Systematic Testing for Distributed Systems

Michael Walker and Colin Runciman

Department of Computer Science, University of York, UK
{msw504,colin.runciman}@york.ac.uk

Abstract. We propose a method of applying systematic concurrency testing (SCT) to distributed systems. SCT enables us to verify assertions for many possible executions of a concurrent program, varying according to scheduling decisions. Distributed systems are similar to concurrent programs in the message-passing style, but messages sent over the network may be lost, re-ordered, or duplicated. Prior approaches avoid these problems by assuming that communications are reliable. Our approach does not have this limitation, and allows a variety of network failure cases to be tested.

Keywords: concurrency, distributed systems, message passing, networking, schedule bounding, swarm testing, systematic concurrency testing.

1 Introduction

Testing concurrency is traditionally difficult due to the scheduler. A program can be run multiple times and produce multiple results, whereas effective testing fundamentally assumes *determinism*. Systematic concurrency testing (SCT) is a family of techniques[1, 2], all with the same general aim: to try to find bugs in concurrent programs, more reliably than just running a program several times.

Distributed systems can be thought of as concurrent programs using message passing as the only communication mechanism, with no shared memory. Erlang is an example of a highly concurrent language using this model[3]. When all communication happens through message passing, in principle it doesn't matter how the threads or processes are assigned to machines.

Protocols such as TCP ensure reliable in-order message delivery in the event of network disruption. Protocols such as UDP do not, and message loss can occur. TCP is the most common protocol used on the Internet, but sometimes UDP is used in cases where the overhead of TCP is great enough to offset the cost of occasional message loss, or when different guarantees are desired. A typical example is real-time online games, where the need for low latency trumps the cost of occasional data loss.

Prior work in this field, by Deligiannis *et al.*[4, 5], assumes reliable in-order message delivery. Their P# work is an extension of C# which allows abstracting away the network layer and testing different interleavings. By modelling network behaviour explicitly, we can do away with this reliability assumption at the cost of some additional complexity.

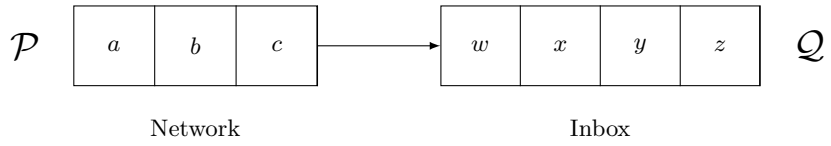

Fig. 1. The directed network link from process $\mathcal{P}$ to process $\mathcal{Q}$.

The paper is organised as follows: Section 2 introduces the network model, showing how it unifies network nondeterminism with scheduler nondeterminism; Section 3 discusses how to guide testing towards diverse network conditions; Section 4 shows a simple domain-specific language we use to express processes; and Section 5 summarises the current state and points to future work.

2 Modelling the Network

SCT techniques assume that any nondeterminism in a program comes solely from the scheduling decisions made. If sending a message from one process to another may nondeterministically fail, that assumption is violated. A similar problem arises in multi-core concurrency when testing *relaxed memory*, as writes to shared memory become visible to other threads nondeterministically. Zhang *et al.*[6] proposed an elegant approach in their 2015 paper, which solves the problem by introducing additional *shadow threads*. Shadow threads are threads which do not exist in the actual program, but are introduced to control other sources of nondeterminism: executing a shadow thread corresponds to that action happening. We can use the same technique here.

In Figure 1, we show the directed network link from process $\mathcal{P}$ to process $\mathcal{Q}$. It is modelled as two queues, where one is the "network" and one is the "inbox". Messages on the network can be interfered with, whereas those in the inbox cannot.

To model the network behaviour, we introduce four shadow threads for each directed network link: **commit** dequeues from the network and enqueues in the inbox; **drop** deletes the head message in the network; **swap** swaps the two head messages in the network; **dup** duplicates the head message in the network.

More complex network behaviours can be built up from these. For example, the network queue can be re-ordered to $\langle c, a \rangle$ by the sequence $\langle$**swap, commit, swap**$\rangle$. By introducing additional threads to model the network behaviour, we once again have a single source of nondeterminism: the scheduler.

3 Guiding the Testing

By introducing four shadow threads for every directed network link (or eight for every pair of processes), we greatly expand the space of possible executions. This makes a *complete* approach to testing infeasible, and poses difficulty for

naive random testing. With no special consideration, the shadow threads—as a whole—are far more likely to be scheduled than the actual program threads. This prolongs execution and causes a great amount of network disruption. We can tackle this problem through two techniques: biassing and bounding[7–9].

- We can *bias* execution away from the shadow threads, by giving every thread a probability of being chosen at a scheduling point, weighted towards program threads.
- We can *bound* the maximum amount of interference (for example, how far any one message can be re-ordered), and forbid any operations which exceed the chosen bounds.
- In addition, some combinations of network disruptions can simply be forbidden, as they do not lead to a new state. For example, ⟨**swap, swap**⟩, ⟨**dup, swap**⟩, or ⟨**dup, drop**⟩.

