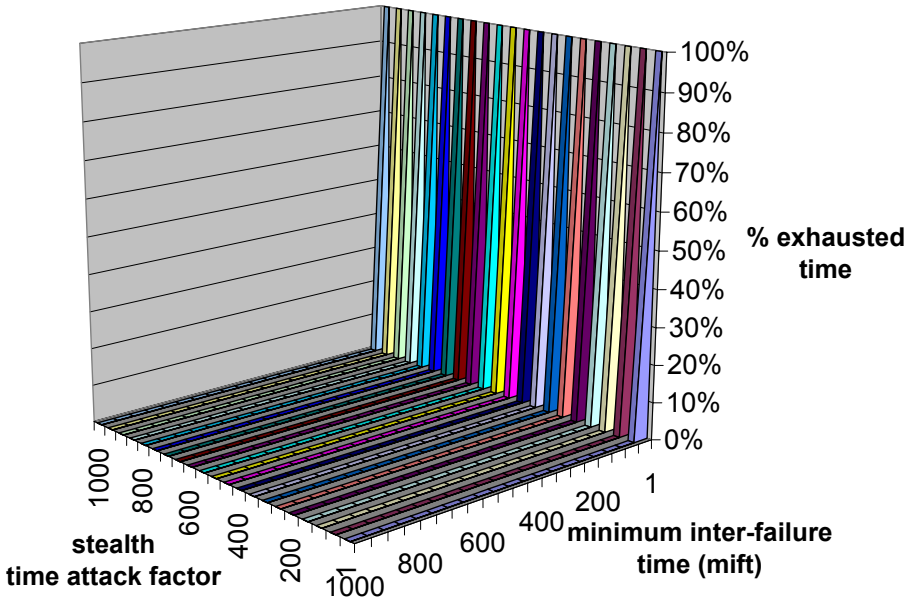

Figure 4.13: Exhausted time per stealth time attack factor and minimum inter-failure time (PRW recovery).

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

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

unavailability would make the system systematically exhausted and thus in danger of being compromised.

Figure 4.14 compares exhaustion time and unavailability in each of the scenarios. For this simulation, we set T_P=100 and T_D=10. We see that with $mift$=1000, system resources are never exhausted if the system stops during recoveries: otherwise, exhaustion will occur 0.66% of the time. Then, if the system stops during recoveries, it remains 0% exhausted with $mift$=100, but it is in turn unavailable for 7.17% of the time (precisely the amount of time the system is exhausted if it does not stop during recoveries). Thus, if the system does not stop during recoveries, it is naturally never unavailable due to recoveries, but it exhausts faster and thus has a greater probability of failing.

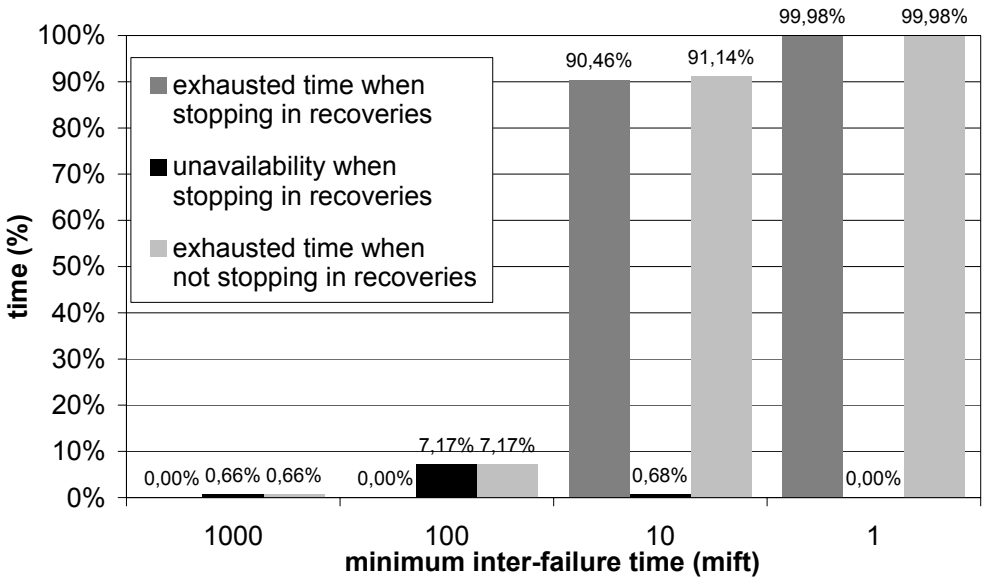

Figure 4.14: Trade-off between intrusion-tolerance and availability with T_P=100, T_D=10, mrd=1. ($n = 4, f = 1$, with PRW).

In order to ensure both intrusion-tolerance and availability, the system needs a sufficient redundancy quorum to avoid exhaustion between and during recoveries. From Theorem 4.2.3, the system should be able to resist at least

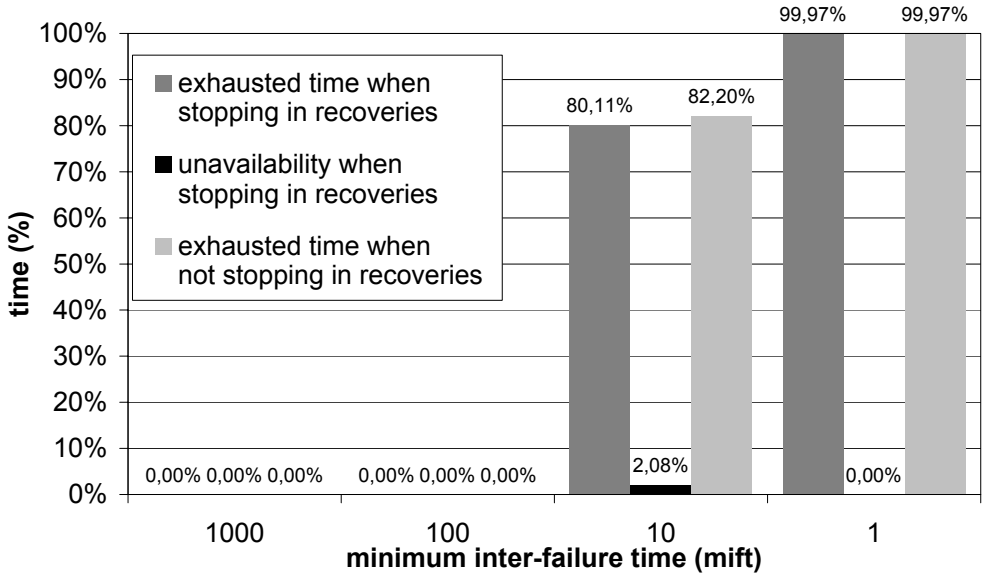

Figure 4.15: Trade-off between intrusion-tolerance and availability with T_P=100, T_D=10, mrd=1. ($n = 7, f = 2$, with PRW).

$f_a \geq \lceil \frac{\min(met, T_P + T_D)}{mift} \rceil$ arbitrary node failures, and $f_c \geq mrd$ crash node failures. Using the values of the scenario above, and if we assume at design time that $mift \geq 110$, we see that $f = f_a + f_c \geq 1 + 1 = 2$. In order to confirm these calculations, we simulated such a configuration and obtained the progress of exhaustion time and unavailability in comparison with the decrease of $mift$. Figure 4.15 illustrates the behavior of a distributed system with f=2 and n=7[1]. We see that, independent of the strategy followed, no exhaustion or unavailability occurs if the adversary behaves as assumed (i.e., if $mift \geq\sim 110$). Otherwise, we are in problem 1, which is unsolvable. Notice that we are not taking into consideration the effects of intentional or random asynchrony; therefore, T_P and T_D are guaranteed in both Figures 4.14 and 4.15 (the PRW was used in both experiments). The important message here is that, at design time, the system architect should calculate a sufficient redundancy quorum to resist in-

[1]Given that f=2, $3f + 1 = 7$ nodes are required in a Byzantine fault-tolerant scenario.

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

Chapter 5

Application Scenarios

This chapter describes two examples of application scenarios where proactive resilience can be applied. Section 5.1 describes the design of a distributed f fault/intrusion-tolerant *secret sharing system*, which makes use of a specific instantiation of the PRW – the Proactive Secret Sharing Wormhole – targeting the secret sharing scenario. Section 5.2 describes a resilient f fault/intrusion-tolerant *state machine replication architecture*, which guarantees that no more than f faults ever occur while ensuring availability. The architecture makes use of another instantiation of the PRW – the State Machine Proactive Recovery Wormhole – to periodically remove the effects of faults from the replicas.

5.1 Resilient Secret Sharing

5.1.1 Proactive Secret Sharing

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 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. This work uses Shamir's approach (Shamir, 1979) to implement a $(k+1, n)$-threshold scheme. This scheme can be defined as follows: given an integer-valued secret s, pick a prime q that 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. Moreover, given k or fewer shares, it is impossible to reconstruct s.

A special case where $k = 1$ (that is, two shares are required for reconstructing the secret), is given in Figure 5.1. The polynomial is a line and the secret is the point where the line intersects with the y-axis (i.e., $(0, f(0)) = (0, s)$). Each share is a point on the line. Any two (i.e., $k+1$) points determine the line and hence the secret. With just a single point, the line can be any line that passes the point, and hence it is insufficient to determine the right y-axis cross point.

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

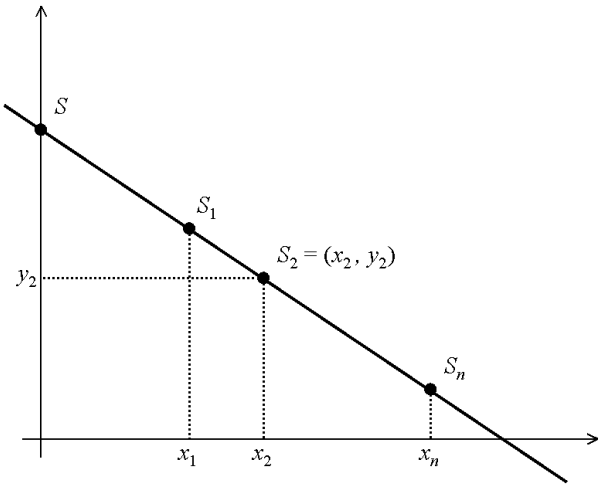

Figure 5.1: Shamir's secret sharing scheme for $k = 1$.

keys, legal documents).

In consequence, 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 Herzberg *et al.* (1995) 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 over several periods.

Let *consistent shares* designate shares which, when combined in a sufficient number, make possible the calculation of s. The goal of proactive secret sharing is to harden the difficulty of an adversary being able to collect a set of $k+1$ consistent shares. This is done by periodically changing the shares, assuring that the interval between consecutive share rejuvenations is not sufficient for

an adversary to collect $k+1$ (consistent) shares.

5.1.2 The Proactive Secret Sharing Wormhole

In this section, we address the exhaustion-safety of distributed systems based on secret sharing, i.e., the assumption φ_{node} is $n_{fail} \leq k$, where n_{fail} represents the number of consistent shares that, during an execution, are disclosed simultaneously. A share is considered disclosed when it is known by an adversary.

We propose the Proactive Secret Sharing Wormhole (PSSW) as an instantiation of the PRW$^d\langle F, T_D, T_P, T_\pi\rangle$ presented in Section 4.1. Notice that this means that there exists a single cluster composed of all local PSSWs and therefore all local PSSWs are interconnected by the same synchronous and secure control network. The PSSW targets distributed systems which are based on secret sharing and the goal of the PSSW is to periodically rejuvenate the secret share of each node, so that the overall system is exhaustion-safe wrt φ_{node}.

The presentation of the PSSW is divided in two parts. The first part describes the procedure *refresh_share* that renews the shares without changing or disclosing the secret, and enumerates the assumptions that need to be ensured in the construction of the PSSW. The second part discusses how the values of T_P, T_D and T_π may be chosen in order to ensure that a secret sharing system enhanced with a PSSW = PRW$^d\langle$ *refresh_share* $, T_D, T_P, T_\pi\rangle$ is exhaustion-safe wrt φ_{node}. The choice of the values T_D, T_P, T_π is conditioned by the PSSW assumptions, including the assumed adversary power.

The PSSW executes Algorithm 1 (page 47) in order to periodically and timely execute the procedure *refresh_share* presented in Algorithm 2 (page 80). This procedure is based on the share renewal scheme of Herzberg *et al.* (1995). In lines 1–2, local PSSW i picks k random numbers $\{\delta_{im}\}_{m\in\{1\ldots k\}}$ in $[0, q[$. These numbers define the polynomial $\delta_i(z) = \delta_{i1}z^1 + \delta_{i2}z^2 + \ldots + \delta_{ik}z^k$. In lines 3–6,

local PSSW i sends the value $u_{ij} = \delta_i(j) \bmod q$ to all other local PSSWs j. Then, in lines 7–9, local PSSW i receives the values u_{ji} from all other local PSSWs. These values are used to calculate, in line 10, the new share. Notice that the calculation is done by combining the previous share with a sum of the random numbers sent by each local PSSW, and that, in the execution of the first refreshment, the previous share corresponds to the initial share $f(i)$.

In this work it is not described how the payload applications obtain the share. We envisage that this could be done in two different ways, either through a PSSW library composed by functions that could be used to access the current value of the share, or by resorting to a multi-port memory periodically written by the local PSSWs and with read-only access by the payload applications. In both approaches, it should be guaranteed that the payload applications are aware of the current version of the shares.

Note that, after the termination of the procedure *refresh_share* in all local PSSWs, the time necessary for condition φ_{node} to be violated is extended. The system is exhaustion-safe if the interval between consecutive rejuvenations is not sufficient for φ_{node} to be violated. Next we present the assumptions that the PSSW must satisfy in order to guarantee the correct and timely execution of *refresh_share* .

A1 There exists a known upper bound $T_{proc_{max}}$ on local processing delays.

A2 There exists a known upper bound $T_{drift_{max}}$ on the drift rate of local clocks.

A3 Any network message is received within a maximum delay $T_{send_{max}}$ from the send request.

A4 The content of the network traffic cannot be read by unauthorized users.

In what follows it is first proved that the *refresh_share* function has a bounded execution time when executed under assumptions A1–A4. Then, it is shown

Algorithm 2: *refresh_share* procedure executed by each local PSSW i

initialization: $share \leftarrow f(i)$

begin

/* Define the polynomial $\delta_i(z) = \delta_{i1}z^1 + \delta_{i2}z^2 + ... + \delta_{ik}z^k$ using $\{\delta_{im}\}_{m\in\{1...k\}}$ */

1 **for** $m = 1$ *to* k **do**

2 $\quad \delta_{im} \leftarrow$ generate_random_number($[0, q[$)

/* Send $\delta_i(j)$ to each P_j */

3 **for** $j = 1$ *to* n **do**

4 $\quad$ **if** $j \neq i$ **then**

5 $\quad\quad u_{ij} \leftarrow \delta_i(j) \bmod q$

6 $\quad\quad$ send u_{ij} to P_j

/* Receive $\delta_j(i)$ from each P_j */

7 **for** $j = 1$ *to* n **do**

8 $\quad$ **if** $j \neq i$ **then**

9 $\quad\quad$ receive u_{ji} from P_j

/* Calculate the new share */

10 $share \leftarrow (share + u_{1i} + u_{2i} + ... + u_{ni}) \bmod q$

end

that it is possible to build a PSSW and that by choosing appropriate values for T_D, T_P, T_π, and k, one can have an exhaustion-safe intrusion-tolerant secret sharing system.

Theorem 5.1.1. *If all local PSSWs execute Algorithm 1 with F=*refresh_share *under assumptions A1–A4, then:*

Bounded execution time *Once all nodes are in the same round, there is an upper bound $T_{exec_{max}}$ on the execution time of* refresh_share *, i.e., the difference between the real time instant when the last node starts executing* refresh_share *and the real time instant when the last node finishes executing is not greater*

than $T_{exec_{max}}$.

Robustness *After all nodes finish the execution of each round of* refresh_share *, the new shares computed correspond to the initial secret (i.e., any subset* $k+1$ *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 Herzberg *et al.* (1995). The proof of Secrecy uses assumption A4.

