Encoded Protocols: Overhead Analysis on Elections

Diogo Becker Christof Fetzer
Technische Universität Dresden, Germany
{diogo,christof}@se.inf.tu-dresden.de

Introduction

Practical distributed systems are typically built under a crash-fault model. Recently, arbitrary faults such as bit flips have been observed surprisingly often [5], and have disrupted large services such as Amazon S3 [2].

We present a framework for building distributed protocols which *automatically* improves their fault coverage by means of an encoded processing compiler [4]. Although encoded protocols cannot withstand attacks by malicious adversaries, they can tolerate a wide variety of *non-malicious arbitrary faults*.

This preliminary work focuses on leader election, a fundamental primitive in distributed systems. In these protocols, a bit flips can possibly violate liveness and safety properties, for instance, by never electing a leader, or by electing more than one leader at the same time. We implement two election algorithms in our framework and experimentally analyze the transformation's overhead on CPU utilization and election time.

Encoding Protocols

Encoded protocols are functionally equivalent to their unencoded counterparts in fail-free executions. We automatically encode protocols written in C with the encoding compiler by Schiffel *et al.* [4]. We use AN-encoding, which detects bit flips in the protocol's state and hardware design faults with high probability by adding redundancy to the state. In short, each 32-bit value is encoded in a 64-bit value by multiplying it by a prime constant A. The domain of an encoded variable is divided into a few valid values (multiples of A) and many invalid values (not multiples). Furthermore, operations are transformed to support encoding, *i.e.*, they take encoded arguments and produce encoded results. Bit flips during *operation* or *transmission* invalidate encoded values.

Because these transformations introduce processing overhead, we built a framework where only the protocol is encoded, leaving untouched the transmission and reception of messages (implemented with TCP/IP), and scheduling of alarms. Figure 1 shows a process in our framework – $e(x)$ means x is encoded. The top-most layer is the encoded protocol. A protocol can send and receive messages (left), schedule alarms (right), and query a non-synchronized local clock (not depicted in the figure). Scheduling of alarms and transmission of messages are handled in an event loop (bottom layer).

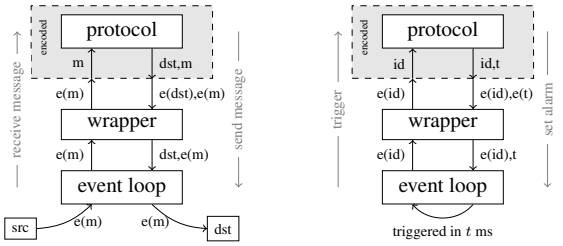

Figure 1: Framework for encoded protocols: send-receive flow (left), alarm-trigger (right)

Alarms and processes are identified and addressed with an integer. We assume no process identifies itself as any other process. This can be ensured by starting processes in different machines, each having its own binary file. Additionally, we assume that clocks do not fail.

The wrapper is responsible for decoding the necessary fields of messages and alarms. To be correctly translated in an IP address, the destination of a message, $e(dst)$, has to be decoded first. Similarly, to schedule an alarm, the time field $e(t)$ has to be decoded. In contrast, alarms can be identified by their encoded *id* uniquely, thus no decoding is necessary. Note that the payload of a message is transmitted encoded. If the sender *id* is required by the receiver, it has to be added to payload by the protocol.

Our framework also allows the protocol to be run unencoded with no source-code modification. In this case, the wrapper is simply removed, and the protocol communicates directly with the event loop layer via the same interface. Unencoded compilation is however necessary.

Fault-tolerance. In Fig. 1, when a bit flip inside the shaded area is detected, the process is aborted. Nevertheless, bit flips might propagate to messages or alarms when occurring outside the shaded area, *e.g.*, wrapper, on the wire, etc. To *virtualize* these failures, the wrapper discards invalid messages or alarms in both ways. The protocol treats then such failures as normal performance/omission failures. Note that, late messages and alarms are possible even with no bit flips and are detected with help of the clock inside the (encoded) protocol.

Election Algorithms. As example applications we implemented two leader election algorithms in our framework. hale is a strong leader election algorithm by Fetzer and Cristian [3]. rotel4 is the (n+4)-stable algorithm by Aguilera *et al.* [1] which rotates the leader on each crash, implementing a weak leader election (Ω).

