Resilient State Machine Replication

Paulo Sousa
pjsousa@di.fc.ul.pt
Univ. of Lisboa*

Nuno Ferreira Neves
nuno@di.fc.ul.pt
Univ. of Lisboa

Paulo Veríssimo
pjv@di.fc.ul.pt
Univ. of Lisboa

Abstract

Nowadays, one of the major concerns about the services provided over the Internet is related to their availability. Replication is a well known way to increase the availability of a service. However, replication has some associated costs, namely it is necessary to guarantee a correct coordination among the replicas. Moreover, being the Internet such an unpredictable and insecure environment, coordination correctness should be tolerant to Byzantine faults and immune to timing failures. Several past works address agreement and replication techniques that tolerate Byzantine faults under the asynchronous model, but they all make the assumption that the number of faulty replicas is bounded and known. Assuming a maximum number of f faulty replicas under the asynchronous model is dangerous – there is no way of guaranteeing that no more than f faults will occur during the execution of the system. In this paper, we describe a resilient f fault/intrusion-tolerant state machine replication system, which guarantees that no more than f faults ever occur. The system is asynchronous in its most part and it resorts to a synchronous oracle to periodically remove the effects of faults/attacks from the replicas.

1 Introduction

Nowadays, one of the major concerns about the services provided by computer systems is related to their availability. Building highly available services involves, on one hand, the design and implementation of correct services tolerant to Byzantine faults [13, 7], and on other hand, the assurance that the access to them is always guaranteed with a high probability. Interestingly, these two tasks can be both accomplished by recurring to replication techniques.

Replication is a well known way to improve 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 replicas. Moreover, the Internet being an unpredictable and insecure environment, coordination correctness should be assured under the worst possible operation conditions: absence of timing guarantees and possibility of Byzantine faults triggered by malicious adversaries.

Several past works address agreement and replication techniques that tolerate Byzantine 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 [1, 3, 9, 8, 6, 10, 5].

This assumption is of course non-controversial in the context of an abstract algorithm design, but care should be taken when the same algorithm is used to implement a system. Systems are supposed to work in the real world, under actual physical constraints and concrete assumptions. So, the system assumption of 'known maximum bound on the number of failures' deserves a closer look. In fact, under the asynchronous model, this type of assumption may be dangerous. There is no way of guaranteeing that no more than f faults will occur during the execution of the system offering the service: unbounded processing and message delays may result in a very long execution time.

Recent works [4, 18, 2, 11] use the proactive recovery approach [12] with the goal of weakening the assumption on the number of faults. The above mentioned assumption 'known maximum bound on the number of failures' is confined to a window of vulnerability, which in turn would allow the algorithms proposed in these papers to tolerate any number of faults over the lifetime of the system. However, this window is again defined under the asynchronous model, and can therefore have an unbounded length. This is specially true in a malicious environment, such as the Internet.

In a recent work, we looked at this problem with the help of a novel theoretical Physical System Model (PSM) [15]. PSM takes in account the environmental resources and their evolution along the timeline of system execution. The model builds on the concept of *resource exhaustion* – the situation when a system no longer has the necessary re-

*Faculdade de Ciências da Universidade de Lisboa. Bloco C6, Campo Grande, 1749-016 Lisboa, Portugal. Tel. +(351) 21 750 0532. Fax +(351) 21 750 0533. This work was partially supported by the FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE).

sources to execute correctly (e.g., bandwidth, replicas). PSM allowed us to introduce the predicate *exhaustion-safe*, meaning freedom from *exhaustion-failures* – failures that result from accidental or provoked resource exhaustion. We showed that, under the asynchronous model, it is theoretically impossible to have an exhaustion-safe replication technique that can only tolerate a bounded number of faults, even if we enhance it with proactive recovery [16].

This theoretical result applies to most of the systems deployed over the Internet, specially to those using replication to achieve fault-tolerance and availability. Practical systems are typically not completely asynchronous under normal operation – some eventual guarantees can be given on the bounds of processing and message delays. However, in an environment prone to malicious faults an adversary may make the system as asynchronous as she or he wants.

Therefore, a replicated system is adequate to be deployed in an asynchronous and insecure environment, such as the Internet, if it does not make timing assumptions and if it does not assume a maximum number of faulty replicas. This could be done by enhancing the system with a detection mechanism responsible for detecting faulty replicas and recovering them. This is relatively easy if replicas can only suffer crash or omission faults, but things get more complicated with Byzantine faults – a malicious adversary may remain dormant until the compromise of $f + 1$ replicas and only deploy the "real" attack afterwards.