Bounded execution time:

We shall prove a stronger result: assuming that all nodes are ready to execute *refresh_share* , i.e., all nodes are in the same round, the difference between the real time instant when the *first* node starts executing *refresh_share* and the real time instant when the last node finishes executing is not greater than $T_{exec_{max}}$. Let I be the set of all instructions used in each execution round of *refresh_share* (i.e., all instructions executed between lines 1 and 10). Let T_{exec_i} be a bound on the execution time of instruction $i, \forall i \in I$. Given that the execution time of any instruction, with the exception of *receive*, depends only on the local processing delays, let $T_{proc_{max}}$ be an upper bound on the execution time of such instructions (assumption A1). This entails that $T_{exec_i} < T_{proc_{max}}, \forall i \in I \setminus \{receive\}$. The execution time of *receive* depends on the local processing and network delivery delays, such that, $T_{exec_{receive}} < T_{proc_{max}} + T_{send_{max}}$ (assumption A2). Therefore, one can upper bound the execution time of the algorithm by $T_{exec_{max}} = (7n + 2k - 2)T_{proc_{max}} + (n-1)T_{send_{max}}$. This value results from the following calculations. The instructions in lines 1 and 2 are within a cycle with k iterations. Thus, their total execution time is bounded by $2kT_{proc_{max}}$. Then, the instructions in lines 3, 4, 5 and 6 are executed in the context of a cycle with

n iterations. However, lines 5 and 6 are not executed in one of the iterations given that $0 < i \leq n$. This means that the total execution time of lines 3 and 4 is bounded by $2nT_{proc_{max}}$ and that the total execution time of lines 5 and 6 is bounded by $2(n-1)T_{proc_{max}}$. Following the same logic, the total execution time of lines 7 and 8 is bounded by $2nT_{proc_{max}}$. Regarding line 9, given that it includes the instruction *receive*, its maximum execution time is bounded by $(n-1)(T_{proc_{max}} + T_{send_{max}})$. Finally, the execution time of line 10 is bounded by $T_{proc_{max}}$. ■

According to Theorem 5.1.1, *refresh_share* is a distributed procedure appropriate for rejuvenating the secret shares of a distributed system: upon termination of a round of *refresh_share* , all the nodes have new shares (and, hence, are not corrupted) and; once all nodes are in the same round, there exists a known upper bound $T_{exec_{max}}$ on the execution time of *refresh_share* . The following proposition shows that is possible to use this rejuvenation procedure *refresh_share* to build a PSSW that offers a periodic timely execution service.

Proposition 5.1.2. *Let PSSW be a PRW^d built under assumptions A1–A4 and triggering the* refresh_share *procedure through the execution of Algorithm 1 with* $\delta = 4T_{proc_{max}} + T_{drift_{max}}$. *Let* $T_P, T_D, T_\pi \in \Re_0^+$ *such that*

a) $T_P > T_D + T_\pi + \delta$

b) $T_D \geq T_{exec_{max}}$

c) $T_\pi \geq T_{proc_{max}} + T_{send_{max}}$

Then, the PSSW offers the periodic timely execution service $\langle$ refresh_share , $T_D, T_P, T_\pi \rangle$.

Proof: Given that $T_D + T_\pi < T_P$ due to a), we only need to show that conditions 1, 2, 3 and 4 of Definition 4.1.1 are satisfied by the PSSW under assumptions A1–A4. Consider the Algorithm 1 executed by the PSSW.

Condition 1 This condition is trivially satisfied given that the PSSW is composed by a single cluster and every local PSSW executes *refresh_share* .

Condition 2 Without line 2, local PSSW ${C_1}^1$ would execute F within $4T_{proc_{max}} + T_{exec_{max}}$ from the last triggering, given that the procedure F and four instructions would be executed between consecutive triggering. Therefore, setting $T_P \geq 4T_{proc_{max}} + T_{exec_{max}}$ would satisfy condition 2. The addition of the *wait* instruction in line 2 potentially decreases the frequency of F execution in order to enforce a certain periodicity that is sufficient to guarantee exhaustion-safety. Notice that this addition is in fact weakening the system, but it is necessary to minimize the potential overhead provoked by each rejuvenation, and in order to guarantee that consecutive rejuvenations do not overlap (in different local PSSWs). Regarding the value of δ, if the local PSSW clocks were perfect, one could set $\delta = 4T_{proc_{max}}$ in order to satisfy condition 2, as long as the chosen T_P would be greater than $T_{exec_{max}} + 4T_{proc_{max}}$. However, according to assumption A2, local PSSW clocks have a bounded drift rate $T_{drift_{max}}$. Therefore, given that δ has also to cancel this drift rate, we have that if $\delta = 4T_{proc_{max}} + T_{drift_{max}}$ and $T_P > T_{exec_{max}} + \delta$, the PSSW satisfies condition 2.

Condition 3 First of all, given that $T_P > T_D + T_\pi + \delta$, consecutive executions of *refresh_share* do not overlap. This means that whenever the local PSSW ${C_1}^1$ finishes waiting in line 2, all other local PSSWs are already ready to receive the message *trigger* and start a new round. Therefore, the difference between the *refresh_share* triggering instants on every local PSSW depends on the delivery delay and processing of the message *trigger* sent by local PSSW ${C_1}^1$ in line 4 and received by every other local PSSW in line 6. This means that setting $T_\pi \geq T_{proc_{max}} + T_{send_{max}}$ allows the PSSW to satisfy condition 3.

Condition 4 According to Theorem 5.1.1, the PSSW satisfies condition 4 if $T_D \geq T_{exec_{max}}$. ■

As a corollary of Theorem 4.1.6, we have that under some conditions, a secret sharing system S enhanced with an appropriate PSSW is exhaustion-safe wrt φ_{node}. As before, we use $[\![S]\!]$ to denote the set of executions of a secret sharing system S under the REX model for assumption φ_{node}.

Corollary 5.1.3. *Suppose that*

1. *S is a secret sharing system composed of a total of n nodes, each one with a share that never changes, and let* $T_{exh_{min}} = \inf(\{T^{\mathcal{E}}_{exh} : \mathcal{E} \in [\![S]\!]\})$;

2. *The time needed to discover* $k+1$ $(\leq n)$ *shares at any instant is independent of the number of shares that are currently known.*

Then, the system S enhanced with a PSSW s.t. $T_P + T_D + T_\pi < T_{exh_{min}}$ *is exhaustion-safe w.r.t.* φ_{node}.

Proof: This result is a straightforward consequence of Theorem 4.1.6. Notice that assumption 3 of that theorem is entailed by the robustness property of *refresh_share* , as stated in Theorem 5.1.1. ■

All these results are based on the assumption that no local PSSW crashes during the lifetime of the system. Section 4.1 described generically how one could build a fault-tolerant PRW able to resist f_c crashes. Here it is explained more concretely how could be that done in the context of the PSSW.

Consider a PSSW composed by a total of n local PSSWs, and assume that at most f_c local PSSWs crash during the lifetime of the system, such that $n \geq f_c + k + 1$ (this condition guarantees that it is always possible to reconstruct the secret). In such a system and under assumptions A1 and A3 it is possible to build a leader election protocol (Garcia-Molina, 1982). This protocol could

be used in Algorithm 1 to tolerate the fault of local PSSW $\mathcal{C}_1$[1]. In each round, the *leader* would be the responsible for sending the message *trigger*. In the case of a leader crash, the following leader would be then the responsible, and so on. The parameter δ would have to take into consideration the worst case execution time of the leader election protocol. Regarding the *refresh_share* procedure presented in Algorithm 2, each local PSSW would have to resort to a perfect failure detector (Chandra & Toueg, 1996) in order to detect the crash of the other PSSWs and avoid waiting forever for messages from failed PSSWs. Under assumptions A1 and A3, it is possible to build a perfect failure detector with bounded detection time. This bound would then be used in the calculation of $T_{exec_{max}}$.

5.1.3 Experimental Results

We have implemented a prototype[1] of the PSSW using RTAI (Cloutier *et al.*, 2000), an operating system with real-time capabilities, and a switched Fast-Ethernet control network. The feasibility of achieving timeliness guarantees using this type of operating system and network are discussed in Casimiro *et al.* (2000). RTAI allows the construction of an architecturally-hybrid execution environment (Veríssimo, 2006), with the PSSW executing as a set of real-time tasks, and the normal applications executing at Linux user-level.

The PSSW prototype makes use of the GNU Multiple Precision Arithmetic Library (GMP)[2], a free library for arbitrary precision arithmetic. The Linux version of the GMP library was ported to RTAI, and it is available together with the PSSW prototype source code.

This section presents the results of a set of experiments that were conducted

[1] Available at `http://sourceforge.net/projects/rt-pss/`
[2] `http://www.swox.com/gmp`

using this prototype, with the goal of observing the execution time of the *refresh_share* procedure (Algorithm 2) when triggered in the context of the PSSW periodic timely execution service (Algorithm 1). More precisely, the measurements that will be presented represent the interval of time between the first local PSSW triggering the procedure and the last PSSW finishing executing it. These measurements allow one to study: the possible values of T_P, T_D and T_π in a real environment; predict the types of adversary it is possible to resist; determine the cost of the rejuvenation overhead (i.e., rejuvenation time vs total execution time).

The experimental infrastructure was composed by 500 MHz single-processor Pentium III based PCs running RTAI, and interconnected by a 3COM SuperStack II Baseline 100 Mbps switch. The experiments presented below used 1024-bit shares. The share can be any type of data. It can be, for instance, a 1024-bit RSA key, and in this case, the PSSW could be used as part of a proactive threshold RSA scheme (Rabin, 1998). The results of every configuration are based on the analysis of 65535 periodic executions triggered by the PSSW.

The first experiment tested configurations from 2 to 6 machines with $k = 1$. Remember that the exhaustion-safety condition is $n_{fail} \leq k$, where n_{fail} represents the number of consistent shares that, during an execution, are disclosed simultaneously. The goal was to evaluate the overhead introduced by the algorithm when the number of machines increases. The results (mean, standard deviation, minimum and maximum execution time) are presented in Table 5.1. One of the main conclusions is that the mean execution time increases with the number of machines. This was expected given that more machines require more messages to be exchanged and thus greater processing and network delays. The maximum execution time, however, remains quite stable independently of the number of machines. This is very important and shows in

practice that there exists an upper bound $T^{\pi}_{exec_{max}}$ on the execution time (notice that $T^{\pi}_{exec_{max}}$ corresponds to the interval between the first local PSSW triggering the refresh and the last PSSW finishing executing it, whereas the bound $T_{exec_{max}}$ mentioned in section 5.1.2 does not include the interval between the first and the last triggerings). Moreover, these measurements also allow us to conclude that one could trigger a rejuvenation every 2 seconds with a maximum overhead of less than 2% (given that $T^{\pi}_{exec_{max}} < 30ms$, one could say that $T_D + T_\pi = 30ms$ and set $T_P = 2000ms$). An adversary would have to obtain $k + 1 = 2$ shares in less than 2.1 seconds ($\approx T_P + T_D + T_\pi$) in order to reconstruct the protected secret. In Figure 5.2 it is possible to observe the distribution of the (65535) execution times of the experiment in the configuration with 6 machines and $k = 1$. In terms of probability distribution it is clear that the probability of execution time values above 24 msec is very low.

n	T_{exec} $(msec)$		
	mean	std dev.	min, max
2	11.4	3.4	10.0, 20.0
3	15.0	3.1	10.1, 22.3
4	17.0	2.6	10.8, 22.3
5	18.1	2.3	11.3, 23.0
6	19.0	1.5	15.4, 22.8

Table 5.1: *refresh_share* execution time with $k = 1$ (n – number of machines).

The next step was to evaluate the impact of increasing k. Notice that increasing k means that one is attempting to resist a stronger adversary, in other words, resisting the disclosure of a higher number of shares. Therefore, in the second experiment, 6 machines were used to test the behavior of the system with k varying between 1 and 5. The results are presented in Table 5.2. One can see that there is an increment in the mean and maximum execution when k increases. This increment is also visible in the execution time distribution de-

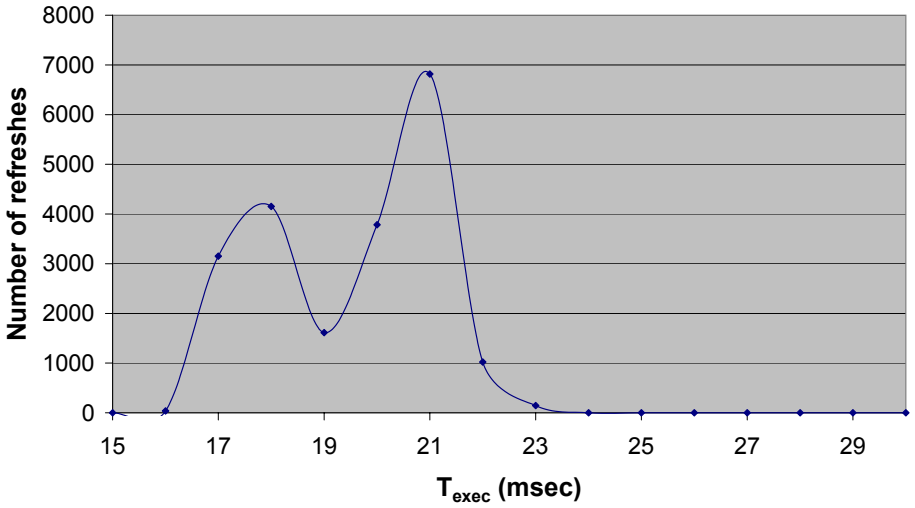

Figure 5.2: *refresh_share* execution time distribution with 6 machines and $k = 1$.

picted in Figure 5.3 and it happens because the size of k impacts the processing delay. Nevertheless, the maximum execution time for $k = 5$ remains still under 30 ms. This means that one can extend the previous conclusions and say that an adversary would have to obtain 6 shares in less than 2.1 seconds in order to discover the secret.

k	$T_{exec}(msec)$		
	mean	std dev.	min, max
1	19.0	1.5	15.4, 22.8
2	19.8	1.9	12.7, 24.1
3	20.6	2.0	14.6, 25.4
4	21.3	2.3	14.2, 26.4
5	22.6	2.4	14.8, 27.4

Table 5.2: *refresh_share* execution time with 6 machines.

Depending on the assumed adversary strength and on the desired overhead, the system architect can use the above results to calculate the appropriate degree of fault-tolerance k and the values of T_P, T_D and T_π. To illustrate how can this be done, two different adversary types (*Hare* and *Tortoise*) are

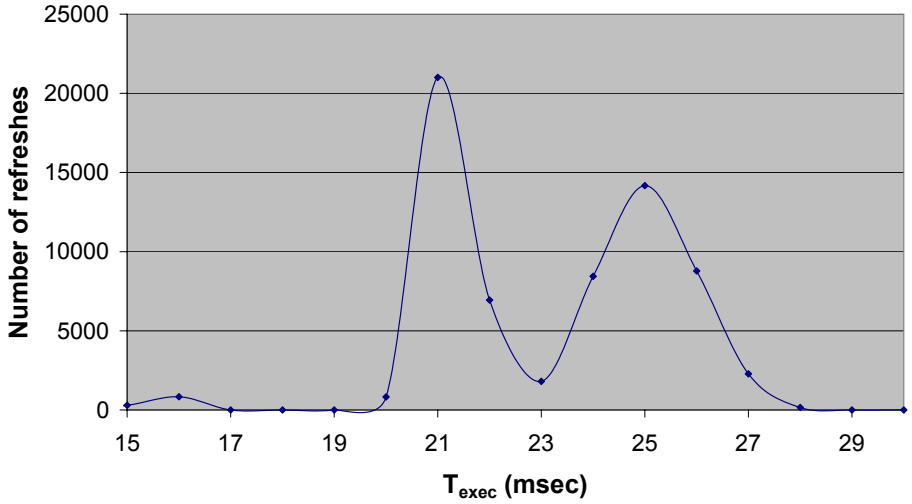

Figure 5.3: *refresh_share* execution time distribution with 6 machines and $k = 5$.

presented next, and it is described how to configure an appropriate PSSW in each scenario. In both scenarios the system is deployed with 6 machines.

Hare This adversary is able to compromise any machine (i.e., disclose a single share) in one second. Such an adversary can be envisaged in the context of ultra-resilient systems (e.g., national security related) defending against fierce cyber-attacks.

Without proactive secret sharing, *Hare* would take $k + 1$ seconds to discover $k + 1$ shares and reconstruct the secret. With 6 machines and $k = 5$, this would mean that the system could be compromised after 6 seconds.

In order to resist *Hare*, one has to build a PSSW able to refresh $k + 1$ shares in less than $k + 1$ seconds. Table 5.3 compares the resulting overhead of choosing different values of k and the corresponding maximum value of T_P. The overhead is calculated using the formula $\frac{T_D+T_\pi}{T_P+T_D+T_\pi}$, with $T_D + T_\pi = 30$. The conclusion is that independently of the value of k, it is possible to defend against *Hare* with a negligible overhead. Therefore,

one can say that the PSSW can be used efficiently to secure secret sharing systems even in the presence of very powerful adversaries.

k	max T_P	overhead
1	1.9 sec	1.6%
2	2.9 sec	1.0%
3	3.9 sec	0.8%
4	4.9 sec	0.6%
5	5.9 sec	0.5%

Table 5.3: PSSW overhead in order to resist *Hare*.

Tortoise This adversary is slower than *Hare*, being able to compromise any machine (i.e., disclose a single share) in one minute. *Tortoise* may be used to model typical cyber-attacks on the web.

Without proactive secret sharing, *Tortoise* would take $k+1$ minutes to discover $k+1$ shares and reconstruct the secret. With 6 machines and $k = 5$, this would mean that the system could be compromised after 6 minutes.

In order to resist *Tortoise*, one has to build a PSSW able to refresh $k+1$ shares in less than $k+1$ minutes. Table 5.4 compares the resulting overhead of choosing different values of k and the corresponding maximum value of T_P. The overhead is calculated using the same formula as above with $T_D + T_\pi = 30$. As expected, the overhead is significantly lower than when defending against *Hare*. Therefore, the conclusion is that the PSSW can also increase the resilience of money-critical secret sharing systems deployed on the web.

To the best of our knowledge, we are the first to present and evaluate a proactive secret sharing implementation in a real time environment. In Barak *et al.* (1999), a Java prototype of a proactive security toolkit (using the same

k	max T_P	overhead
1	119.9 sec	0.03%
2	179.9 sec	0.02%
3	239.9 sec	0.01%
4	299.9 sec	0.01%
5	359.9 sec	0.01%

Table 5.4: PSSW overhead in order to resist *Tortoise*.

