Practical Hardening of Crash-Tolerant Systems[*]

Miguel Correia
IST-UTL / INESC-ID, Portugal
miguel.p.correia@ist.utl.pt

Daniel Gómez Ferro, Flavio Junqueira and Marco Serafini
Yahoo! Research Barcelona, Spain
{danielgf, fpj, serafini}@yahoo-inc.com

1 Introduction

Many production systems make use of fault-tolerance techniques, such as replication, to prevent faults from disrupting the operation of the system. This is particularly important in large-scale systems where faults are frequent. Many practical fault tolerant systems focus on tolerating crashes because they are frequently observed and easily diagnosed. Observations from real systems, however, have shown that undetected commission faults leading to incorrect behavior instead of crashes do happen in practice. A famous example is given by the internal state corruptions that caused an 8-hour outage of the Amazon S3 service and was diagnosed as follows.[1]

> *A handful of messages had a single bit corrupted such that the message was still intelligible, but the system state information was incorrect. We used MD5 checksums throughout the system (but not) for this particular internal state information. (...) When the corruption occurred, we did not detect it and it spread throughout the system causing the symptoms described above.*

The use of CRCs, MD5 hashes or other error detection codes is common practice in practical distributed fault-tolerant systems to prevent such undetected corruptions. Due to the lack of principled approaches, however, adding these checks manually is a difficult and cumbersome process that is sometimes not effective.

Such observations have led in the recent years to a variety of protocols targeting *Byzantine* failures, especially in the context of state machine replication (also called BFT), as for example PBFT [1]. Assuming Byzantine failures is a very weak assumption on the behavior of faulty processes and thus leads to more robust replicated systems. Despite the good performance of BFT systems and the presence of complex prototypes of realistic distributed systems such as [2], the industry has not adopted BFT, to the best of our knowledge, and has favored instead the hardening of crash-tolerant systems through error detection. There can be multiple explanations for this. BFT requires a higher degree of replication than crash tolerance. Furthermore, applying state machine replication in scalable systems can be a difficult endeavor, especially as many distributed systems use weaker application-specific consistency models where active replication is not a suitable design choice.

In this paper, we propose an approach to harden processes of crash-tolerant systems in a sound and transparent manner, relieving developers of the burden of deciding where to place error detection checks. We focus on tolerating *Arbitrary State Corruption* (ASC) faults, where the state of a process can be modified by faults but not its code. We need this restriction to reason about local error detection guarantees, since we cannot reason about the local behavior of a process that does not follow its specification. Hardening guarantees the *error isolation* property, which reduces non-silent faults into crashes or omission, making the hardening of crash-tolerant systems into ASC-tolerant ones trivial.

One can harden processes of a variety of ASC-tolerant distributed systems, ranging from state machine replication to scalable eventually-consistent storage.

2 ASC hardening

ASC model. We propose a new ASC fault model, which allows faults to arbitrarily modify all variables of a faulty process state. The process state includes the program counter, so faults can let the control flow of a faulty process jump to an arbitrary instruction. Faults can occur an unbounded number of times and at any point in time given that at most one fault occurs during the processing of a message.

We consider a data integrity property called *fault diversity* that corresponds, in the ASC model, to the cryptographic assumptions made about the strength of an adversary by protocols using the Byzantine fault model. Assume that v is a variable of the state of a process and that the value of v is replicated in a replica variable v'. When a fault modifies the value of v, this will be different from the one of v'. After the fault, the condition $v \neq v'$ is only required to hold until v or v' are modified again.

[*]This work has been partially supported by the SRT-15 project (ICT-257843), funded by the European Community.