Given that it is difficult to detect faults, and assuming that compromising a replica takes some time, in alternative one can calculate the minimum time necessary for $f + 1$ replicas to be compromised and periodically trigger the execution of a recovering procedure. If an appropriate triggering period is chosen and if the recovering procedure is timely executed in every replica, one can guarantee that no more than f faults will occur, for some f. In a recent work we present a system design methodology based on this reasoning and formally prove that it allows the construction of exhaustion-safe systems [16]. The present paper proposes to use this methodology in order to build a resilient state machine replication system, i.e., a system which triggers periodic rejuvenations and in this way guaranteedly tolerates the assumed number of faults. A rejuvenation protocol is presented and the conditions for exhaustion-safety are derived. The protocol is executed by a synchronous and secure component, which guarantees that these conditions are either satisfied or the system switches to a fail-safe state.

2 Resilient State Machine Replication

2.1 State Machine Replication

Almost every computer program can be modeled as a state machine [14]. In particular, we will focus on client/server applications, which also fit under this model. The simplest form of implementing a client-server application is by deploying a single centralized server which 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 one considers that failures 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 failures, one has to replicate the server. The replication degree depends both on the type (e.g., crash, Byzantine) and quantity of the failures to be tolerated. Several protocols have been proposed to implement state machine replication tolerant to crash faults, and some also targeting the Byzantine scenario. Given that, our focus in this paper is on Internet services, we will not make any restrictions on the type of faults that 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 – we call this f*-resilient* systems.

f-resilient systems (with $f \geq 1$) are not necessarily better than systems without fault-tolerance. In fact, given that f-resilient systems have an increased complexity, the performance of the replicated system is typically worse than of the centralized version. The advantage is, of course, the resilience to a certain number of faults. However, 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 the replicas use the same operating system and the service implementation is equal in all of them, then, an adversary only needs to discover how to compromise a single replica in order to easily compromise more than f replicas. On the other hand, even if all the replicas operate over different operating systems and use different implementations, 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, even if replicas are attacked in sequence, the probability of more than f being compromised is significant.

In order to build truly dependable systems, one has to guarantee, by construction, that no more than f faults ever occur during system execution. With this goal in mind, proactive recovery seems to be a very interesting approach: replicas can be 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 that rejuvenations are regularly triggered and have a bounded execution time.

In the case of state machine replication, we argue that despite being difficult to guarantee a bounded execution time on the proactive recovery procedure, because it involves time-consuming tasks such as state transfer, one can devise an architecture under which anomalous recovery times can be dependably detected before more than f replicas being compromised. We propose to apply the Proactive Resilience Model (PRM) [16] to the state machine replication scenario. Under PRM, the proactive recovery procedure is executed in the context of a synchronous and secure distributed architectural component – the Proactive Recovery Wormhole (PRW). The PRW timely execution service is used to proactively recover replicas, guaranteeing that: 1) no more than f replicas are ever corrupted; and 2) the execution of the distributed state machine is never interrupted. Our approach is minutely explained in the next section.

2.2 The State Machine Proactive Recovery Wormhole

In [16], we describe the PRW as an abstract component, and explain that a PRW instantiation is defined by a triple $\langle D, \langle F, T_p, T_d\rangle, S\rangle$, such that:

- D represents the set of *data* which is proactively recovered;
- $\langle F, T_p, T_d\rangle$ represents the *function* F which is periodically triggered with period T_p and timely executed within T_d of each triggering. F makes operations over the data defined in D;
- S represents the set of *self-checking* mechanisms, which have the goal of guaranteeing a fail-safe behaviour.

In this section, we describe an instantiation of the PRW for state machine replication – the State Machine Proactive Recovery Wormhole (SMW). The goal of the SMW is to periodically rejuvenate replicas such that no more than f replicas are ever compromised.