PSS protocol our PSSW is based on) is presented, but authors do not discuss the temporal guarantees of their approach. The work presented in Zhou *et al.* (2005) describes APSS, a proactive secret sharing protocol for asynchronous systems. APSS is in theory a fine replacement of PSS protocols in asynchronous environments. However, to be useful, APSS needs to be executed with guaranteed periodicity and, by definition, this cannot be guaranteed in asynchronous conditions (see Section 2.2.1). Nevertheless, one could envision a PSSW using APSS instead of the synchronous PSS it currently uses. We leave this as future work.

5.2 Resilient and Available State Machine Replication

5.2.1 Motivation

Nowadays, one of the major concerns about the services provided by computer systems is related to their availability. This applies specially to services provided over the Internet. Building highly available services involves, on the one hand, the design and implementation of correct services that are tolerant to a wide set of faults, and on the other hand, the assurance that the access to them is always guaranteed with a high probability. Interestingly, these two

tasks can both be accomplished by employing replication techniques.

Replication is a well-known way of improving the availability of a service: if a service can be accessed through different independent paths, then the probability of a client being able to use it increases. But replication has costs, namely it is necessary to guarantee a correct coordination between the servers. Moreover, the Internet being an unpredictable and insecure environment, coordination correctness should be assured under the worst possible operational conditions, i.e., the absence of local or distributed timing guarantees and the possibility of arbitrary faults triggered by malicious adversaries. Several past works addressed agreement and replication techniques that tolerate arbitrary faults under the asynchronous model. The majority of these techniques make the assumption that the number of faulty replicas is bounded by a known value (Bracha & Toueg, 1985; Cachin *et al.*, 2000; Canetti & Rabin, 1993; Correia *et al.*, 2004; Doudou *et al.*, 1999; Malkhi & Reiter, 1997a,b, 2000). However, under the asynchronous model, this type of assumption may be problematic. As shown in Section 3.2.2, there is no way of ensuring that no more than f faults will occur during the execution of the system offering the service (arbitrary processing and message delays may result in a very long execution time).

5.2.2 State Machine Replication

A state machine is defined by a set of state variables and a group of commands. The collection of state variables defines the state of the system. Commands are used to perform modifications on the state variables and/or to produce some output (e.g., read the value of a state variable) (Lamport, 1978; Schneider, 1990). Almost every computer program can be modeled as a state machine. In particular, we will focus on client/server applications, which also fit under this

model: the server is responsible for maintaining the state and the clients issue commands that modify or read the state. This way of looking at client-server applications facilitates the reasoning on how to make this type of applications fault-tolerant. The simplest way of implementing a client-server application is by deploying a single centralized server that processes all the commands issued by clients. As long as the server does not fail, commands are performed according to the order they are received from clients. But if faults may happen, then this centralized approach does not work. The server may crash and render the system unavailable or worst, the server may be compromised by some malicious adversary, which can arbitrarily modify the state. In order to tolerate these types of faults, one has to replicate the server. The replication degree depends both on the type (e.g., crash, Byzantine) and quantity of the faults to be tolerated. As such, some protocols allow to implement state machine replication tolerant to crash faults (Lamport, 1998; Oki & Liskov, 1988; Skeen, 1982), whilst others target the Byzantine scenario (Amir *et al.*, 2006; Castro & Liskov, 2002; Correia *et al.*, 2004; Reiter, 1995). In the present work, we do not make any restrictions on the type of faults than can happen – a server may fail arbitrarily, either by crash or by compromise of the state and/or the execution logic. The current state of the art allows one to build client/server applications resilient to a specified number f of arbitrary faults.

f fault-tolerant replicated systems (with $f \geq 1$) are not unconditionally resilient to failure. In fact, the actual resilience of the replicated system depends both on the correlation between replica failures and on the strength of the malicious adversary. On the one hand, if all replicas use the same design and implementation (operating system, protocols, applications), then an adversary only needs to discover how to compromise a single replica in order to (easily) compromise the remaining replicas. On the other hand, even if the replicas

are diverse, a malicious adversary with the ability of triggering attacks in parallel may substantially reduce the time needed to corrupt more than f replicas. Moreover, in long-lived systems, whether or not replicas are attacked in sequence, the probability of more than f replicas being compromised is significant, given enough time. From this reasoning, we identify three key factors that influence the actual resilience of an f fault-tolerant system:

- Diversity of replicas: design and implementation;
- Type of attack: in sequence or in parallel;
- Service duration: total execution time.

The first two factors determine the time needed to corrupt more than f replicas. By assessing if this value is greater or lower than the third factor, the total execution time of the system, one can determine the resilience of an f fault-tolerant replicated system.

These systems can only be highly resilient if one can guarantee that at most f faults occur simultaneously during system execution. This goal should be ensured by construction at design time, and proactive recovery seems to be a very interesting approach to achieve it: replicas are periodically rejuvenated and thus the effects of accidental and/or malicious faults can be removed. However, proactive recovery execution needs some synchrony guarantees in order to ensure regular triggering and bounded execution time.

We propose to apply the Proactive Resilience Model (PRM), presented in Section 4.1, to the state machine replication scenario. The PRW periodic timely execution service is used to proactively recover replicas, ensuring that:

- no more than f replicas are ever corrupted;
- the execution of the distributed state machine is never interrupted.

Our approach is minutely explained in the next section.

5.2.3 The State Machine Proactive Recovery Wormhole

We propose the State Machine Proactive Recovery Wormhole (SMW) as an instantiation of the PRW$^{l}\langle F, T_D, T_P, T_\pi\rangle$ presented in Section 4.1. This means that there exists one replica per cluster, and that no control network is used. Figure 5.4 depicts the architecture of a system with an SMW. The goal of the SMW is to periodically rejuvenate replicas such that no more than f replicas are ever compromised and thus node-exhaustion-safety is guaranteed. Moreover, recoveries should not affect the overall system availability.

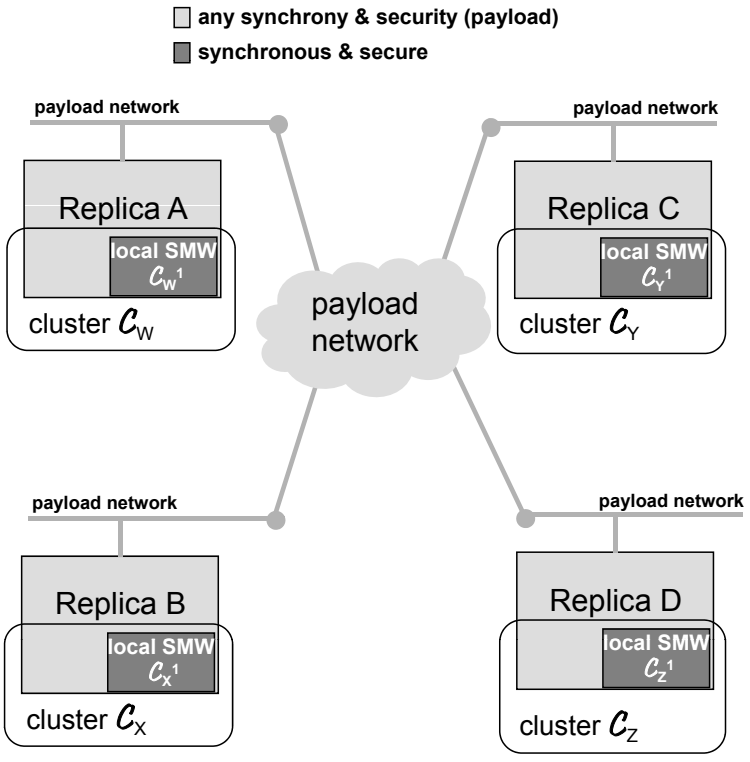

Figure 5.4: The architecture of a state machine replicated system with an SMW.

The SMW executes Algorithm 1 (page 47) in order to periodically and timely

execute the procedure *refreshCodeAndState* presented in Algorithm 3 (page 97). Notice that there is a single local SMW $\mathcal{C}_i{}^1$ in each cluster $\mathcal{C}_i$. Therefore, there is no need to multicast the trigger message in line 4 of Algorithm 1.

In Algorithm 3, line 1 shutdowns the payload operating system, and consequently stops the execution of the local state machine. Notice that the algorithm continues to execute even after the payload operating system being shutdown. This happens because of architectural hybridization: the SMW is a distinct subsystem, which can be achieved in practice, for instance, by implementing each local SMW in a PC appliance board (Kent, 1980; Trusted Computing Group, 2004), or by making use of virtualization techniques (Barham *et al.*, 2003). Line 2 checks if the operating system code is corrupted. To accomplish this task, a digest of the operating system code can be initially stored in some read-only memory, and then the digest of the current code is compared with the stored one. In line 3, the operating system code can be restored from a read-only medium, such as a Read-Only Memory (ROM) or a write-protected hard disk (WPHD), where the write protection can be turned on and off by setting a jumper switch (e.g., Fujitsu MAS3184NP). In order to introduce some diversity, different versions of the operating system may be used in each recovery. It is even possible to use different operating systems, but then it would be necessary to have a version of the state machine code for each of them. In lines 4–5, the state machine code can be checked and restored using similar methods to the ones used to check and restore the operating system code. Alternatively, both the operating system and the state machine code can be installed on a read-only medium, thus avoiding the execution of lines 2–5. Line 6 boots the operating system from a clean code and thus brings it to a correct state. We assume that the local state machine is automatically started once the operating system finishes the booting process.

Algorithm 3: *refreshCodeAndState* procedure executed by each local SMW.

```
   begin
       /* shutdown the payload operating system                    */
1      shutdownOS()
       /* restore operating system code if corrupted               */
2      if OS code is corrupted then
3          restoreOScode()
       /* restore state machine code if corrupted                  */
4      if SM code is corrupted then
5          restoreSMcode()
       /* at this point, the OS and the SM can be safely booted because their
          code is correct                                          */
6      bootOS()
   end
```

Given that the state of the local state machine may have been compromised before the rejuvenation, it may be necessary to transfer a clean state from remote replicas. We assume that this state transfer/recovery is done automatically by the state machine code once it starts. Castro & Liskov (2002) present an efficient mechanism to perform checkpoints and state transfer, specially tailored for state machine replication subject to Byzantine faults. However, depending on the synchrony guarantees of the network, state recovery may have an unbounded execution time, given that it requires the exchange of information through the payload network. Since the payload network is potentially asynchronous, messages sent through it can take an unbounded time to be delivered. Nevertheless, one can estimate an upper-bound on the delivery time which will be satisfied with high probability in normal conditions. In the worst-case, if abnormal delays occur, the availability of the replicated state machine may be affected but safety is preserved. This issue will be discussed in Section 5.2.4.1.

After the termination of the rejuvenation procedure *refreshCodeAndState* in a

local SMW, the corresponding replica is correct. The system is exhaustion-safe w.r.t a given adversary if rejuvenations are organized in a way such that the adversary does not have enough time to compromise more than f replicas between rejuvenations. Next we present the assumptions that the SMW must satisfy in order to guarantee the correct and timely execution of *refreshCodeAndState* in every replica.

A1 There exists a known upper bound $T_{proc_{max}}$ on local processing delays.

A2 There exists a known upper bound $T_{drift_{max}}$ on the drift rate of local clocks.

In what follows it is first proved that the *refreshCodeAndState* procedure has a bounded execution time when executed under assumption A1. Then, it is shown that it is possible to build an SMW under assumptions A1 and A2, and that by choosing appropriate values for T_D and T_P, one can have an exhaustion-safe intrusion-tolerant replicated state machine system.

Theorem 5.2.1. *If a local SMW executes Algorithm 1 with F=*refreshCodeAndState *under assumption A1, then there is an upper bound $T_{exec_{max}}$ on the execution time of* refreshCodeAndState .

Proof: In the worst-case scenario, i.e., when the code of both the operating system and the local state machine are corrupted, Algorithm 3 (*refreshCodeAndState*) executes a total of 6 operations. The execution time of these operations can be upper-bounded in the following manner.

- $shutdownOS()$: Typically, an operating system can be shutdown through a hardware interrupt. This operation has predictable execution time and thus one can define an upper-bound $T_{shutdown}$.
- check OS/SM code correctness: This can be implemented through a bitwise comparison between the digest of the current OS/SM code with a

digest initially stored in a read-only medium. An upper-bound $T_{checkcode}$ on the execution time of this operation can be defined based on the maximum time necessary to copy the digest from the medium and to make a simple bitwise comparison.

- *restoreOScode*() and *restoreSMcode*(): Restoring the OS/SM code corresponds to a copy from a read-only medium. So, an upper-bound $T_{restorecode}$ can be defined based on the maximum time necessary to copy the code from the medium.

- *bootOS*(): Booting an operating system can take some time, but given that we are always booting the same OS (because no application gets installed between boots), it is possible to estimate an upper-bound T_{boot} on the boot time.

So, one can define an upper-bound $T_{exec_{max}}$ on the execution time of Algorithm 3, such that $T_{exec_{max}} = T_{shutdown} + 2T_{checkcode} + 2T_{restorecode} + T_{boot}$.

■

The following proposition shows that it is possible to use this rejuvenation procedure *refreshCodeAndState* to build an SMW that offers a periodic timely execution service.

Proposition 5.2.2. *Let SMW be a PRW[l] built under assumptions A1–A2 and triggering the* refreshCodeAndState *procedure through the execution of Algorithm 1 with* $\delta = 4T_{proc_{max}} + T_{drift_{max}}$. *Let* $T_P, T_D, T_\pi \in \Re_0^+$ *such that*

a) $T_P > T_D + T_\pi + \delta$

b) $T_D \geq T_{exec_{max}}$

c) $T_\pi = 0$

Then, the SMW offers the periodic timely execution service $\langle$ refreshCodeAndState $, T_D, T_P, T_\pi \rangle$.

Proof: Given that $T_D + T_\pi < T_P$ due to a), we only need to show that conditions 1, 2, 3 and 4 of Definition 4.1.1 are satisfied by the SMW under assumptions A1–A2. Consider the Algorithm 1 executed by the SMW.

Condition 1 This condition is trivially satisfied given that the SMW clusters are composed of a single local SMW that executes *refreshCodeAndState* .

Condition 2 Without line 2, the local SMWs ${\mathcal{C}_i}^1$ would execute F within $4T_{proc_{max}} + T_{exec_{max}}$ from the last triggering, given that the procedure F and four instructions would be executed between consecutive triggering. Therefore, setting $T_P \geq 4T_{proc_{max}} + T_{exec_{max}}$ would satisfy condition 2. The addition of the *wait* instruction in line 2 potentially decreases the frequency of F execution in order to enforce a certain periodicity that is sufficient to guarantee exhaustion-safety. Regarding the value of δ, if the local SMW clocks were perfect, one could set $\delta = 4T_{proc_{max}}$ in order to satisfy condition 2, as long as the chosen T_P would be greater than $T_{exec_{max}} + 4T_{proc_{max}}$. However, according to assumption A2, local SMW clocks have a bounded drift rate $T_{drift_{max}}$. Therefore, given that δ has also to cancel this drift rate, we have that if $\delta = 4T_{proc_{max}} + T_{drift_{max}}$ and $T_P > T_{exec_{max}} + \delta$, the SMW satisfies condition 2.

Condition 3 Given that there is only one replica per cluster, this condition is trivially satisfied with $T_\pi = 0$.

Condition 4 According to Theorem 5.2.1, the SMW satisfies condition 4 if $T_D \geq T_{exec_{max}}$. ■

Following the reasoning presented in Section 4.1.2, we have that (1) if one enhances a replicated state machine with an SMW and (2) if it is possible to

lower-bound the exhaustion time (i.e., the time needed to produce $f + 1$ replica failures) of every system execution by a known constant $T_{exh_{min}}$, then node-exhaustion-safety is achieved by assuring that $T_P + T_D < T_{exh_{min}}$.

5.2.4 Achieving Both Node-Exhaustion-Safety and Availability

We now discuss the recovery strategy to be applied in order that no more than f replicas are ever corrupted, and the execution of the distributed state machine is never interrupted.

A straightforward solution to achieve node-exhaustion-safety would be to rejuvenate all the replicas at once: the replicas would be simultaneously stopped, rejuvenated, and restarted again. Given that no progress would occur during rejuvenation, only the previously compromised replicas would have to restore their state. The problem with this solution is that the distributed state machine would be unavailable during the rejuvenation, which is contrary to one of our goals. However, in scenarios where the interruption of the service is not a problem, this solution has the advantage of minimizing the number of state transfers, given that only compromised states have to be restored.

In order to avoid service interruption, one needs to do two things:

1. Define the maximum number of replicas allowed to recover simultaneously (call it k). Note that a recovering replica may not process client requests until the recovery finishes. In this way, a recovering replica can be considered as being crashed from the perspective of the state machine replication algorithms (e.g., atomic broadcast (Défago *et al.*, 2004)).

2. Deploy the system with a sufficient number of replicas to tolerate f Byzantine servers and k crashed servers. We derived a new bound on the to-

tal number of replicas for *constantly available state machine replication* of $n \geq 3f + 2k + 1$, which will be explained next. The requirement that no more than f replicas be compromised between rejuvenations is obviously maintained.

