A New Approach to Proactive Recovery

Paulo Sousa
pjsousa@di.fc.ul.pt

Faculdade de Ciências da Universidade de Lisboa
Bloco C6, Campo Grande, 1749-016 Lisboa, Portugal*

Abstract. Recent papers propose asynchronous protocols that can tolerate any number of faults over the lifetime of the system, provided that at most f nodes become faulty during a given window of time. This is achieved through the so-called proactive recovery, which consists of periodically rejuvenating the system. Proactive recovery in asynchronous systems, though a major breakthrough, has some limitations which we identified in a recent work. In fact, proactive recovery protocols typically require stronger environment assumptions (e.g., synchrony, security) than the rest of the system. In this paper, we take this in consideration and propose a new approach to proactive recovery that is based on an architecturally hybrid distributed system model. In this context, we present a secure real-time distributed component — the Proactive Recovery Wormhole (PRW) — that aims to execute, in a more dependable and effective way, proactive recovery protocols. We also briefly show how PRW can be used in practice to enhance the dependability of an existent proactive recovery based system.

1 Introduction

Nowadays, and more than ever before, system dependability is an important subject because computers are pervading our lives and environment, creating an ever-increasing dependence on their correct operation.

Distributed systems are usually dependent on a set of protocols. Protocol correctness is thus vital to guarantee system correctness. Assessing protocol correctness evolves, on the one hand, analyzing if the protocol is correctly implemented in an *abstract* target computational system at *design time*, and on the other hand, verifying if the protocol executes correctly at *run time*, in a *real* target computational system.

Typically, a computational system is defined by a set of assumptions regarding aspects like the processing power, the type of faults that can happen, the synchrony of the execution, etc. From these collectively one can define the resources the protocol has access to, both in the abstract and in the real target computational system, both at design and at run time. These resources may include CPU, memory, clock and network with a given capacity. In the case of

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

fault-tolerant protocols, it may also include a certain level of replicated components. The violation of these resource assumptions — which we call *resource exhaustion* — may affect the safety or liveness of the protocols and hence of the system.

One of the most interesting approaches to avoid resource exhaustion due to malicious compromise of components is through proactive recovery [11]. The aim of this mechanism is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/failures. If the rejuvenation if performed frequently often, then an adversary is unable to corrupt enough resources to break the system. Proactive recovery has been suggested in several contexts. For instance, it can be used to refresh cryptographic keys in order to prevent the disclosure of too many secrets [8, 7]. It may also be utilized to restore the system code from a secure source to eliminate potential transformation carried out by the adversary [11, 3]. Moreover, it may include substituting the programs to remove vulnerabilities existent in previous versions (e.g., software bugs that could crash the system or software errors exploitable by outside attackers).

Some proactive recovery protocols for asynchronous systems have been proposed in the literature [15, 1, 3]. Despite having different goals, their effectiveness depends on the same assumptions: periodic and timely execution. They assume that the proactive subsystem is regularly executed, and that the rejuvenation operation does not take a very long period to complete. In other words, these proactive recovery works make timing assumptions about the environment, which by definition, can be violated in an asynchronous setting.

Recently, we presented a novel system model that allowed us to prove that it is theoretically impossible to have permanently correct f fault-tolerant asynchronous systems, i.e., asynchronous systems which can only tolerate a bounded number of faults [12]. These include systems that use asynchronous proactive recovery. We illustrated this impossibility result in practice, through the description of a possible attack to CODEX [10] that is based on a time-related vulnerability of the proactive recovery protocol it uses.

In fact, proactive recovery protocols typically require stronger environment assumptions (e.g., synchrony, security) than the rest of the system (i.e., the part that is proactively recovered). In this paper, we take this in consideration and propose a new approach to proactive recovery that is based on an architecturally hybrid distributed system model. In this context, we present a secure real-time distributed component — the Proactive Recovery Wormhole (PRW) — that aims to execute, in a more dependable and effective way, proactive recovery protocols. As the name suggests, PRW is based on the concept of wormholes: subsystems capable of providing a small set of services, with good properties that are otherwise not available in the rest of the system [13]. Wormholes must be kept small and simple to ensure the feasibility of building them with the expect trustworthy behavior. Moreover, their construction must be carefully planned to guarantee that they have access to all required resources when needed.