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

- $D_{SMW} = \{$ `OS code`, `SM code`, `SM state` $\}$, where `OS/SM code` is the code of the operating system/state machine and `SM state` is the state of the state machine. These are the three types of data to be periodically refreshed.
- $F_{SMW} = \langle$ `refresh`, $T_p, T_d\rangle$, where the concrete values of T_p and T_d depend on several factors that will be discussed later in the section, and the `refresh` function is presented as Algorithm 1. Each non crashed local SMW $P_i, i \in \{1..n\}$, executes Algorithm 1 at some point of each time period defined by T_p. The precise execution start instant depends on the recovery strategy that will be discussed in Section 2.3.
- $S_{SMW} = \{$switch to a fail-safe state and/or alert an administrator, if the state is not periodically and timely rejuvenated, as specified by T_p and $T_d\}$.

Regarding Algorithm 1, we assume that the state of both operating system and local state machine is stored in volatile Random-Access Memory (RAM). Moreover, the state of the local state machine is periodically saved to stable storage. Also, we assume that the local state machine is automatically started after every boot of the operating system, and that the previous state is loaded from the stable storage.

Algorithm 1 refresh() for each local SMW $P_i, i \in \{1...n\}$

```
1: $shutdownOS()$
2: **if** OS code is corrupted **then** {restore operating system code}
3:     $restoreOScode()$
4: **if** SM code is corrupted **then** {restore state machine code}
5:     $restoreSMcode()$
6: $bootOS()$ {at this point, the OS and the SM can be safely booted
   because their code is correct}
7: wait until state recovery is finished
```

In Algorithm 1, Line 1 shutdowns the operating system, and consequently stops the execution of the local state machine. Notice that the algorithm continues to execute even after the operating system being shutdown. This happens because the SMW does not depend on the operating system, which can be achieved in practice by implementing each local SMW in a PC board. 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 on some read-only memory, and then assessing if it is correct is only a matter of comparing the digest of the current code 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 Lines 4–5, the state machine code can be checked and restored using similar methods to the ones we 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. The local state machine is also automatically started.

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. In Line 7, we wait until a potential state recovery is finished. A generic state recovery mechanism for fail-stop replicas is described in [14]. This mechanism can be easily generalized to the case when we can have Byzantine failures. State recovery introduces an unbounded delay on the proactive recovery procedure, given that it requires the exchange of information through the regular network. Since the regular network is asynchronous, messages sent through it can take an unbounded time to be delivered. However, one can estimate an upper-bound on the delivery time which will be satisfied with high probability in normal conditions.

In the worst-case scenario, i.e., when the code of both the operating and the local state machine is corrupted, Algorithm 1 executes a total of 7 operations. The execution time of these operations can be upper-bounded by a constant $T_{localexec}$, as explained in [17].

2.3 Bounding the Number of Faults

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. Consider that there exists a function $T_{exhaust}(x)$ which returns the minimum time needed to compromise x replicas.

If all replicas are periodically rejuvenated within a period T_p and the rejuvenation execution time is bounded by T_d, then one can guarantee a maximum of f faulty replicas if $T_p + T_d < T_{exhaust}(f+1)$. A straightforward solution to achieve this objective would be to rejuvenate all the replicas at once: the replicas would be simultaneously stopped in a consistent state, 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, the number k of replicas simultaneously recovered should be such that $k \leq f$. If k is greater than f, then the state machine may not continue its operation during a recovery. Moreover, we have to guarantee that the maximum number k' of faults that can occur between rejuvenations is such that $k + k' \leq f$. In the worst case scenario, k' replicas are compromised when a different set of k replicas are recovering.

Each recovering replica executes the code presented in Algorithm 1. 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 $\lceil n/k \rceil$ replica groups are rejuvenated in sequence. Figure 1 illustrates the rejuvenation process. The SMW coordinates the rejuvenation process, 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 $T_{exec} = \lceil n/k \rceil T_{localexec}$.

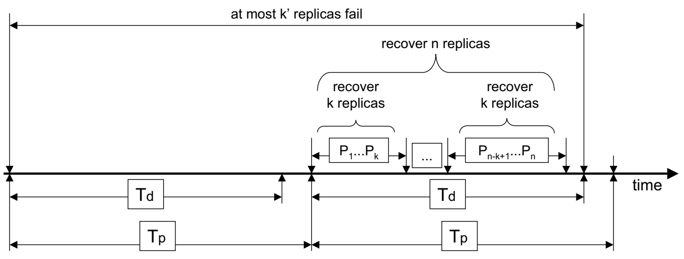

Figure 1. Relationship between the rejuvenation period T_p, the rejuvenation maximum execution time T_d, k and k'.