Section 5.2.4.1 explains how one can coordinate recoveries such that no more than a certain number of replicas recover at the same time, then Section 5.2.4.2 clarifies why we need $n \geq 3f + 2k + 1$ replicas, and, finally, Section 5.2.4.3 discusses some recovering strategies and the results of combining different values of f and k.

5.2.4.1 Recoveries Coordination

The goal is to devise a way of enforcing an upper-bound k on the number of replicas that recover simultaneously. In other words, we want to ensure that, at any time, no more than k replicas are recovering. This goal can be achieved using different approaches, with distinct guarantees and cost.

One possible approach is to use a technology that has been readily available for some years: GPS (Parkinson & Gilbert, 1983) (Global Positioning System). GPS provides an extremely precise absolute time reference and thus can be used to maintain the local SMW clocks highly accurate and precise (Veríssimo *et al.*, 1997). Therefore, by enhancing the local SMWs with a GPS receiver, recoveries can be scheduled such that no more than k replicas are recovering at the same time. Although this approach offers high guarantees, it has the cost of buying a GPS receiver per local SMW.

A less costly approach is to use the payload network to (internally) synchronize the clocks of local SMWs. Although the payload network may be asynchronous, local SMWs are synchronous and thus have the ability to detect when the precision of the clock synchronization is not sufficient to guarantee

that at most k replicas are recovering (Fetzer & Cristian, 1997). In this case, an alarm could be triggered to warn the system administrator that problems were detected. Notice that, even with the GPS approach, one depends on the payload network if state transfer is needed after the recovery. A more detailed explanation is given next.

The recovery algorithm presented in Algorithm 3 does not include the state transfer operation that may be potentially necessary after a replica being rebooted. We consider that a replica is correct after being rebooted with clean code, but it may happen that correct replicas take some time to resume normal operation. We argue that the system architect should estimate a reasonable upper-bound $T_{reintegrate}$ on the time necessary for the replica to reintegrate normal operation, and then choose an adequate T_D (the bound on the recovery execution time) sufficient to allow recovery and reintegration. However, during unstable periods (e.g., a DoS attack), resuming normal operation may take more time than assumed, potentially leading to more than k recoveries at the same time, which may cause unavailability. Nevertheless, the system is always assuredly correct (as long as the power of the adversary is within the assumed limits). Moreover, if the network is under a DoS attack, then unavailability was most probably being already experienced by the service clients, and thus having more than k recoveries at the same time does not degrade significantly the QoS (Quality-of-Service).

In order to avoid complete unavailability, the SMW could be extended with a second service to be used by the state machine application running in the payload, to notify the SMW that reintegration was completed. In this way, each local SMW could monitor if the assumed reintegration times were being met, and take appropriate measures when problems were detected, such as warn the system administrator.

5.2.4.2 Why the Need for the $n \geq 3f + 2k + 1$ Bound?

Typically, a replicated state machine with $3f + 1$ replicas is able to tolerate up to f arbitrary faults (Pease *et al.*, 1980). This bound is associated with the specific atomic broadcast (or consensus) protocol used to guarantee the consistency of the replicated state machine. If more than f faults occur, both the safety and the liveness of the atomic broadcast protocol may be compromised. In particular, if $f + 1$ crash faults occur, the protocol will (normally) block. Therefore, a replicated state machine with $3f + 1$ replicas may become unavailable during recoveries when more than f of them are compromised/crashed.

Consider now that you have a replicated state machine system with n replicas, able to tolerate a maximum of f Byzantine faults, and where rejuvenations occur in groups of at most k replicas. At any time, the minimum number of replicas assuredly available is $n - f - k$. So, in any operation, either intra-replicas (e.g., an atomic broadcast execution) or originated from an external participant (e.g., a client request), a group with $n - f - k$ replicas will be used to execute the operation. Given that some of these operations may affect the state of the replicated system, one also needs to guarantee that any two groups of $n - f - k$ replicas intersect in at least $f + 1$ replicas (i.e., since f replicas can be malicious, this bound guarantees the participation of at least one correct replica). Therefore, we need to ensure that $2(n - f - k) - n \geq f + 1$, which can only be satisfied if $n \geq 3f + 2k + 1$.

The above result can also be obtained by making use of the Byzantine quorum systems theory (Malkhi & Reiter, 1997a), namely by analyzing the Byzantine dissemination quorum system construction. This construction applies to replicated services storing self-verifying data, i.e., data that only clients can create and about which clients can detect any attempted modification by a faulty server (e.g., public key distribution system). In a dissemination quorum

system, the following properties are satisfied:

Intersection any two quorums have at least one correct replica in common;

Availability there is always a quorum available with no faulty replicas.

If one designates $|Q|$ as the quorum size, then the above properties originate the following conditions:

Intersection $2|Q| - n \geq f + 1$;

Availability $|Q| \leq n - f - k$.

From these conditions, it results that we need $n \geq 3f + 2k + 1$ in order to have a dissemination quorum system in a environment where at most f replicas may behave arbitrarily, and at most k replicas may recover simultaneously (and thus become unavailable during certain periods of time). In the special case when $n = 3f + 2k + 1$, it follows that $|Q| = 2f + k + 1$.

5.2.4.3 Recovery Strategies

Each recovering replica executes the code presented in Algorithm 3. Replicas are recovered in groups of at most k elements, by some specified order: for instance, replicas $P_1, ..., P_k$ are recovered first, then replicas $P_{k+1}, ..., P_{2k}$ follow, and so on. A total of $\lceil \frac{n}{k} \rceil$ replica groups are rejuvenated in sequence. Figure 5.5 illustrates the rejuvenation process. The SMW coordinates the rejuvenation process (see Section 5.2.4.1), triggering the rejuvenation of replica groups one after the other. The maximum execution time of the rejuvenation process, i.e., the maximum time interval between the first group rejuvenation start instant and the last group rejuvenation termination instant, is upper-bounded by $T_{D^{global}} = \lceil \frac{n}{k} \rceil T_D$. Given that recoveries should not overlap if one wants to

guarantee availability (or else more than k replicas could end recovering at the same time), the system architect must choose a T_P value greater than $T_{D^{global}}$.

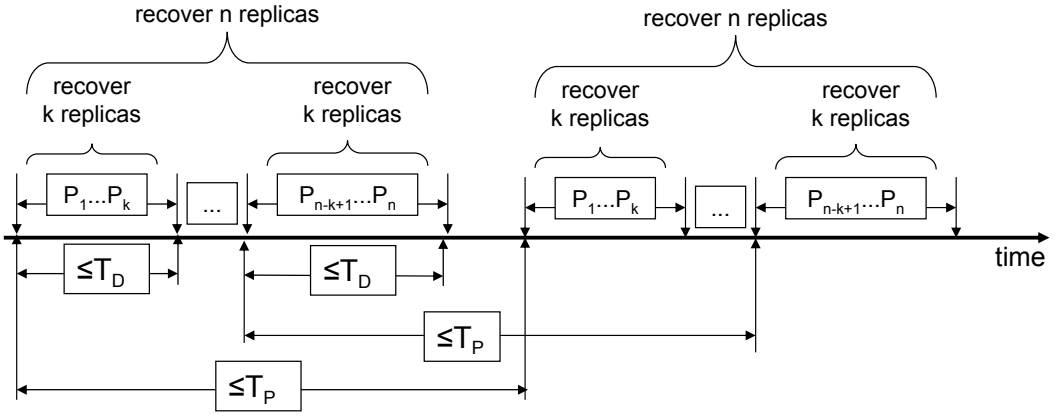

Figure 5.5: Relationship between the rejuvenation period T_P, the rejuvenation execution time T_D and k.

Therefore, when analyzing the feasibility of a concrete value of T_P, one should also take into consideration the different possibilities in terms of recovery strategies, and their impact on the recovery execution time. Thus, an important design parameter that should be considered is the value of k. This value defines the number of nodes that may recover simultaneously, and consequently the number of distinct $\lceil \frac{n}{k} \rceil$ replica groups that recover in sequence. Intuitively, choosing a higher k, reduces the number of rejuvenation groups, but increases the total number of nodes n (required to guarantee availability during recoveries). On the other hand, a smaller k means more recoveries in sequence, but a lower n. Remember that each rejuvenation group will reboot and, after recovery, potentially execute a state transfer. Both these operations may take a considerable amount of time, and therefore one should try to minimize the number of rejuvenation groups. This being said, let us analyze how the value of $\lceil \frac{n}{k} \rceil$ evolves in different scenarios.

First of all, a minimum of 3 rejuvenation groups will exist in every config-

uration with $k \geq 1$, i.e., either the replicas do not rejuvenate at all ($k = 0$), or if they rejuvenate, at least a sequence of 3 rejuvenations will occur. This follow from the fact that $n \geq 3f + 2k + 1$, and thus $\frac{n}{k} \geq 3$.

Secondly, the number of rejuvenation groups is upper-bounded by n. In the worst case, $k = 1$, and therefore $\lceil \frac{n}{k} \rceil = n$. So, we see that the number of rejuvenation groups may vary between 3 and n, depending on the value of k. In fact, it is easy to see that in order to have a certain number $l = \lceil \frac{n}{k} \rceil$ of rejuvenation groups, one has to choose $k \geq \frac{3f+1}{l-2}$. A special case of this condition is that the minimum number of rejuvenations ($l = 3$), is obtained by setting $k \geq 3f + 1$. For instance, if $f = 1$, and one wants $l = 3$ rejuvenation groups, then k should be greater or equal than 4. Notice that, setting $k \geq 3f + 1$ impacts the total number of replicas in an interesting way: $n \geq 3(3f + 1)$. This means that in order to ensure availability with a minimum number of rejuvenation groups, one needs three times more replicas than in the scenario where no availability is guaranteed. Table 5.5 summarizes the minimum number of replicas needed when different strategies are followed.

Goal	**#rejuv. groups (l)**	k	f	**#replicas (n)**
no availability guarantees	–	0	1 2 3	4 7 10
availability & min. num. of replicas	n	1	1 2 3	6 9 12
availability & min. num. of rejuv. groups	3	$\geq 3f + 1$	1 2 3	12 21 30

Table 5.5: Examples of strategies that may guide the choice of the values of n, f, and k.

Figure 5.6 depicts the evolution of the number of rejuvenation groups resulting from combinations of k and f values. As expected, the number l of reju-

venation groups is lower for higher values of k, and $l = 3$ for every $k \geq 3f + 1$.

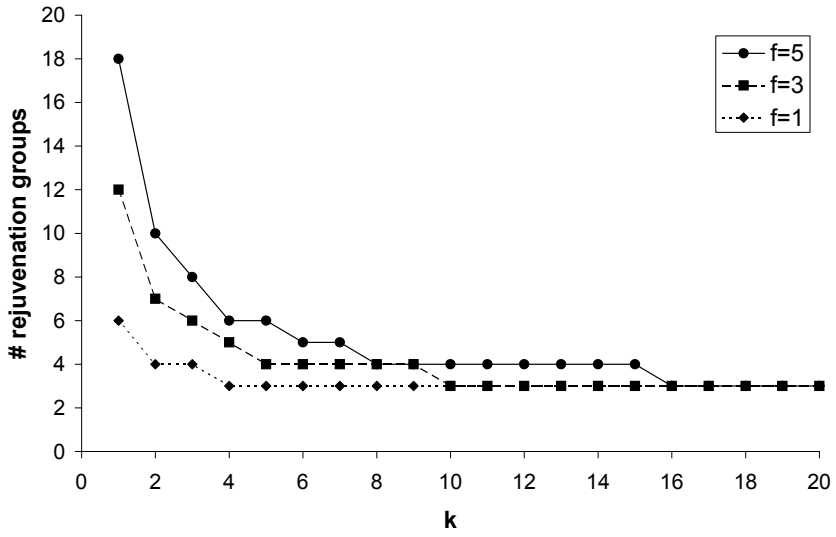

Figure 5.6: Number of rejuvenation groups (l) required per k and f.

Figure 5.7 shows the cost, in terms of the minimum number of replicas required, to guarantee $n \geq 3f + 2k + 1$. We see that n linearly increases with k, at a rate that is independent of the specific value of f.

Figures 5.6 and 5.7 can help the system architect to calculate the cost of choosing a specific f and k, in terms of the total number of replicas and the number of rejuvenation groups. A different perspective is presented in Figure 5.8. In this figure, one can analyze what is the minimum number of rejuvenation groups allowed by a given n and f. It can help the system architect to decide if it pays to tolerate a higher k in order to decrease the number of rejuvenation groups. For instance, with $n = 13$ and $f = 3$, a minimum of 13 rejuvenation groups is required (because $k = 1$), but with a single one more replica ($n = 14$), only 7 rejuvenations in sequence are needed, because k can now be set to 2. If one continues to follow the line representing $f = 3$, the conclusion is that adding still one more replica ($n = 15$) would rather worsen recoveries than improving it, given that both the number of replicas and reju-

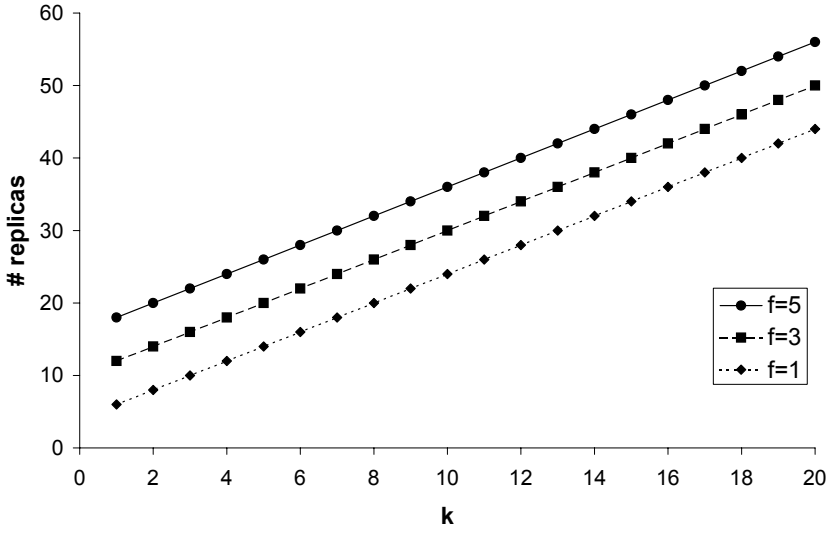

Figure 5.7: Number of replicas required per k and f.

venation groups increase.

The general conclusion is that, if one starts from a configuration with $k = 1$, the higher gain in terms of the number of rejuvenation groups is achieved by adding the necessary number of replicas such that k can be set to 2. By doing this, the number of rejuvenation groups is reduced from n to $\lceil \frac{n}{2} \rceil$. In practice, if the system is deployed with $n = 3f + 2k + 1 = 3f + 3$ replicas because $k = 1$, then one should add two more replicas to allow $k = 2$. These two extra replicas decrease substantially the minimum number of rejuvenation groups. This decrease is proportional to the value of f. For instance, with $f = 1$, the decrease is from 6 to 4 rejuvenation groups, but with $f = 5$, the decrease is from 18 to 10 rejuvenations in sequence.

5.2.5 Evaluation

This section evaluates, through simulation, the actual resilience and availability of a generic state machine replication system enhanced with an SMW and

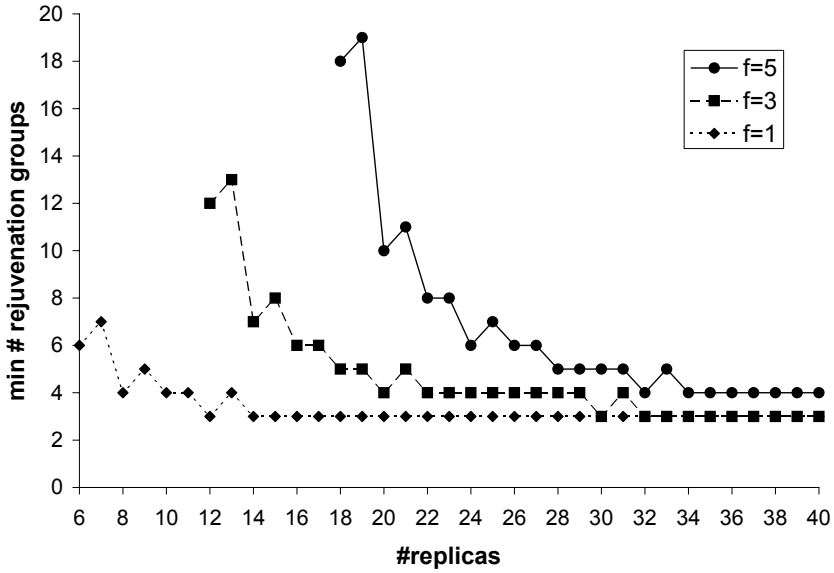

Figure 5.8: Minimum number of rejuvenation groups required per n and f.

using the proposed recovery strategies. Moreover, it is also shown that previous Byzantine-resilient state machine replication approaches may not guarantee availability under realistic settings.

First, we present the modeling formalism and the models developed to represent both a generic state machine replication system that periodically recovers, and the adversary that tries to break the system by compromising more than f replicas. Then, we present the results of simulating such an environment under different settings and we conclude that our approach is indeed effective in assuring both resilient and available operation.