The paper is organized as follows. Section 2 describes the Proactive Recovery Wormhole component and briefly discusses how CODEX could be redesigned to benefit from it. Then, our conclusions and some directions for future work appear in Section 3.

2 The Proactive Recovery Wormhole

2.1 Intuition

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

This type of requirement make us believe that one of the possible approaches to dependably use proactive recovery, is to execute it in the context of a wormhole: a subsystem capable of providing a small set of services, with good properties that are otherwise not available in the rest of the system [13]. In the past, two incarnations of distributed wormholes have already been created, one for the security area [4] and another for the time domain [14].

We now formally describe this intuition, by defining the Proactive Recovery Wormhole component.

2.2 Description

The Proactive Recovery Wormhole (PRW) is a secure real-time distributed component that aims to execute proactive recovery procedures. The architecture of a system with a PRW is suggested in Figure 1. An architecture with a PRW has a local module in some hosts, called the *local PRW*. These modules are optionally interconnected by a *control network*. This set up of local PRWs optionally interconnected by the control network is collectively called *the* PRW. The PRW is used to execute proactive recovery procedures of protocols/applications (e.g., the CODEX system) running between participants in the hosts concerned, on any usual distributed system architecture (e.g., the Internet). We call the latter the *payload system and network*, to differentiate from the PRW part.

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

The local PRWs are assumed to be fail-silent (they fail by crashing). Every local PRW has a clock with a known bounded drift rate, but the clocks do not need to be synchronized.

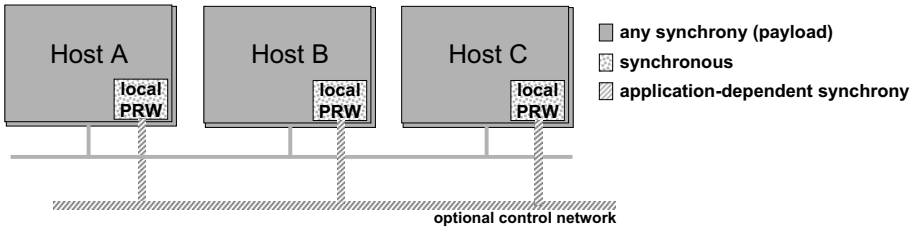

Fig. 1. The architecture of a system with a PRW

As mentioned, the PRW may or may not have a control network. It depends on the specific needs of each protocol/application proactive recovery procedure. For instance, if a proactive recovery procedure only requires local information, then the control network is expendable. Even when the control network is required, its characteristics will depend on the specific requirements of the proactive recovery procedure.

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

Given any function F with a calculated worst case execution time of T_{Xmax}, and a time interval (period) T_P, there exist T_{Pmin}, T_{Pmax} satisfying $T_{Xmax} < T_{Pmin} \leq T_P \leq T_{Pmax} < \infty$, such that F is triggered by the PRW at real time instants t_i (the i-th triggering occurs at instant t_i), with $T_{Pmin} \leq t_i - t_{i-1} \leq T_{Pmax}$, and F terminates within T_{Xmax} from $t_i, \forall i$.

In short, the PRW has the ability to periodically execute well-defined functions in known bounded time. The system architect defines, in design time, the functions corresponding to proactive recovery procedures that have to be periodically executed. This constitutes the PRW *execution plan*. The PRW has admission control procedures that verify, before system startup, if its execution plan is feasible. This feasibility checking involves the calculation of the functions worst-case execution time (WCET) and a schedulability analysis. After a function being accepted by the PRW admission control procedure, it is integrated in the PRW in order to be directly called by it in runtime — if the function would reside in the protocol/application address space, it could have an unbounded execution time.

2.3 Relation with Other Wormholes

The Trusted Timely Computing Base (TTCB) wormhole [4] deals with the problem of handling application timeliness requirements in insecure environments with loose real-time guarantees. This wormhole offers, among others, the following *timely execution* service [14]:

Timely Execution Given any function F with an execution time bounded by a known constant T_{Xmax}, for any execution of F triggered at real time t_{start}, the TTCB is able to execute F within T_{Xmax} from t_{start}.