Therefore, if one sets $T_d \geq T_{exec}$, $T_p > T_d$, and choose k' (with $k + k' \leq f$), such that, $T_p + T_d < T_{exhaust}(k' + 1)$, then no more than f faults will occur. A more detailed explanation can be found in [16].

Given that we need to tolerate at least one faulty replica between rejuvenations[1], k' should be greater than zero. This implies that $f \geq 2$, given that $k \geq 1$ by definition. Therefore, a Byzantine fault-tolerant state machine (where $n \geq 3f + 1$) should apparently have a minimum of 7 replicas in order to satisfy availability and no more than f faults. However, albeit k' faults may be of Byzantine nature, the k faults provoked by the rejuvenation process are fail-silent. So, we need in fact $n \geq 3k' + 2k + 1$ replicas, and thus a minimum of 6 replicas. This is further discussed in [17].

3 Related Work

Castro and Liskov were the first ones to propose the combination of asynchronous state machine replication with proactive recovery [4]. They propose BFT-PR – a Byzantine fault-tolerant, state machine replication algorithm, which uses proactive recovery. BFT-PR can tolerate any number of faults provided fewer than one third of the replicas become faulty within a window of vulnerability.

BFT-PR works mainly under the asynchronous model, but the proactive recovery mechanism makes some extra assumptions, namely about watchdog timers and eventual timely delivery of messages. Watchdog timers are used to periodically interrupt processing and hand control to a recovery monitor that executes the (proactive) recovery procedure. And it is 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 Δ is a constant that depends on the *timeout* values used by the algorithm and that an appropriate choice of Δ allows recoveries at a fixed rate. This suggests that the

[1]We could assume no faults between rejuvenations, but then we would be assuming that the adversary would be unable to compromise any replica.

length T_v of the window of vulnerability can have a known bounded value in normal conditions.

However, given that BFT-PR targets malicious environments (e.g., Internet), the bound on T_v should either be guaranteed under an attack, or it should be possible to timely detect any increase on the window of vulnerability and then take the necessary actions in order to avoid the compromise of the system. The problem is that BFT-PR does not provide adequate mechanisms to achieve any of these goals. Our approach, on the other hand, guarantees precisely this behaviour: in normal conditions, the rejuvenation is periodically and timely executed, and if some abnormal situation occurs, the system switches to a fail-safe state or alerts an administrator, before being compromised.

4 Conclusions

One of the current main challenges is how to build and deploy highly available services on the Internet. The traditional approach, and the one which seems more appropriate, is based on replication. But replication has costs, namely on how to guarantee a correct coordination between the replicas. These costs are even higher on the Internet, given that its unpredictability and insecurity forces the coordination protocols to be Byzantine fault-tolerant and immune to timing failures. Several agreement and replication techniques of this type were already described in the past, but all of them make the dangerous assumption that the number of faulty replicas is bounded by a known value during an unbounded execution time interval.

In this paper, we described a resilient f fault/intrusion-tolerant state machine replication system, which guarantees that no more than f faults ever occur. The system is asynchronous in its most part, using a synchronous oracle – the State Machine Proactive Recovery Wormhole – to periodically remove the effects of faults/attacks from the replicas. We performed a quantitative assessment of the level of redundancy required to achieve resilient state machine replication, i.e., simultaneously securing availability and exhaustion-safety. We see that 6 replicas are required for tolerating one Byzantine failure, versus the 4 replicas required in sheer algorithmic terms, believed until now sufficient.

References

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

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

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

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

[5] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, Oct. 2004.

[6] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *EDCC-3: Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, pages 71–87, London, UK, 1999. Springer-Verlag.

[7] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[8] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578, New York, NY, USA, 1997. ACM Press.

[9] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.

[10] D. Malkhi and M. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, 2000.

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

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

[13] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

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

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

[16] P. Sousa, N. F. Neves, and P. Veríssimo. Proactive resilience through architectural hybridization. DI/FCUL TR 05–8, Department of Informatics, University of Lisbon, May 2005. http://www.di.fc.ul.pt/biblioteca/techreports/05-8.pdf. Submitted for publication.

[17] P. Sousa, N. F. Neves, and P. Veríssimo. Resilient state machine replication. DI/FCUL TR 05–17, Department of Informatics, University of Lisbon, Sept. 2005. http://www.di.fc.ul.pt/biblioteca/techreports/05-17.pdf.

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