5.2.5.1 SAN Models

As in Section 4.2.2, stochastic activity networks (SANs) were used as the modeling formalism.

We built atomic SAN submodels for a state machine client, a state machine replica and the typical adversary that is constantly trying to corrupt system

replicas. The complete model of the system is composed using join operations. We first present a description of each submodel, and then show how the submodels are combined to form the composed model.

In the remainder of the section, Figures 5.9, 5.10, and 5.11 present the SAN models. It is not necessary to understand them in order to follow the explanations in the text. To understand fully the graphical representation of the models and the models themselves, the interested reader can find detailed explanations of the generic SAN's formalism in Sanders & Meyer (2000), and a detailed documentation of the models is presented in Appendix B.

SAN Model for the Adversary

The SAN in Figure 5.9 models the typical adversary that is constantly trying to corrupt system replicas, and that ultimately exhausts the replicated state machine when more than the assumed number of replicas are compromised. In our model, replicas are attacked in sequence not in parallel (see Section 5.2.2). Parallel attacks are more complex to model, and for the purpose of the evaluation sequential attacks are sufficient. Moreover, in certain conditions, a parallel attack may be represented by a fast sequential one. The adversary behavior is specified through one parameter:

- `minimum inter-failure time (mift)` specifies the minimum interval between attacks. In each attack, the adversary randomly compromises one replica.

The adversary SAN distinguishes between the first and the remaining attacks in order to generate different sequences of attacks. The first attack (represented by the activity `first_attack`) occurs at a random time instant t_1 such that $t_1 \in [0, \texttt{mift}]$, and the following attacks (represented by `attack2`) occur

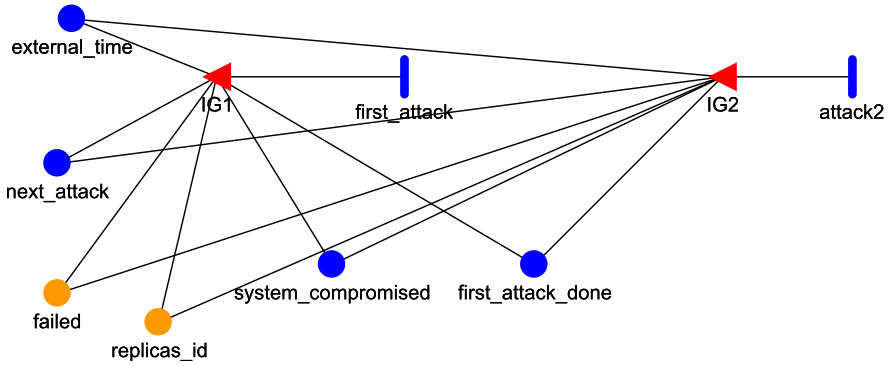

Figure 5.9: SAN model for the adversary.

at time instants $t_i, i > 1$ such that $t_i = t_{i-1} + \texttt{mift}$. The alternative would be to always deploy the first attack at the first time instant of the system execution time, but this fixed behavior would not allow to make an exhaustive evaluation of the resilience and availability of the attacked system under different attack patterns. The most relevant variables of the adversary SAN are described next:

- `external_time` stores the current execution time instant of the simulation.
- `next_attack` stores the identifier of the next replica that should be attacked. This identifier is calculated randomly after each attack.
- `failed` stores an array of boolean values, one per replica, indicating which replicas are failed. The value of each position of the array is shared with the corresponding replica. When the adversary attacks a replica, the corresponding position in the array is updated. Each replica constantly checks the value of its position in order to know when it is failed.
- `replicas_id` stores an array of integer values, one per replica, indicating which replicas are active in the simulation. The composed model defines

a total of 9 replicas, but in some simulations only some of them are used. The adversary needs to know which replicas are active in order to attack just these ones.

- `system_compromised` stores a boolean value indicating if the system is compromised, i.e., if more than f replicas are failed at the same time. This variable is only updated by the adversary (when $f + 1$ replicas are simultaneously failed), but the remaining SAN models also use it in order to know when a simulation can be stopped.

SAN Model for a State Machine Replica

The SAN in Figure 5.10 models a state machine replica. The replica periodically receives a request from a client, orders the request, and executes it. The ordering and the execution are modelled by a normal distribution with a certain mean and variance. The goal is that different atomic broadcast protocols (ordering) and different applications (execution) can be represented by the model. Moreover, the replica proactively triggers periodic recoveries. The replica behavior is specified through seven parameters:

- `ab_mean`, `ab_variance` specify the mean and the variance values of the normal distribution modelling the request ordering.
- `exec_mean`, `exec_variance` specify the mean and the variance values of the normal distribution modelling the request execution.
- $\mathtt{T}_P$ specifies the maximum interval between two consecutive replica recoveries.
- $\mathtt{T}_D$ specifies the maximum interval between the start and termination of a replica recovery.

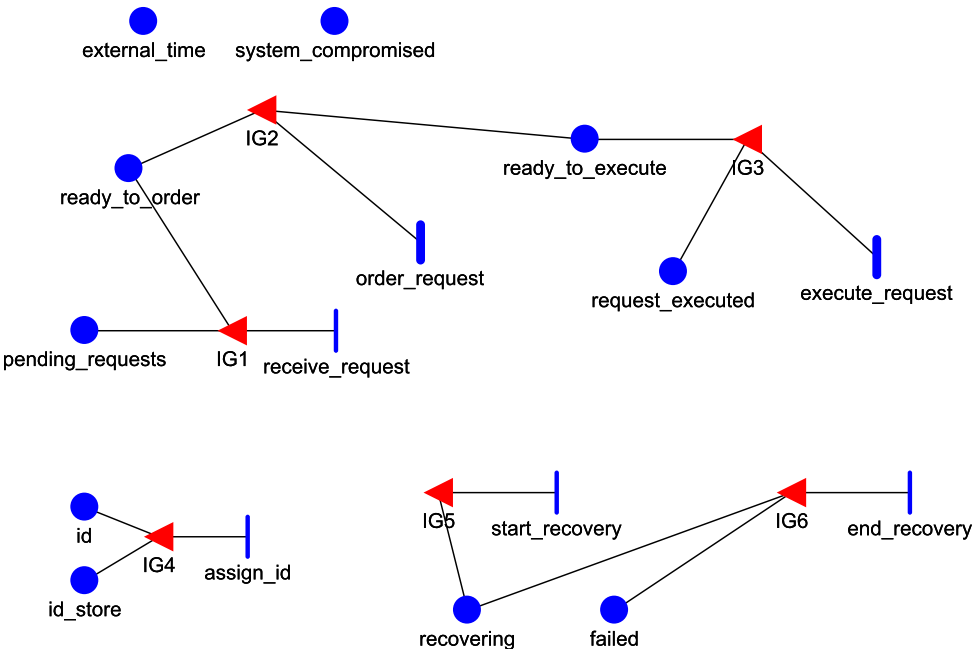

Figure 5.10: SAN model for a state machine replica.

- `k` specifies the maximum number of replicas recovered simultaneously.

The state machine replica model can be decomposed in the following three parts:

request processing This part is composed of three activities: `receive_request`, `order_request`, and `execute_request`.

The `receive_request` activity is instantaneous and it is triggered when a request is received from the state machine client.

The `order_request` activity is modelled by a normal distribution with mean `ab_mean` and variance `ab_variance` and it is triggered after a request being received by a sufficient number of replicas. This triggering condition is used to simulate the behavior of a normal atomic broadcast

protocol: if a sufficient number of replicas is not available (due to accidental or malicious actions), the protocol blocks. The exact number of replicas that is considered sufficient varies from protocol to protocol but, given that it has to ensure the intersection property revisited in Section 5.2.4.2, this number is lower-bounded by $\lceil \frac{n+f+1}{2} \rceil$. Therefore, we used this number in our experiments.

The `execute_request` activity is modelled by a normal distribution with mean `exec_mean` and variance `exec_variance` and it is triggered after a request being ordered.

recovery This part is composed of two activities: `start_recovery` and `end_recovery`.

The `start_recovery` activity is instantaneous and it is triggered in the time slots allocated for the replica recovery. Each replica recovers exactly once in each T_P time interval, but given that at most k recover at the same time, the interval T_P is split into $\lceil \frac{n}{k} \rceil$ slots with duration T_D, and each group of k replicas recover in each one of these slots (see Section 5.2.4.3). During a recovery, the replica behaves as being failed, and thus requests are neither received, ordered, or executed.

The `end_recovery` activity is instantaneous and it is triggered T_D time units after the triggering of the `start_recovery` activity. The `end_recovery` activity resets the value of the variable `failed`, thus recovering the replica if it was failed before.

id assignment This part is composed of a single activity: `assign_id`. This activity is instantaneous and it is triggered in the beginning of a simulation. The goal is to dynamically assign identifiers to the quantity of replicas used in each simulation. Although 9 replicas are always present, only n

can have a valid identifier and effectively participate in the simulation. Therefore, the first n replicas to trigger the `assign_id` activity obtain a valid identifier, whereas the remaining ones do not obtain any identifier and therefore are inactive during the simulation.

The most relevant variables of the state machine replica SAN are described next:

- `external_time` stores the current execution time instant of the simulation.
- `pending_requests` stores the number of requests sent by the state machine client that were not yet received (by the `receive_request` activity).
- `failed` stores a boolean value indicating if the replica is failed. This variable can be updated by the adversary as a result of an attack, or by the `end_recovery` activity as result of a recovery.
- `system_compromised` stores a boolean value indicating if the system is compromised, i.e., if more than f replicas are failed at the same time.

SAN Model for a State Machine Client

The SAN in Figure 5.11 models a state machine client. The state machine client periodically sends a request to the state machine replicas and waits replies. A new request is only sent after the reception of $2f + 1$ replies to the previous request. We choose to model a client in this way in order to avoid concurrent requests, and thus reduce the complexity of the various SANs. However, we are still able to measure the impact of concurrency on the ordering and execution operations of the replicated state machine. This is possible because

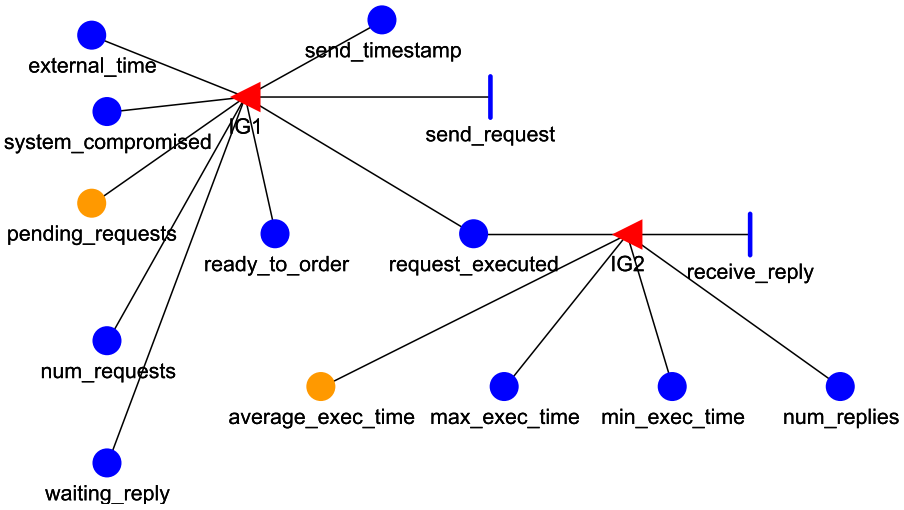

Figure 5.11: SAN model for a state machine client.

these operations are modelled using normal distributions, and one may use mean and variance values obtained from experiments with real request loads. In addition, the state machine client SAN maintains statistics about the execution time of a request and there is a counter of the number of replies received.

The state machine client model is composed of two activities: `send_request` and `receive_reply`.

The `send_request` activity is instantaneous and it is triggered if $2f + 1$ replies were received in response to the previous request. This activity increments the number of pending requests of every replica.

The `receive_reply` activity is instantaneous and it is triggered if $2f + 1$ replies are received. This activity is responsible by updating the execution time statistics.

The most relevant variables of the state machine client SAN are described next:

- `external_time` stores the current execution time instant of the simulation.

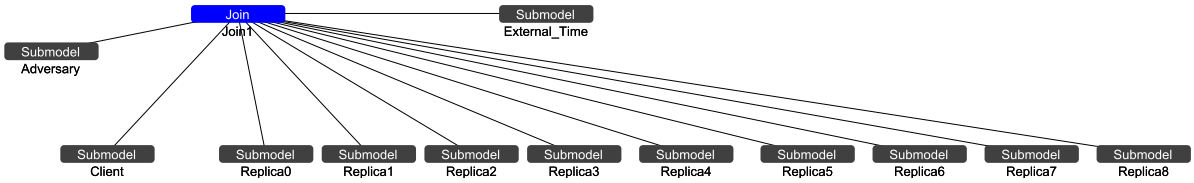

Figure 5.12: SAN model for the composed model.

- `pending_requests` stores an array of integer values, one per replica, indicating how many requests are pending for each replica. The value of each position of the array is shared with the corresponding replica. Given that there are no concurrent requests, the number of pending requests is never greater than one.

- `system_compromised` stores a boolean value indicating if the system is compromised, i.e., if more than f replicas are failed at the same time.

Composed Model

The composed model for the simulation environment is presented in Figure 5.12 and consists of the three atomic SAN submodels presented before, organized in the following way: one replica SAN per state machine replica, one client SAN, and one adversary SAN. The composed model also includes an external time submodel, which simply simulates an external time reference. Note that the composed model includes only 9 replicas. More could have been added, but this number was sufficient for the experiments that are described in Section 5.2.5.2.

The overall model behavior is specified by the parameters defined for the three submodels, plus the following: n specifies the total number of state ma-

chine replicas, and f represents the assumed maximum number of failed replicas.

5.2.5.2 Simulation Results

We used the Möbius (Deavours *et al.*, 2002) tool to build the SANs, and simulate the model. The goal of the simulations was to show the importance of taking into account recoveries, and the unavailability that may be provoked by them, when choosing the total number of replicas of a state machine replicated system.

We used a single metric in the simulations: *request average execution time*. This metric measures the average execution time between a request being sent and $2f + 1$ replies being received. Notice that in the worst case a state machine client needs $2f + 1$ replies: although $f + 1$ identical replies guarantee that the response is correct, at most f replies may be sent by compromised replicas. Therefore, for simulation purposes, we consider the worst-case scenario.

In the next subsections we first analyze the impact of different adversary powers on the execution time when using $3f + 1$ replicas and $3f + 2k + 1$ replicas; and then it is evaluated the impact of increasing recovery durations on the execution time when using the same two types of replica quorums. The results confirm our initial expectations in that $3f + 1$ replicas are not sufficient to guarantee availability.

Tables 5.6 and 5.7 present the actual parameter values used to perform the simulations. The time unit is seconds. The maximum execution time is set to 10.000 seconds ($\sim$ 3 hours) just for simulation purposes. Real systems execute during much longer intervals of time, but we found that 3 hours is a good tradeoff, because it allows reasonable simulation processing times, and it represents a significant window of time to derive conclusions on the expected

system resilience and availability. The majority of the remaining parameter values are based on the reported performance results of BFS (Castro & Liskov, 2002), a Byzantine fault-tolerant state machine replication system. Although most of these values are not relevant in terms of the goal of the simulations, we choose to extract them from a real system, in order to extrapolate how such a system would behave in the conditions evaluated in the simulations. Some specific parameter values are explained next. `ab_mean`, `ab_variance`, `exec_mean`, `exec_variance`, and T_D (in the analysis of the impact of the adversary power), were extracted from the benchmark results presented in Castro & Liskov (2002). The value of T_P was chosen in order to guarantee that recoveries do not overlap, as explained in Section 5.2.4.3. Therefore, T_P is always greater than $\lceil \frac{n}{k} \rceil T_D$ in every simulation configuration.

Parameter	**Impact of the Adversary Power**	
	Figure 5.13	Figure 5.14
n	4 vs 6	7 vs 9
f	1	2
k	1	1
met	10000	10000
T_P	260	400
T_D	43	43
ab_mean	0.0015	0.0015
ab_variance	0	0
exec_mean	0.04	0.04
exec_variance	0	0
mift	**∞, 3600, 2700, 1800, 900, 600, 300, 240, 180, 120, 60**	

Table 5.6: Parameter values used in the simulations regarding the impact of the adversary power.

Parameter	Impact of the Recovery Duration	
	Figure 5.15	Figure 5.16
n	4 vs 6	7 vs 9
f	1	2
k	1	1
met	10000	10000
$\mathbf{T}_P$	1000	1000
$\mathbf{T}_D$	**10, 30, 60, 90, 120, 150**	
ab_mean	0.0015	0.0015
ab_variance	0	0
exec_mean	0.04	0.04
exec_variance	0	0
mift	1000	500

Table 5.7: Parameter values used in the simulations regarding the impact of the recovery duration.

Impact of the Adversary Power

We performed two different sets of simulations in order to assess the impact of adversaries of increasing power on the same state machine replicated system, deployed with different replica quorums.