It is not obvious which probabilities are the *correct* probabilities for biassing, so picking sensible defaults is tricky. Swarm testing[10] tackles a similar problem, the problem of deciding which features to enable in a fuzz tester. In the swarm testing case, it was found that trying different random configurations of enabled features was more effective than trying to hand-tune a single configuration. We have preliminary results indicating a similar approach works well here, with multiple collections of randomly-chosen probabilities performing better than a single hand-tuned set.

The interaction between biassing and schedule bounding can produce interesting executions. For example, a high probability of message loss combined with a low loss bound would tend to manifest as an initial burst of message loss followed by relatively stable execution. This must be investigated, as some combinations may correspond to known network failure cases.

4 Expressing Processes

We have a simple embedded domain-specific language (EDSL), implemented in Haskell, for expressing distributed systems. Our implementation is based on three primitives: **await** blocks until a message has been received or a timeout elapses; **send** sends a message to a named process; **halt** terminates the current process.

Below is a simple example of a process that echoes every received message:

```
echoServer = do
  await >>= \case
    Just (procId, msg) -> send procId msg
    Nothing -> skip
  echoServer
```

The model assumes all processes are defined at system start. Process names can be communicated, but processes cannot be created or started during execution. It is a limited model, but we think it is sufficient to express interesting distributed algorithms and protocols.

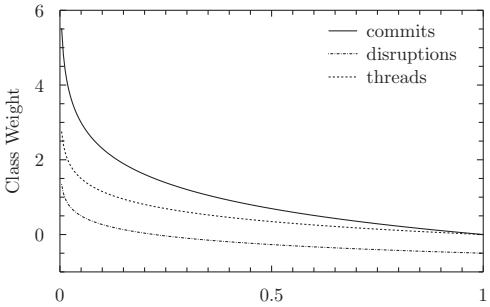

Fig. 2. Class weighting distributions.

5 Current State and Future Work

This work is in its infancy. We have a proof-of-concept implementation showing that the approach is feasible, with few concrete results so far.

Our implementation uses both the swarm testing-like approach to try executions with different probabilities of network disruption, and bounding. We found that assigning probabilities to classes of actions (running a process, dropping a message, etc) and then deriving probabilities for all the actual threads from these to be more effective than assigning a random probability to each thread. We constrain the probabilities so that only the **swap**, **dup**, and **drop** shadow threads may have a probability of zero. Figure 2 shows the weighting distributions for each class of action, where a weight below zero disables the class entirely. Weights are chosen by uniformly picking a value in the range (0,1], and looking up the corresponding weight. Weights are normalised and then transformed into a discrete probability distribution using the condensed table approach[11].

We found that bounding is necessary to ensure termination, as combinations of network actions can easily lead to an unbounded number of messages being produced.

We have applied our technique to Kontiki[12], a Haskell implementation of the Raft consensus algorithm[13], which we seeded with faults. Our technique appears to be effective in finding these faults, but a more careful investigation is called for.

We plan to investigate:

- How fault-tolerant "fault-tolerant" algorithms actually are.
- Do protocols with weaker-than-TCP consistency guarantees correctly achieve what they *do* guarantee?
- Can biassing and bounding be combined to model known network failure cases?
- Can transient network disruption be incorporated into this model?

References

1. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using schedule bounding: an empirical study. In: Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM (2014) 15–28
2. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using controlled schedulers: an empirical study. ACM Trans. Parallel Comput. **2**(4) (February 2016) 23:1–23:37
3. Vinoski, S.: Concurrency with Erlang. IEEE Internet Computing **11**(5) (Sept 2007) 90–93
4. Deligiannis, P., Donaldson, A.F., Ketema, J., Lal, A., Thomson, P.: Asynchronous programming, analysis and testing with state machines. In: 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). (7 2015) 154–164
5. Deligiannis, P., McCutchen, M., Thomson, P., Chen, S., Donaldson, A.F., Erickson, J., Huang, C., Lal, A., Mudduluru, R., Qadeer, S., Schulte, W.: Uncovering bugs in distributed storage systems during testing (not in production!). In: 14th USENIX Conference on File and Storage Technologies (FAST'16). (2 2016) 249–262
6. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2015, ACM (2015) 250–259
7. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '07, ACM (2007) 446–455
8. Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '08, ACM (2008) 362–371
9. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11, ACM (2011) 411–422
10. Groce, A., Zhang, C., Eide, E., Chen, Y., Regehr, J.: Swarm testing. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ISSTA 2012, New York, NY, USA, ACM (2012) 78–88
11. Marsaglia, G., Tsang, W.W., Wang, J.: Fast generation of discrete random variables. Journal of Statistical Software **11**(1) (2004) 1–11
12. Trangez, N.: An implementation of the raft consensus protocol. https://github.com/NicolasT/kontiki (accessed 2016-10-20).
13. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14), Philadelphia, PA, USENIX Association (June 2014) 305–319