In [12], we point out that this service could be used to timely execute proactive recovery procedures. Although this is true, we think that there are many advantages

in using a wormhole specifically tailored for proactive recovery. First, it requires potentially weaker environment assumptions. The TTCB requires a control network with some strong properties such as bounded delay, confidentiality, authenticity, whereas the PRW requirements are protocol/application dependent and can thus result in weaker environment assumptions. Second, the PRW is simpler than the TTCB. By incorporating only one service, the PRW will have a simpler design and implementation, which can be more easily validated and achieve better performance. Finally, because the ultimate goal of the functions executed by the PRW periodic timely execution service is known — to proactively rejuvenate systems — and there is a limited number of rejuvenation procedures, they can be grouped into a limited number of function types. This may contribute to a even better performance and reduces the complexity associated with the calculation of the WCET of arbitrary code.

2.4 Application Examples

In [12] we describe an attack to CODEX that explores one flaw on its assumptions. CODEX implicitly assumes that although embracing the asynchronous model, it can have access to a clock with a bounded drift rate. CODEX requires the existence of clocks with bounded drift rate to allow the periodic and timely triggering of its asynchronous proactive secret sharing protocol (APSS) [15]. But, by definition, in an asynchronous system no such bounds exist [6, 5]. Moreover, if an adversary gains access to the clock, she or he can arbitrarily change its progress in relation to real time.

We can make CODEX immune to this type of attacks, by redesigning it in order to use the PRW component presented in Section 2.2. A straightforward solution would be to execute the same unmodified APSS protocol through the PRW periodic timely execution service. However, given that local PRWs are synchronous and secure by construction and that the control network can be as synchronous and secure as needed, we could certainly derive a more simpler protocol, similar to the one described in [8]. We let as future work the design of a proactive secret sharing protocol tailored for the PRW environment assumptions.

3 Conclusions and Future Work

We proposed a new approach to proactive recovery, with the goal of amplifying its usefulness and dependability. This was achieved by revisiting the system and the proactive recovery subsystem under an architecturally hybrid distributed system model, and using a wormhole — the Proactive Recovery Wormhole (PRW) — to implement the latter.

As future work, we intend to implement a experimental prototype of the proposed PRW. We will use this prototype to evaluate the dependability of our wormhole-based proactive recovery approach. This will also involve the design and implementation of proactive recovery protocols suited to be executed by the PRW. In this context, we are currently working on redesigning the proactive secret sharing protocol proposed in [8] according to the PRW environment assumptions.

In this paper we only focused on how the PRW can be used for periodic rejuvenation. However, we can use the ideas proposed in [9, 2] to enhance the PRW with a service that monitors the system behavior and that triggers recovery whenever some failure is predicted. For instance, in addition to periodically refreshing cryptographic keys, the refresh could also be triggered if there was an indication that the keys were in the imminence of being compromised.

References

1. 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.
2. A. Casimiro and P. Veríssimo. Using the Timely Computing Base for dependable QoS adaptation. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, October 2001.
3. M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.
4. M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 234–252, October 2002.
5. F. Cristian and C. Fetzer. The timed asynchronous system model. In *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 140–149, 1998.
6. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.
7. J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theor. Comput. Sci.*, 243(1-2):363–389, 2000.
8. A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 339–352. Springer-Verlag, 1995.
9. M. Malek, F. Salfner, and G. Hoffmann. Self-rejuvenation - an effective way to high availability. In *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, June 2004.
10. 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.
11. 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.
12. P. Sousa, N. F. Neves, and P. Veríssimo. How dependable are distributed f fault/intrusion-tolerant systems? In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, page to appear, June 2005. Preliminary version available as DI/FCUL Technical Report 05–3.
13. P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 108–113. Springer-Verlag, 2003.
14. P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, August 2002.
15. L. Zhou, F. B. Schneider, and R. van Renesse. Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Cornell University, Ithaca, New York, October 2002.