In the first set of simulations, f is set to 1 and thus the replicated state machine is able to resist up to one intrusion between recoveries. When no attack occurs, the average execution time of the system is very low, in the order of 50 milliseconds. Figure 5.13 compares the average execution time of systems using $3f + 1 = 4$ replicas and $3f + 2k + 1 = 6$ replicas. The main conclusion is that the average execution time remains constant (close to 50 milliseconds) when 6 replicas are used, whereas a system with only 4 replicas is highly sensitive to the interval between attacks. One can observe that when the adversary power is near the maximum value assumed (i.e., when the adversary is almost as fast as recoveries), the average execution time of a system using $3f + 1$ replicas is greater than 3 seconds.

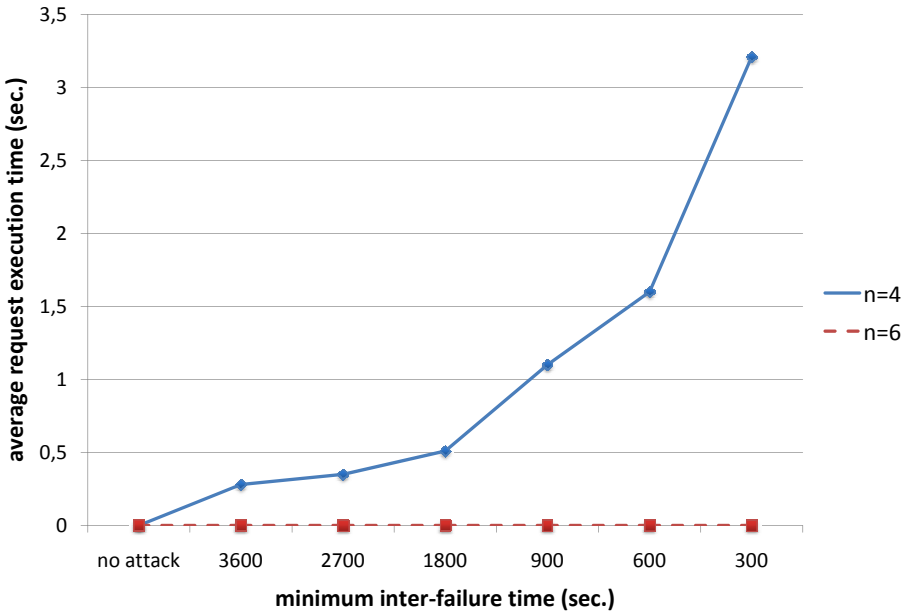

Figure 5.13: Impact of the adversary power when $f = 1$.

The second set of simulations makes the same type of comparison, but now f is set to 2 and thus the replicated state machine is able to resist up to two intrusions between recoveries. When no attack occurs, the average execution time is still in the order of 50 milliseconds. Figure 5.14 compares the average execution time of systems using $3f + 1 = 7$ replicas and $3f + 2k + 1 = 9$ replicas. As before, the conclusion is that the average execution time remains constant when $3f + 2k + 1$ replicas are used, whereas a system with only $3f + 1$ replicas is sensitive to the interval between attacks. However, one can observe that a system with $3f + 1$ replicas is less sensitive to the adversary power when $f = 2$ than when $f = 1$. This happens because when $f = 2$, the system is able to tolerate two arbitrary faults between recoveries. But this also means that the system is able to tolerate one intrusion and one recovering replica, without any impact on availability. Therefore, availability is only affected when the interval between attacks is sufficient to compromise two replicas.

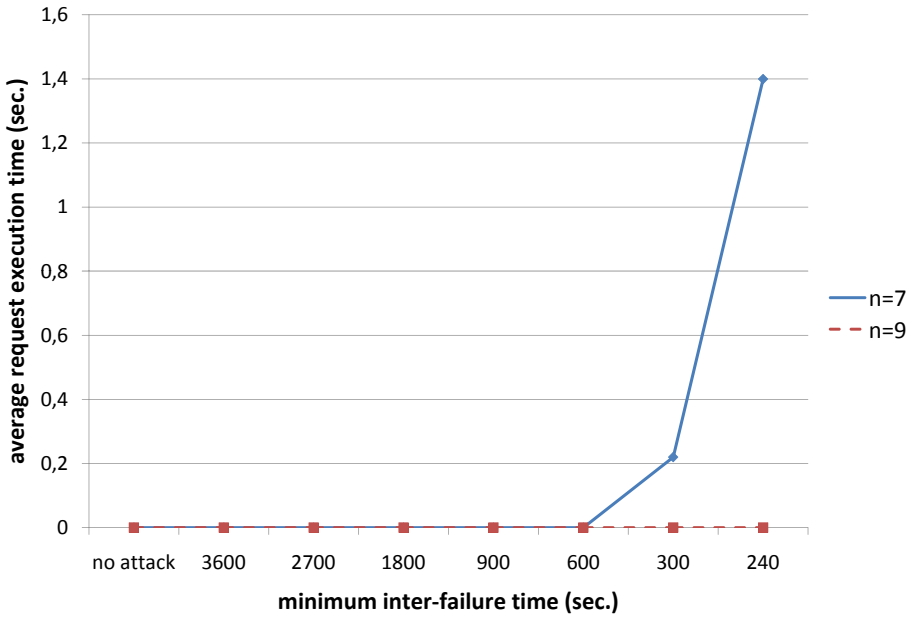

Figure 5.14: Impact of the adversary power when $f = 2$.

Impact of the Recovery Duration

As in the previous section, we performed two different sets of simulations, but now the goal was to evaluate the impact of having different recovery execution times on the same state machine replicated system deployed with different replica quorums. Moreover, in order to observe the worst-case behavior of the system, the minimum inter-failure time (mift) is set to the lowest possible "safe" value, i.e., sufficient to provoke exactly f replica failures between recoveries.

In the first set of simulations, f is set to 1. Figure 5.15 compares the average execution time of systems using $3f + 1 = 4$ replicas and $3f + 2k + 1 = 6$ replicas. The conclusion is that the average execution time remains constant when 6 replicas are used, whereas a system with only 4 replicas is highly sensitive to the duration of a replica recovery: with 10 seconds recoveries the average

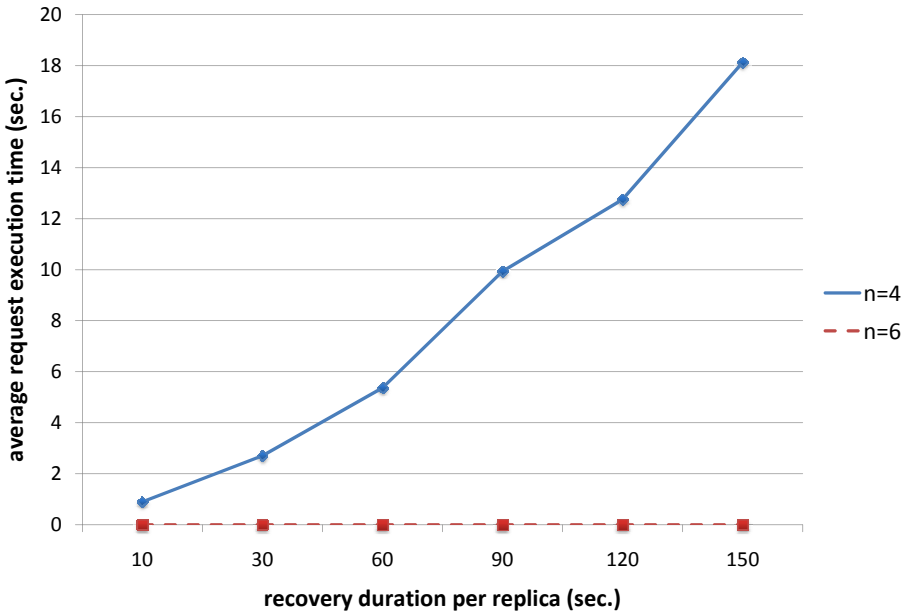

Figure 5.15: Impact of the recovery duration when $f = 1$.

request execution time is around 1.5 seconds, whereas if the recovery duration increases to 150 seconds, the resulting average request execution time is around 18 seconds!

In the second set of simulations, f is set to 2. Figure 5.16 compares the average execution time of systems using $3f + 1 = 7$ replicas and $3f + 2k + 1 = 9$ replicas. Given that this set of simulations uses a higher number of replicas (nine at maximum), and that the recovery period $T_P = 1000$, the set of recovery durations considered in this graph do not include some that appear in the graph of Figure 5.15. Otherwise, the recovery of all replicas could take more time than the recovery period. The simulations results show that the average execution time remains constant when $3f + 2k + 1$ replicas are used, whereas a system with only $3f + 1$ replicas is sensitive to the duration of a replica recovery.

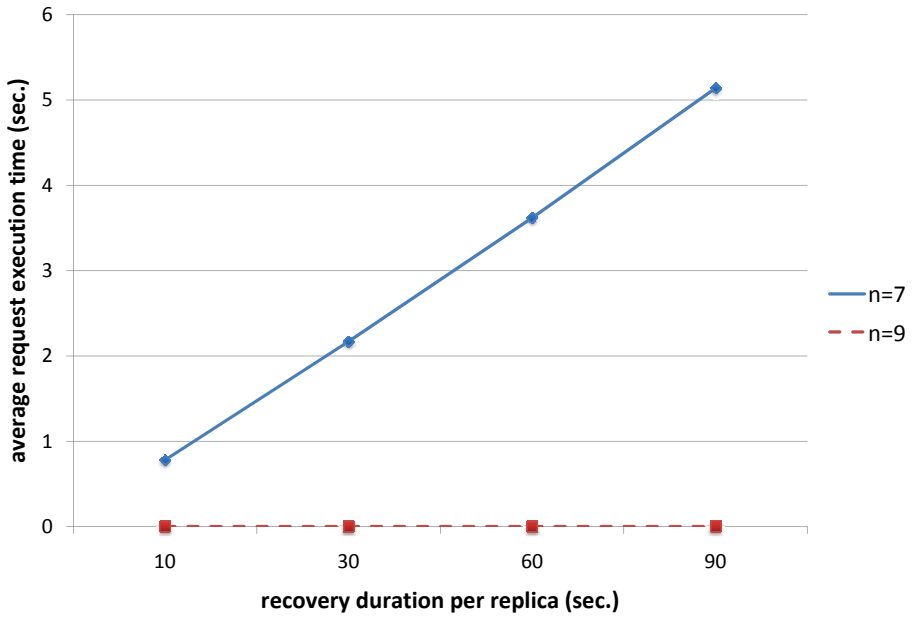

Figure 5.16: Impact of the recovery duration when $f = 2$.

5.2.5.3 Summary

The different types of simulations that were performed show that a state machine replicated system needs at least $3f + 2k + 1$ replicas in order to guarantee availability. Classical approaches, such as BFS, using only $3f + 1$ replicas, may naturally present very high unavailability times, specially in two relevant scenarios: in the presence of strong adversaries; or when recoveries have long execution times. We showed that in practical terms the addition of two more replicas may considerably improve the situation.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis makes two distinct contributions. The first part of the thesis (Chapters 2 and 3) was devoted to a discussion about the actual resilience of current intrusion-tolerant synchronous and asynchronous systems. We proposed a model that takes into account the evolution of a specified resource along the timeline of system execution. We offered a predicate that allows to reason formally about the possibility or impossibility of achieving *exhaustion-safety*, i.e., safety against resource exhaustion. On the possibility side, we showed that it is feasible to build a node-exhaustion-safe intrusion-tolerant synchronous system, as long as it has a bounded lifetime, and timing assumptions are never violated. However, and against the current belief, we also showed that it is impossible to build a node-exhaustion-safe intrusion-tolerant system under the classical asynchronous model, even using proactive recovery.

The *theoretical impact* of these findings remains across the fault spectrum. That is, such failure syndromes (by exhaustion) were previously unknown and even with accidental faults they can cause the inadvertent failure of asynchronous or synchronous distributed systems (without bounded lifetime). These systems are to our findings fairly the same as "apparently working" pre-FLP

asynchronous consensus systems were to FLP. In consequence, our results may alert researchers and help conceive better distributed systems. The *practical impact* of the same findings can in our opinion become higher, commensurate to the measure in which systems, critical or generic, are becoming prey to hacker attacks (malicious faults). This means that, with increasing probability, systems having the failure syndrome discovered in this thesis not only can but *will* be attacked and made to fail.

In the second part of the thesis (Chapters 4 and 5), we did a deeper investigation of the impossibility of building a node-exhaustion-safe intrusion-tolerant proactively recovered system under the asynchronous model, concluding that proactive recovery mechanisms typically require stronger environment assumptions (e.g., synchrony, security) than the rest of the system, which can remain asynchronous. Based on this, we proposed *proactive resilience* as a novel approach to proactive recovery that is based on a hybrid distributed system model and architecture: the proactive recovery mechanisms execute in a subsystem with "better" properties than the rest of the system.

The Proactive Resilience Model (PRM) was presented and it was shown that it can be used to build node-exhaustion-safe systems. This model was applied to the secret sharing and the state machine replication scenarios, in order to derive corresponding node-exhaustion-safe versions of these systems.

Regarding the secret sharing application scenario, we presented some experimental results that confirm our theoretical postulates. Our experimental secret sharing prototype is intrusion-tolerant and can be configured to tolerate any number of intrusions as long as the intrusion rate is not greater than one intrusion per second.

We also performed a quantitative assessment of the level of redundancy required to achieve resilient and available state machine replication, i.e., simulta-

neously securing node-exhaustion-safety and availability: we established the new result that a minimum $3f + 2k + 1$ replicas are required for tolerating f Byzantine faults, with at most k replicas recovering simultaneously.

6.2 Future Work

We plan to implement an experimental prototype of an intrusion-tolerant proactive resilient state machine replication system, in order to confirm the simulation results described in Section 5.2.5.2. This prototype can be used to enhance the resilience and availability of any existing deterministic service.

Moreover, we intend to research other application scenarios of proactive resilience in the context of the CRUTIAL[1] EU-IST project. The goal is to enhance the resilience and availability of critical infrastructures like the power, water and gas distribution networks, which have a fundamental role in modern life.

As described in the thesis, proactive recovery is crucial if one wants to build intrusion-tolerant distributed systems that are simultaneously node-exhaustion-safe. Reactive recovery can be seen as a complementary approach to proactive recovery, in the sense that it may trigger recoveries sooner when malicious behavior is detected. These early recoveries may have benefits not only in terms of performance, but also in terms of system safety. Proactive recovery guarantees node-exhaustion-safety as long as recoveries are faster than fault production, i.e., if recoveries take less time than a lower-bound on the time needed to produce $f + 1$ node failures. This lower-bound is calculated at design time and must have a very high coverage (Powell, 1992). However, during system execution, malicious adversaries may prove to be more fierce than expected and may have the ability to compromise $f + 1$ nodes within the interval between two consecutive recoveries. Proactive recovery alone would

[1] CRitical UTility InfrastructurAL Resilience: http://crutial.cesiricerca.it/

not be sufficient to maintain node-exhaustion-safety in this scenario (because design time assumptions were violated), but reactive recovery has the ability to defend the system against such fierce attacks if it is possible to detect the malicious behavior of some nodes before $f + 1$ being compromised. We are currently studying ways of combining proactive and reactive recovery.

One of the fundamental assumptions of intrusion-tolerant distributed systems is that nodes are different - or diverse - in order to have a different set of vulnerabilities. Otherwise, an attack that is effective against one node is effective against all of them, and a coordinated attack could compromise all the nodes almost at the same time. Many different diverse techniques have been proposed in the past targeting accidental and/or malicious faults (Chen & Avizienis, 1978; Joseph & Avizienis, 1988; Littlewood & Strigini, 2004; Pucella & Schneider, 2006; Randell, 1975). Moreover, Obelheiro *et al.* (2006) identify several possible axes of diversity, i.e., several components of a system that may admit different instances: application software, administrative domain, physical location, operating system, and hardware. Proactive resilience should be combined with diversity techniques in order to increase its effectiveness. For instance, each recovery may randomize certain parts of the system in order that vulnerabilities are somehow changed or removed and the adversary cannot make use of knowledge learnt before the recovery. This is another possible research line to follow as future work.

Appendix A

Proactive Resilience Evaluation - Detailed SAN Models

A.1 Model: External/Internal Timebases

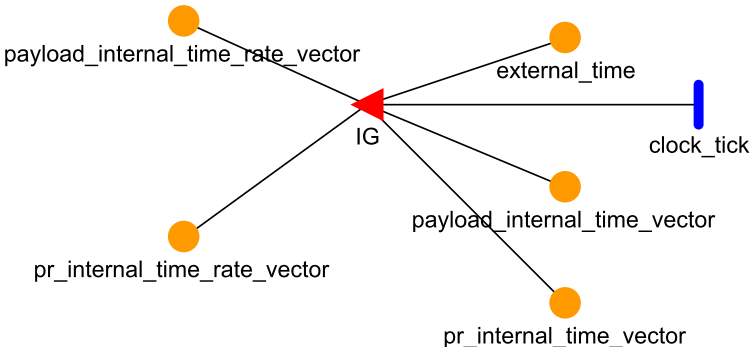

Figure A.1: SAN model for the external/internal timebases.