[1]http://status.aws.amazon.com/s3-20080720.html

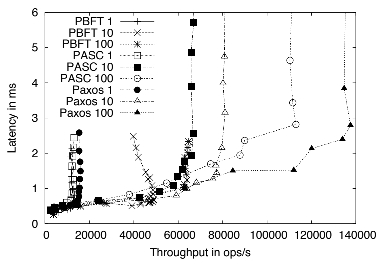

Figure 1: 0/0 benchmarks w. batching

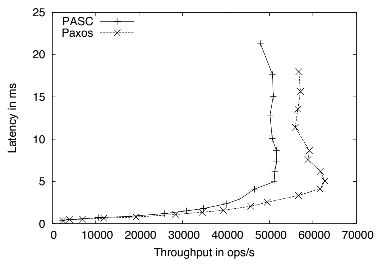

Figure 2: Zookeeper benchmark

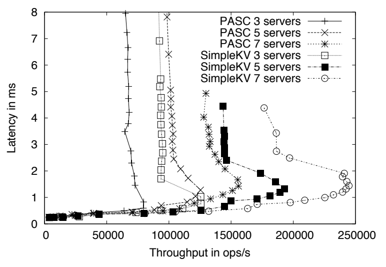

Figure 3: Eventually-consistent store

Hardening processes. Hardening can be applied to any process following the event-based paradigm. Hardened processes are guaranteed to respect *error confinement*, which is defined as follows. Let m be a message with some corrupted field. No correct recipient of m modifies its state according to m. If a faulty recipient modifies its state according to m, it crashes before sending any output message. Our notion of "corrupted" at a time t is formally defined based on the correct state of the process at t. Any divergence from the correct state, be it through a fault directly corrupting a variable or through internal error propagation, is considered as corruption.

A hardened process keeps two local replicas of the process state. A hardened process guarantees that if a variable is corrupted when an output message is sent, then it is different from its replica. Error isolation can thus be guaranteed by attaching a CRC of the output message replica to each output message.

The original event handler, i.e., the procedure which would have handled the input message in the original non-hardened process, is executed twice, on the original and on the replica states. In both executions, all variables are compared with their replicas before being read and the process crashes in case of a mismatch. This is necessary to prevent error propagation. The changes to the original state caused by the event handler are not directly applied. This is to prevent the presence of incomplete state updates due to control flow faults. A hardened process first computes a set of incremental updates of the original state, then executes event handling on the replica state, and finally applies the incremental changes to the original state.

PASC library. In order to make hardening practical, we implemented it as a Java library called PASC. PASC is a runtime environment that wraps processes, transparently replicating their state and executing checks. The user needs to specify the state of processes and implement the event handlers of its distributed algorithm as implementations of PASC classes. During the execution of the protocol, the runtime accepts a message as input, execute the corresponding message handler, and produces output messages. Beyond defining message handlers and the process state, implementing message passing is left to the developer for better flexibility.

3 Use cases

We are currently doing performance optimization of PASC using a cluster of machines with eight 2.5Ghz cores and 16 GB of memory, connected by a Gigabit network. We implemented the Paxos state machine replication protocol [4] with and without PASC hardening, using $2f + 1$ replicas to tolerate f ASC-faulty and crashed processes, respectively. The performance of PASC Paxos using empty requests and replies is comparable to Paxos and better than the C++ PBFT library (see Figure 1). We then implemented a subset of Zookeeper [3] as a state machine on top of Paxos. We kept the state machine external to PASC to reduce memory and CPU costs. Since PASC-Paxos makes all correct replicas agree on the order of operations in presence of ASC faults, we can simply use voting to tolerate ASC-faulty state machines. In terms of maximum throughput, PASC Paxos achieves only 10-20% less throughput than Paxos and has a small latency overhead (see Figure 2). The steady-state JVM user memory occupation is very similar because the Zookeeper state machine is not replicated by PASC. We also implemented a simple eventually-consistent key-value store called SimpleKV (see Figure 3). The throughput scalability with and without hardening is similar.

References

[1] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186, Berkeley, CA, USA, 1999.

[2] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP*, pages 277–290, 2009.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *ATC*, 2010.

[4] L. Lamport. The part-time parliament. *ACM Transactions on Computing Systems (TOCS)*, 16(2):133–169, 1998.