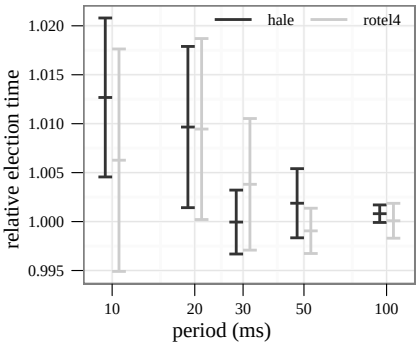

Figure 2: Election time of encoded version relative to original for different periods

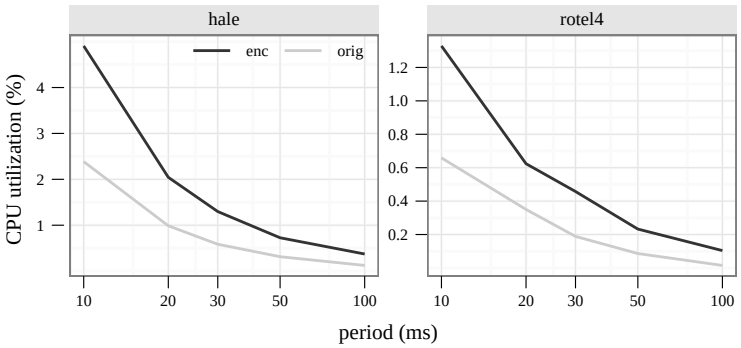

Figure 3: CPU utilization of the encoded (enc) and non-encoded (orig) versions with both algorithms

Overhead Analysis

All of our experiments were performed with 3 election processes, each of them in a workstation with 2 quad-core 2.0 GHz Xeon processors, 8 GB of RAM, and a Gigabit Ethernet interface. The operating system is Debian 5.0. The experimental results are always over 5 runs, and our settings are in ideal conditions, *i.e.*, processes do not crash, links are up and timely, and there are no other jobs running on the machines.

The experiments have three input variables: The algorithm (hale and rotel4), encoded or not (enc or orig), and the period the leader sends heartbeats (*i.e.*, EP parameter in hale, δ in rotel4). The lower the period, the more work processes have to do because more messages are transmitted. hale is configured to elect the leader only when the majority supports it, while rotel4 has no leader support guarantees.

Election Time. Figure 2 shows the election slowdown with the encoded protocol, *i.e.*, the relative election time of the encoded versus the original version (mean and standard deviations are reported). The encoding has no statistically significant impact on the relative election time of rotel4, which stays around 1. Because hale is a more complex algorithm, its encoding results in an overhead of 0.5% to 2% when the period is as low as 10 ms.

CPU Utilization. The CPU utilization (Fig. 3) shows an overhead of 2 times for both algorithms with a period of 10 ms. Although the absolute CPU utilization reduces with longer periods, the overhead increases. When the period is 100 ms, the CPU utilization of the non-encoded versions measured by the operating system is 0. Therefore, we believe that increasing overhead might be due to measurement error (performed via `proc` filesystem).

Message and State Sizes. The encoded protocols send the same message patterns as in the original protocols, so the overhead is simply due to encoding. Because encoding transforms every 32-bit word in a 64-bit word, the space overhead for messages and state is 2 times.

Future Work

Elections are known to be lightweight protocols. We plan to measure the overhead of more elaborated protocols, for example, reliable and atomic broadcast protocols. We expect to see higher CPU utilization and latencies, and, in this sense, we see two interesting problems to be studied. First, how to move the bottleneck to the network interface by running each encoded instance in multiple processors. Second, how to *partially encode* protocols, such that high throughput is achievable and the automatic reliability enhancements are still exploitable.

Additionally, codes more robust than AN can detect, for instance, control flow errors and bit flips on the address bus [4]. In fact, we conjecture that the ANBD-code is sufficient to detect and tolerate all relevant non-malicious faults. We plan to study theoretically and practically (using fault injectors) whether this claim holds.

References

[1] M. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001.

[2] Amazon S3 Team. Amazon S3 Availability Event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, Aug. 2011.

[3] C. Fetzer and F. Cristian. A highly available local leader election service. *IEEE Trans. Softw. Eng.*, 25:603–618, September 1999.

[4] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *Computer Safety, Reliability, and Security*, volume 6351 of *LCNS*. Springer Berlin / Heidelberg, 2010.

[5] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of SIGMETRICS '09*. ACM, 2009.