A.1.1 Places

Place Names	Initial Markings
external_time	0
payload_internal_time_rate_vector	1

payload_internal_time_vector	0
pr_internal_time_rate_vector	1
pr_internal_time_vector	0

A.1.2 Activities

Timed Activity:	**clock_tick**
Deterministic Distribution	**Period**: 1
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	**IG**
Predicate	1

Function	external_time->Mark()++;
	for (int i=0; i<n; i++){
	payload_internal_time_vector->Index(i)->Mark()+=
	1.0/payload_internal_time_rate_vector->Index(i)->Mark();
	pr_internal_time_vector->Index(i)->Mark()+=
	1.0/(pr_internal_time_rate_vector->Index(i)->Mark());
	}

A.2 Model: Stealth Time Adversary

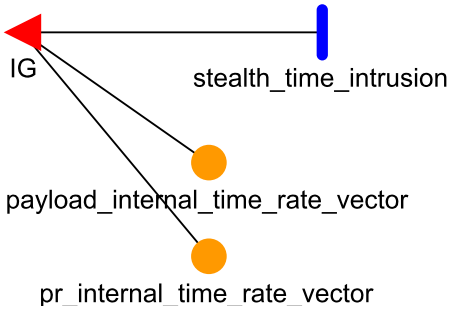

Figure A.2: SAN model for the stealth time adversary.

A.2.1 Places

Place Names	Initial Markings
payload_internal_time_rate_vector	1
pr_internal_time_rate_vector	1

A.2.2 Activities

Timed Activity:	**stealth_time_intrusion**
Deterministic Distribution	**Period**: time_attack_period
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	**IG**
Predicate	(payload_time_attack_factor > 1) || (pr_time_attack_factor > 1)

Function	int already_attacked, count, current_victim;
	if (payload_time_attack_factor > 1) {
	/* randomly time attack 1 node not already attacked */
	count = 0;
	already_attacked=1;
	while (already_attacked && count<n) {
	current_victim = rand()%n;
	if (payload_internal_time_rate_vector->
	Index(current_victim)->Mark()
	!= payload_time_attack_factor) {
	payload_internal_time_rate_vector->
	Index(current_victim)->Mark()
	= payload_time_attack_factor;
	already_attacked = 0;
	count++;
	}
	}
	}
	/* delays both pr and payload */
	if (pr_time_attack_factor > 1) {
	count=0;
	already_attacked=1;
	while (already_attacked && count<n) {
	current_victim = rand()%n;
	if (pr_internal_time_rate_vector->
	Index(current_victim)->Mark()
	!= pr_time_attack_factor) {
	pr_internal_time_rate_vector->
	Index(current_victim)->Mark()
	= pr_time_attack_factor;
	payload_internal_time_rate_vector->
	Index(current_victim)->Mark()
	= pr_time_attack_factor;
	already_attacked = 0;
	count++;
	}
	}
	}

A.3 Model: Conspicuous Time Adversary

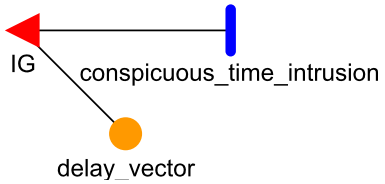

Figure A.3: SAN model for the conspicuous time adversary.

A.3.1 Places

Place Names	Initial Markings
delay_vector	0

A.3.2 Activities

Timed Activity:	conspicuous_time_intrusion
Deterministic Distribution	**Period**: time_conspicuous_attack_period
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	**IG**
Predicate	(time_conspicuous_attack_delay > 0)

Function	int already_attacked, count, current_victim;
	/* delays both pr and payload */
	count=0;
	already_attacked=1;
	while (already_attacked && count<n) {
	current_victim = rand()%n;
	if (delay_vector->Index(current_victim)->Mark()
	!= time_conspicuous_attack_delay){
	delay_vector->Index(current_victim)->Mark()
	= time_conspicuous_attack_delay;
	already_attacked = 0;
	}
	count++;
	}

A.4 Model: Classic Adversary

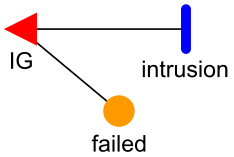

Figure A.4: SAN model for the classic adversary.

A.4.1 Places

Place Names	Initial Markings
failed	0

A.4.2 Activities

Timed Activity:	intrusion
Deterministic Distribution	**Period**: mift
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	**IG**
Predicate	1

Function	int i, count, already_failed, count_failed,
	count_recovering, current_victim;
	/* counts the number of compromised nodes */
	count = 0;
	for (i=0; i<n; i++){
	if (failed->Index(i)->Mark())
	count++;
	}
	/* only does something if at least one node is still ok */
	if (count<n){
	/* randomly compromises 1 node not already failed */
	already_failed=1;
	while (already_failed) {
	current_victim = rand()%n;
	if (failed->Index(current_victim)->Mark()==0){
	failed->Index(current_victim)->Mark()=1;
	already_failed = 0;
	}
	}
	}

A.5 Model: Node

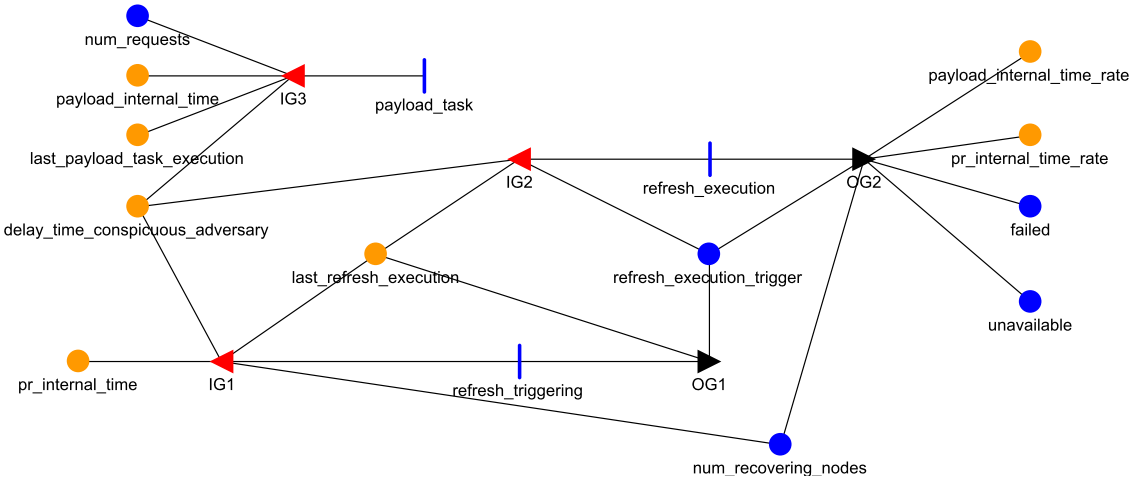

Figure A.5: SAN model for a node.

A.5.1 Places

Place Names	Initial Markings
delay_time_conspicuous_adversary	0
failed	0
last_payload_task_execution	0
last_refresh_execution	0
num_recovering_nodes	0
num_requests	0
payload_internal_time	0
payload_internal_time_rate	1
pr_internal_time	0
pr_internal_time_rate	1
refresh_execution_trigger	0

unavailable	0

A.5.2 Activities

Instantaneous Activities Without Cases:
payload_task
refresh_execution
refresh_triggering

Input Gate:	**IG1**
Predicate	pr_internal_time->Mark()-last_refresh_execution->Mark()>= (Tp-Td+delay_time_conspicuous_adversary->Mark()) && num_recovering_nodes->Mark() < mrd
Function	;

Input Gate:	**IG2**
Predicate	mrd>0 && refresh_execution_trigger->Mark()==1 && pr_internal_time->Mark()-last_refresh_execution->Mark() >= (Td/((n*1.0)/mrd))+ delay_time_conspicuous_adversary->Mark()
Function	;

Input Gate:	IG3
Predicate	failed->Mark()==0 && unavailable->Mark()==0 && payload_internal_time->Mark()- last_payload_task_execution->Mark()>= (payload_task_period+ delay_time_conspicuous_adversary->Mark())

Function	last_payload_task_execution->Mark() =
	payload_internal_time->Mark();
	num_requests->Mark()++;

Output Gate:	OG1

Function	last_refresh_execution->Mark() =
	pr_internal_time->Mark();
	refresh_execution_trigger->Mark()=1;
	num_recovering_nodes->Mark()++;
	if (mrd_exhausts==1){
	//the node is failed (and not available) during recovery
	failed->Mark()=1;
	unavailable->Mark()=1;
	}
	else {
	//the node is only not available during recovery
	unavailable->Mark()=1;
	}

Output Gate:	OG2

Function	refresh_execution_trigger->Mark()=0;
	failed->Mark()=0;
	unavailable->Mark()=0;
	if (internal_time_rejuvenation){
	pr_internal_time_rate->Mark()=1;
	delay_time_conspicuous_adversary->Mark()=0;
	}
	num_recovering_nodes->Mark()- -;

A.6 Model: Monitor

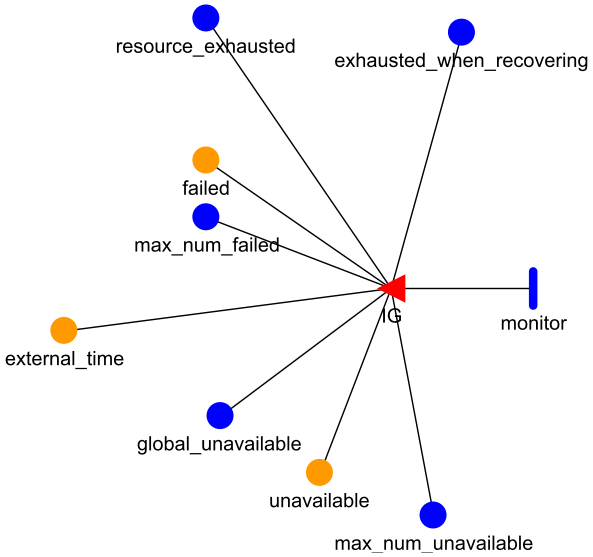

Figure A.6: SAN model for the monitor.

A.6.1 Places

Place Names	Initial Markings
exhausted_when_recovering	0
external_time	0
failed	0
global_unavailable	0
max_num_failed	0
max_num_unavailable	0
resource_exhausted	0
unavailable	0

A.6.2 Activities

Timed Activity:	**monitor**
Deterministic Distribution	**Period**: 1
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	**IG**
Predicate	1

Function	int count_failed, count_recover, i;
	/* checks if more than f nodes are failed */
	count_failed = 0;
	for (i=0; i<n; i++){
	  if (failed->Index(i)->Mark())
	  count_failed++;
	}
	/* max_num_failed is used only for statistics purposes */
	if (count_failed>max_num_failed->Mark())
	  max_num_failed->Mark()=count_failed;
	/* system is globally failed if > f nodes are failed */
	if (count_failed>f)
	  resource_exhausted->Mark()=1;
	else
	  resource_exhausted->Mark()=0;
	/* counts the number of recovering nodes */
	count_recover = 0;
	for (i=0; i<n; i++){
	  if (unavailable->Index(i)->Mark())
	  count_recover++;
	}
	if (count_recover>max_num_unavailable->Mark())
	  max_num_unavailable->Mark()=count_recover;
	if ((count_failed<=f) && (count_failed+count_recover>f))
	  exhausted_when_recover->Mark()=1;
	else
	  exhausted_when_recover->Mark()=0;

Appendix B

State Machine Replication Evaluation - Detailed SAN Models

B.1 Model: External Time

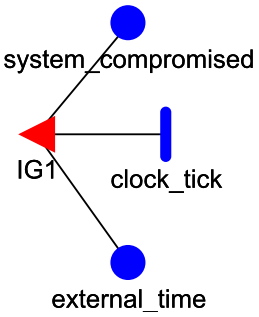

Figure B.1: SAN model for the external time.

B.1.1 Places

Place Names	Initial Markings
external_time	0
system_compromised	0

B.1.2 Activities

Timed Activity:	**clock_tick**
Deterministic Distribution	**Period**: 1
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	**IG1**
Predicate	/******************************** max exec time not reached AND system not compromised ********************************/ external_time->Mark()<=met && (system_compromised->Mark() == 0)
Function	external_time->Mark()++;

B.2 Model: Adversary

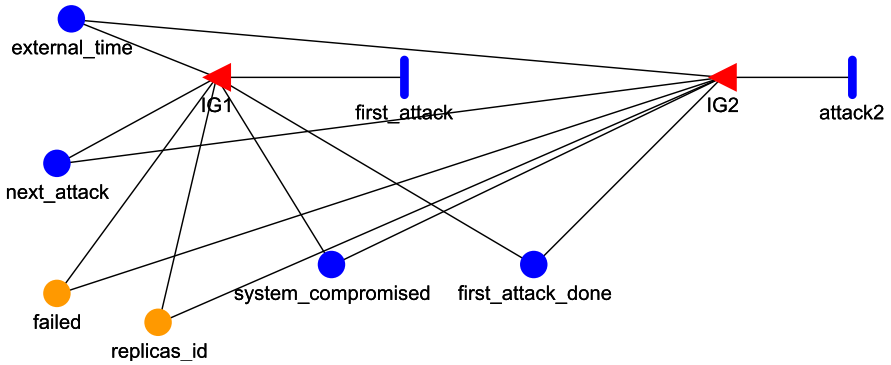

Figure B.2: SAN model for the adversary.

B.2.1 Places

Place Names	Initial Markings
external_time	0
failed	0
first_attack_done	0
next_attack	0
replicas_id	1000
system_compromised	0

B.2.2 Activities

Timed Activity:	attack2
Deterministic Distribution	**Period**: mift
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	first_attack
Uniform Distribution	**LowerBound**: 0 **UpperBound**: mift
Activation Predicate	(none)
Reactivation Predicate	(none)

Input Gate:	IG1
Predicate	/********** max exec time not reached AND system not compromised AND first attack not done AND attacks are enabled **********/ (external_time->Mark() <= met) && (system_compromised->Mark() == 0) && (first_attack_done->Mark() == 0) && (attacks_enabled == 1)

Function	first_attack_done->Mark() = 1;
	//finds the next victim (has valid id AND not failed)
	while (
	(replicas_id->Index(next_attack->Mark())->Mark() == 1000)
	|| (failed->Index(next_attack->Mark())->Mark()==1)) {
	//random attack
	next_attack->Mark() = rand()%10;
	}
	//node next_attack becomes failed
	failed->Index(next_attack->Mark())->Mark() = 1;
	//updates the next node to attack (random attack)
	next_attack->Mark() = rand()%10;
	//checks if more than f nodes are already failed
	int num_failed = 0;
	for (int i=0; i<10; i++) {
	if (failed->Index(i)->Mark() == 1)
	num_failed++;
	}
	if (num_failed>f)
	system_compromised->Mark() = 1;

Input Gate:	IG2
Predicate	/********** max exec time not reached AND system not compromised AND first attack already done ********** / (external_time->Mark() <= met) && (system_compromised->Mark() == 0) && (first_attack_done->Mark() == 1)

Function	//finds the next victim (has valid id AND not failed)
	while (
	(replicas_id->Index(next_attack->Mark())->Mark() == 1000)
	|| (failed->Index(next_attack->Mark())->Mark()==1)) {
	//random attack
	next_attack->Mark() = rand()%10;
	}
	//node next_attack becomes failed
	failed->Index(next_attack->Mark())->Mark() = 1;
	//updates the next node to attack (random attack)
	next_attack->Mark() = rand()%10;
	//checks if more than f nodes are already failed
	int num_failed = 0;
	for (int i=0; i<10; i++) {
	if (failed->Index(i)->Mark() == 1)
	num_failed++;
	}
	if (num_failed>f)
	system_compromised->Mark() = 1;

B.3 Model: Replica

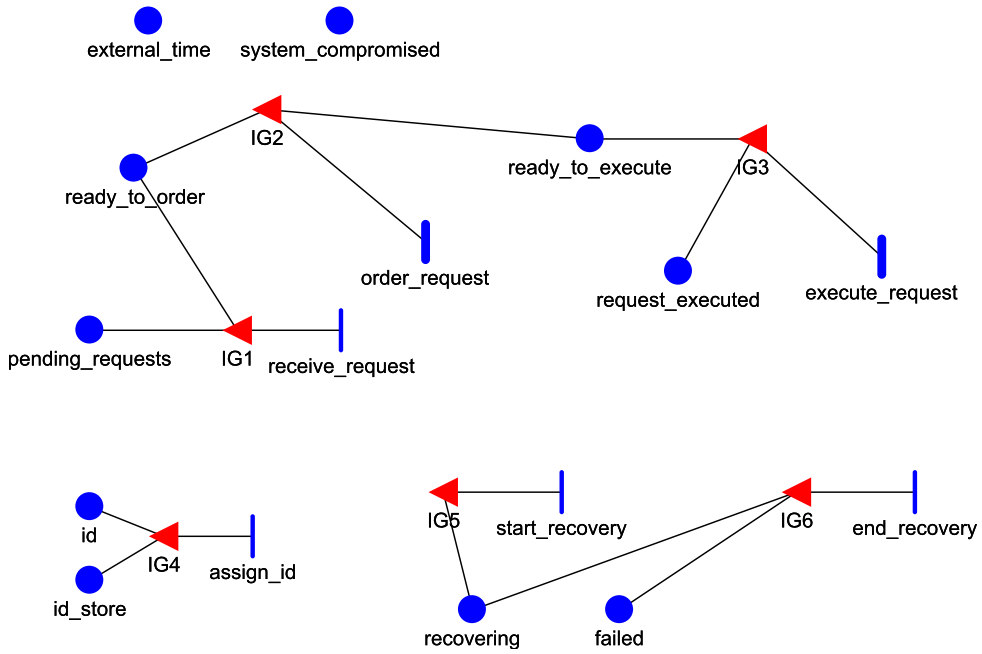

Figure B.3: SAN model for a state machine replica.

