Towards conveniently debuggable distributed systems

Stevan Andjelkovic   Daniel Gustafsson

31th Jan, 2022

Recap and motivation

- Distributed systems are hard
- In a previous episode, we showed how to do so called *simulation testing*
  - Run your software system in a simulated world
  - "Digital twin" in business speak
  - Analogy: wind tunnel
  - Speed up time
  - Fast and determinstic system tests
- Today we will show how to build upon these ideas to enable debuggability of live systems
  - Live as in deployed systems, not just systems running in a test environment
  - Time traveling debugger (step forward *and backwards* and see how the system evolves over time)
  - Analogy: black-box in a plane (journal of all events that happened from takeoff to crash)
  - More than merely logging, we can *replay* the exact concurrent execution of the system determinstically
  - Quickly diagnose problems in production
  - Verify that bug fixes work in production environments (not just test environment)

Overview

- High-level technical idea of how we achieve conveniently debuggable distributed systems
- The design of the journal of events (our "black-box")
  - Low performance overhead
  - Also useful for efficient crash recovery
- Demo comparing our journal design vs SQLite
  - Collect performance metrics in the software under test
  - Simple benchmarking library using said metrics
- Show how rich debugging information is computed/derived from journal

Inspiration and prior work

- Erlang
  - Perhaps best known for: lightweight threads and message passing ("everything is a process")
  - Deeper point stressed in Armstrong's thesis (Armstrong 2003): behaviours (better known as interfaces)
    - separate application code ("business logic") which is *sequential* from networking/communcation which is *concurrent*
- Mozilla's `rr` tool, "time traveling debugger", determinstic replay for concurrent executions (very low-level, syscalls)
- Event sourcing (don't necessarily allow determinstic replay, but they could)
- Write-ahead-log (WAL) in databases (atomicity and durability of transactions)
- Chuck's Bandwagon framework
- Martin "LMAX" Thompson et al's Aeron
  - Aeron: Open-source high-performance messaging (Strange Loop 2014)
  - Cluster Consensus: when Aeron met Raft (GOTO 2018)

High-level idea

- Follow Armstrong's advice:
  - Sequential business logic: state machine (function from input and state to output and new state)
  - Event loop which hides the concurrency associated with client requests and internal communcation between state machines
  - The sequential state machines run on top of the event loop, and get fed one event/message/input at the time (assuming the state machines are determinstic, the whole system will be determinstic)
- Keep a journal/write-ahead-log/event store of all events received/processed by the event loop, this can then be used to *replay* a concurrent execution in a determinstic way
  - Snapshots of the application state can be used to truncate/compact the journal so it doesn't grow too big
- While replaying we can dump intermediate states when stepping the state machines, allowing us to visualise how state machines change over time giving us a time traveling debugger a la `rr` but on a application-level (high-level application events) rather than OS-level (low-level syscalls)
- Can you imagine how all these things together *could* enable convenient debugging of distributed systems?

Design of the journal

- Heavily inspired by Martin "LMAX" Thompson et al's Aeron
- Three (virtual) files (clean, active, dirty)
- Circular buffer implemented on top of `mmap`ed byte array
- `recv` zero-copied straight to byte array (and persisted)

Built-in profiler/metrics

- Idea due to Tyler “sled” Neely
- Counters
- Histograms

Demo

- The first version uses SQLite to persist the application log, all reads and writes go through the database.
- The second version uses a on-disk journal which records all incoming data, and an in-memory application log is built from the journal. Writes are therefore indirectly persisted via the journal, and replaying the log lets us rebuild the in-memory application log in case of crashes. All reads go directly via the in-memory log. Snapshots of the journal can be taken and recovered from.
- The two implementations are benchmarked and compared. Metrics are collected via built-in profilers in both versions. In addition we show:
  - How to calculate latency from metrics in said profiler, using Little's law from queuing theory;
  - How metrics can be viewed from a different processes while the service is still running and that metrics persist in case of service crashes.
- For the journaled version we also show how it can be debugged via the snapshot and journal using deterministic replay to show how the state machines change over time (whether the server is running or not).

Summary

- We have shown how to use the journal to:
  - Faster write path than with a database (append only)
  - Get faster crash recovery for free
  - Get all the deterministic testing stuff for free
  - Rich time traveling debugger
- How to add a built-in a profiler and how to use it in benchmarks

Future work

- Add ability to download remote nodes' snapshots and journals in the debugger for a complete complete view of how the system as a whole changed over time (partial views are OK, in case not all nodes wants to give access);
- Save journal prefixes that lead up to crashes in a separate location so they can be debugged after the fact, even if the journal has been rotated (we don't want to keep all of the journal forever due to space limitations);
  - Broken analogy: have several black-boxes, one for each crash...
- Only save keys/topics and offset/length pairs (pointing to disk locations) in-memory and use `sendfile` for zero-copy reads for the journal version of the service;
- Event loop integration: all the above should be implemented on at the event loop level so that state machines (sequential code / "business logic") running on top of it get all this for free.

Thanks! Questions? References:

Armstrong, Joe. 2003. "Making Reliable Distributed Systems in the Presence of Software Errors." PhD thesis, Royal Institute of Technology, Stockholm, Sweden. https://nbn-resolving.org/urn:nbn:se:kth:diva-3658.