B.3.1 Places

Place Names	Initial Markings
external_time	0
failed	0
id	1000
id_store	0
pending_requests	0
ready_to_execute	0
ready_to_order	0
recovering	0

request_executed	0
system_compromised	0

B.3.2 Activities

Timed Activity:	**execute_request**
Normal Distribution	**Mean**: exec_mean **Variance**: exec_variance
Activation Predicate	(none)
Reactivation Predicate	(none)

Timed Activity:	**order_request**
Normal Distribution	**Mean**: ab_mean **Variance**: ab_variance
Activation Predicate	(none)
Reactivation Predicate	(none)

Instantaneous Activities Without Cases:
assign_id
end_recovery
receive_request
start_recovery

Input Gate:	IG1
Predicate	/***************************** max exec time not reached AND id assigned AND there is a pending request AND node not recovering AND node not failed AND system not compromised *****************************/ (external_time->Mark()<=met) && (id->Mark()!=1000) && (pending_requests->Mark() == 1) && (recovering->Mark() == 0) && (failed->Mark() == 0) && (system_compromised->Mark() == 0)
Function	ready_to_order->Mark()++; pending_requests->Mark()=0;

Input Gate:	IG2
Predicate	/*** max exec time not reached AND not yet ready to execute the request AND at least n-f or n-f-k nodes are available to order AND node not recovering AND node not failed AND system not compromised AND id assigned ***/ (external_time->Mark()<=met) && (ready_to_execute->Mark()==0) && (ready_to_order->Mark()> (n+f)/2) && (recovering->Mark() == 0) && (failed->Mark() == 0) && (system_compromised->Mark() == 0) && (id->Mark()!=1000)
Function	//ready to execute the request ready_to_execute->Mark()=1;

Input Gate:	IG3
Predicate	/***************************** max exec time not reached AND ready to execute the request AND node not recovering AND node not failed AND system not compromised AND id assigned *****************************/ external_time->Mark()<=met && ready_to_execute->Mark()==1 && (recovering->Mark() == 0) && (failed->Mark() == 0) && (system_compromised->Mark() == 0) && (id->Mark()!=1000)

Function	ready_to_execute->Mark()=0;
	//increment the number of requests executed
	request_executed->Mark()++;

Input Gate:	IG4
Predicate	/*** If the node still does not have an id assigned AND the next id available is less than n ***/ (id->Mark()==1000) && (id_store->Mark()<n)

Function	//assing the current value of id_store and increment it
	id->Mark()=id_store->Mark();
	id_store->Mark()++;

Input Gate:	**IG5**
Predicate	/***************************** max exec time not yet reached AND not recovering AND recovery start instant is now AND system not compromised AND id assigned AND recoveries are enabled *****************************/ (external_time->Mark() <= met) && (recovering->Mark()==0) && (external_time->Mark()%Tp == id->Mark()*(recovery_duration)) && (system_compromised->Mark() == 0) && (id->Mark()!=1000) && (recoveries_enabled == 1)
Function	//node recovering recovering->Mark()=1;

Input Gate:	**IG6**
Predicate	/***************************** max exec time not yet reached AND recovering AND recovery end instant is now AND system not compromised *****************************/ (external_time->Mark() <= met) && (recovering->Mark()==1) && (external_time->Mark()%Tp == ((id->Mark()+1)*(recovery_duration)-1)) && (system_compromised->Mark() == 0)

Function	//not recovering
	recovering->Mark()=0;
	//not failed
	failed->Mark()=0;

B.4 Model: Client

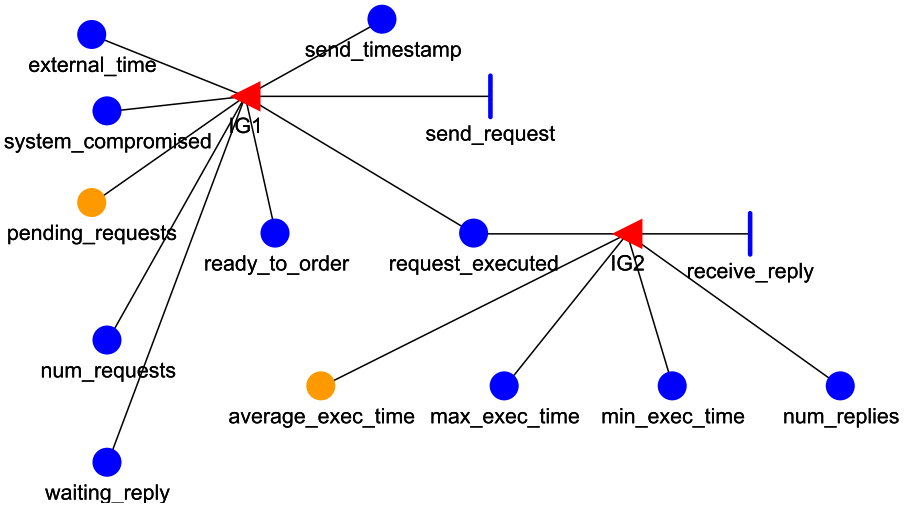

Figure B.4: SAN model for a state machine client.

B.4.1 Places

Place Names	Initial Markings
average_exec_time	0
external_time	0
max_exec_time	0
min_exec_time	0
num_replies	0
num_requests	0
pending_requests	0
ready_to_order	0
request_executed	0
send_timestamp	0

system_compromised	0
waiting_reply	0

B.4.2 Activities

Instantaneous Activities Without Cases:
receive_reply
send_request

Input Gate:	IG1
Predicate	/************************************ max exec time not reached AND no pending requests for replica 0 AND not waiting a reply AND system not compromised ************************************/ (external_time->Mark() <= met) && (waiting_reply->Mark()==0) && (external_time->Mark()-send_timestamp->Mark()>= request_interval) && (system_compromised->Mark() == 0)

Function	//init replica variables
	ready_to_order->Mark() = 0;
	request_executed->Mark() = 0;
	//waiting a reply
	waiting_reply->Mark()=1;
	//store the current timestamp in order to measure
	//request exec time
	send_timestamp->Mark() = external_time->Mark();
	//send the request to every node (max: 10)
	for (int i=0; i<10; i++)
	pending_requests->Index(i)->Mark() = 1;
	//statistics
	num_requests->Mark()++;

Input Gate:	**IG2**
Predicate	/***************************** max exec time not reached AND at least 3f+1 replies AND system not compromised *****************************/ (external_time->Mark()<=met) && (request_executed->Mark() >= 2*f+1) && (system_compromised->Mark() == 0) && (waiting_reply->Mark() == 1)

Function	//calculate the request execution time
	int exec_time;
	exec_time = external_time->Mark()-send_timestamp->Mark();
	//update statistics
	num_replies->Mark()++;
	if (num_replies->Mark()==1)
	min_exec_time->Mark() = exec_time;
	else if (exec_time < min_exec_time->Mark())
	min_exec_time->Mark() = exec_time;
	if (exec_time > max_exec_time->Mark())
	max_exec_time->Mark() = exec_time;
	average_exec_time->Mark() =
	(average_exec_time->Mark()*(num_replies->Mark()-1)+
	exec_time)/num_replies->Mark();
	//resets the number of pending requests in order to send a new request
	for (int i=0; i<10; i++)
	pending_requests->Index(i)->Mark() = 0;
	//resets the number of replies received
	request_executed->Mark()=0;
	//not waiting a reply
	waiting_reply->Mark()=0;
	//init replica variables
	ready_to_order->Mark() = 0;
	request_executed->Mark() = 0;

References

AMIR, Y., DANILOV, C., DOLEV, D., KIRSCH, J., LANE, J., NITA-ROTARU, C., OLSEN, J. & ZAGE, D. (2006). Scaling Byzantine fault-tolerant replication to wide area networks. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, 105–114, IEEE Computer Society. 93

AVIZIENIS, A., LAPRIE, J.C., RANDELL, B. & LANDWEHR, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33. 2

BARAK, B., HERZBERG, A., NAOR, D. & SHAI, E. (1999). The proactive security toolkit and applications. In *CCS '99: Proceedings of the 6th ACM Conference on Computer and Communications Security*, 18–27, ACM Press. 90

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I. & WARFIELD, A. (2003). Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 164–177, ACM Press. 42, 96

BEN-OR, M. (1983). Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, 27–30, ACM Press. 4

BHATKAR, S., DUVARNEY, D.C. & SEKAR, R. (2003). Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, 105–120. 27, 39

BHATKAR, S., SEKAR, R. & DUVARNEY, D.C. (2005). Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, 271–286. 40

BRACHA, G. & TOUEG, S. (1985). Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840. 92

CACHIN, C., KURSAWE, K. & SHOUP, V. (2000). Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, 123–132, ACM Press. 92

CACHIN, C., KURSAWE, K., LYSYANSKAYA, A. & STROBL, R. (2002). Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, 88–97, ACM Press. 4, 5, 24, 39, 55

CANETTI, R. & RABIN, T. (1993). Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, 42–51, ACM Press. 92

CASIMIRO, A., MARTINS, P. & VERÍSSIMO, P. (2000). How to build a Timely Computing Base using Real-Time Linux. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, 127–134. 85

CASTRO, M. & LISKOV, B. (2002). Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461. 4, 5, 20, 39, 55, 93, 97, 120

CHANDRA, T. & TOUEG, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267. 4, 85

CHEN, L. & AVIZIENIS, A. (1978). N-version programming: a fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing 1995, Highlights from Twenty-Five Years*, FTCS. 27, 130

CHOR, B. & DWORK, C. (1989). Randomization in Byzantine agreement. In *Advances in Computing Research 5: Randomness and Computation*, 443–497, JAI Press. 4

CLOUTIER, P., MANTEGAZZA, P., PAPACHARALAMBOUS, S., SOANES, I., HUGHES, S. & YAGHMOUR, K. (2000). DIAPM-RTAI position paper. In *Real-Time Linux Workshop*. 85

CORREIA, M., NEVES, N.F. & VERÍSSIMO, P. (2004). How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, 174–183, IEEE Computer Society. 92, 93

DEAVOURS, D.D., CLARK, G., COURTNEY, T., DALY, D., DERISAVI, S., DOYLE, J.M., SANDERS, W.H. & WEBSTER, P.G. (2002). The Möbius framework and its implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969. 66, 119

DESMEDT, Y. (1998). Some recent research aspects of threshold cryptography. In *ISW '97: Proceedings of the First International Workshop on Information Security*, 158–173, Springer-Verlag. 24

DÉFAGO, X., SCHIPER, A. & URBÁN, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421. 101

DOLEV, D., FRIEDMAN, R., KEIDAR, I. & MALKHI, D. (1996). Failure detectors in omission failure environments. Tech. Rep. TR96-1608, Cornell University, Computer Science Department. 4

DOLEV, D., FRIEDMAN, R., KEIDAR, I. & MALKHI, D. (1997). Failure detectors in omission failure environments (brief announcement). In *PODC '97: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, 286, ACM Press. 4

DOUDOU, A., GARBINATO, B., GUERRAOUI, R. & SCHIPER, A. (1999). Muteness failure detectors: Specification and implementation. In *EDCC-3: Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, 71–87. 92

FETZER, C. & CRISTIAN, F. (1997). A fail-aware datagram service. In *Proceedings of the 2nd Annual Workshop on Fault-Tolerant Parallel and Distributed Systems*. 103

FISCHER, M.J., LYNCH, N.A. & PATERSON, M.S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382. 3, 11, 12, 17, 19, 36

FORREST, S., SOMAYAJI, A. & ACKLEY, D.H. (1997). Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, 67–72. 40

GARAY, J.A., GENNARO, R., JUTLA, C. & RABIN, T. (2000). Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389. 39

GARCIA-MOLINA, H. (1982). Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59. 84

HADZILACOS, V. & TOUEG, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Tech. Rep. TR94-1425, Cornell University, Department of Computer Science. 3, 34

HERZBERG, A., JARECKI, S., KRAWCZYK, H. & YUNG, M. (1995). Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, 339–352, Springer-Verlag. 24, 39, 77, 78, 81

HERZBERG, A., JAKOBSSON, M., JARECKI, S., KRAWCZYK, H. & YUNG, M. (1997). Proactive public key and signature systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, 100–110, ACM Press. 39

JOSEPH, M.K. & AVIZIENIS, A. (1988). A fault tolerance approach to computer viruses. In *Proceedings of the IEEE Symposium on Security and Privacy*, 52–58, IEEE Computer Society. 27, 130

KENT, S. (1980). *Protecting Externally Supplied Software in Small Computers*. Ph.D. thesis, Laboratory of Computer Science, Massachusetts Institute of Technology. 42, 96

LAMPORT, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565. 92

LAMPORT, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169. 93

LAMPORT, L., SHOSTAK, R. & PEASE, M. (1982). The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401. 4

LITTLEWOOD, B. & STRIGINI, L. (2004). *Redundancy and Diversity in Security*, vol. 3193 (9th European Symposium on Research in Computer Security, Sophia Antipolis, France – ESORICS '04) of *LNCS*, 423–438. Springer. 27, 130

LYNCH, N. (1996). *Distributed Algorithms*. Morgan Kaufmann. 3, 11, 17, 19, 36

MALKHI, D. & REITER, M. (1997a). Byzantine quorum systems. In *Proceedings of the 29th ACM Symposium in Theory of Computing*, 569–578, ACM Press. 92, 104

MALKHI, D. & REITER, M. (1997b). Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, 116–124. 92

MALKHI, D. & REITER, M. (2000). An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202. 92

MARSH, M.A. & SCHNEIDER, F.B. (2004). CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47. 4, 5, 13, 14, 39, 55

MEYER, B. (2003). The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, 660–667, IEEE Computer Society. 27

OBELHEIRO, R.R., BESSANI, A.N., LUNG, L.C. & CORREIA, M. (2006). How practical are intrusion-tolerant distributed systems? DI/FCUL TR 06–15, Department of Informatics, University of Lisbon. 130

OKI, B.M. & LISKOV, B.H. (1988). Viewstamped replication: a new primary copy method to support highly-available distributed systems. In *PODC '88:*

Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing, 8–17, ACM Press. 93

OSTROVSKY, R. & YUNG, M. (1991). How to withstand mobile virus attacks (extended abstract). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, 51–59, ACM Press. 15, 24, 39

PARKINSON, B. & GILBERT, S. (1983). Navstar: Global positioning system – ten years later. *Proceedings of the IEEE*, 71(10):1177–1186. 102

PaX (2001). PaX. http://pax.grsecurity.net/. 40

PEASE, M., SHOSTAK, R. & LAMPORT, L. (1980). Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234. 104

POWELL, D. (1992). Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd IEEE International Symposium of Fault-Tolerant Computing*, 386–395, IEEE Computer Society. 129

PUCELLA, R. & SCHNEIDER, F.B. (2006). Independence from obfuscation: A semantic framework for diversity. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, 230–241, IEEE Computer Society. 27, 130

RABIN, T. (1998). A simplified approach to threshold and proactive RSA. In *CRYPTO '98: Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology*, 89–104, Springer-Verlag. 86

RANDELL, B. (1975). System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, 437–449, ACM Press. 130

REITER, M.K. (1995). The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, vol. 938 of *LNCS*, 99–110, Springer. 93

SANDERS, W.H. & MEYER, J.F. (2000). Stochastic activity networks: Formal definitions and concepts. In E. Brinksma, H. Hermanns & J.P. Katoen, eds., *European Educational Forum: School on Formal Methods and Performance Analysis*, vol. 2090 of *LNCS*, 315–343, Springer. 57, 59, 60, 111

SCHLICHTING, R.D. & SCHNEIDER, F.B. (1983). Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238. 4

SCHNEIDER, F.B. (1990). Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319. 92

SHAMIR, A. (1979). How to share a secret. *Communications of the ACM*, 22(11):612–613. 10, 76

SIEWIOREK, D.P. & SWARZ, R.S. (1992). *Reliable Computer Systems: Design and Evaluation (2nd Edition)*. Digital Press. 39

SKEEN, D. (1982). A quorum-based commit protocol. In *Berkeley Workshop*, 69–80. 93

TRUSTED COMPUTING GROUP (2004). TCG Specification Architecture Overview, revision 1.2. https://www.trustedcomputinggroup.org/groups/tpm/. 42, 96

VERÍSSIMO, P. (2003). Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, vol. 2584 of *LNCS*, 108–113, Springer. 9

VERÍSSIMO, P. (2006). Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81. 9, 19, 40, 41, 85

VERÍSSIMO, P. & RODRIGUES, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers. 2, 3, 34

VERÍSSIMO, P., RODRIGUES, L. & CASIMIRO, A. (1997). CesiumSpray: a precise and accurate global time service for large-scale systems. *Journal of Real-Time Systems*, 12(3):243–294. 102

XU, J., KALBARCZYK, Z. & IYER, R.K. (2003). Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*, 260–269, IEEE Computer Society. 40

ZHOU, L., SCHNEIDER, F. & VAN RENESSE, R. (2002). COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368. 4, 5, 18, 39, 55

ZHOU, L., SCHNEIDER, F.B. & RENESSE, R.V. (2005). APSS: proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security*, 8(3):259–286. 15, 26, 39, 55, 91