Towards conveniently debuggable distributed systems

Stevan Andjelkovic   Daniel Gustafsson

24th Mar, 2022

Recap and motivation

- Distributed systems are hard
- Previously we showed *simulation testing* before deploying
  - Run your software system in a simulated world
  - “Digital twin” in business speak
  - Analogy: wind tunnel
  - Speed up time
  - Fast and deterministic system tests
- Today we will show debugging of deployed systems
  - Analogy: black-box in a plane
  - Time traveling debugger
  - More than merely logging
  - Quickly diagnose problems in production
  - Verify that bug fixes work in production environments

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 simulation testing and 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 from journal

Inspiration and prior work

- Tyler Neely's `sled` database
- Erlang
  - Perhaps best known for: "actor model"-like concurrency
  - Deeper point (Armstrong 2003): behaviours
    - separate application code ("business logic") which is *sequential* from networking/communication which is *concurrent*
- Mozilla's `rr` tool, "time traveling debugger", deterministic replay for concurrent executions (very low-level, syscalls)
- Event sourcing (doesn't necessarily allow deterministic replay)
- 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)

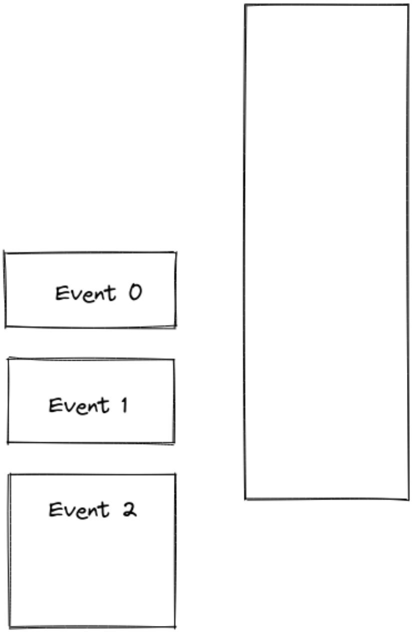
High-level idea

- 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 (client requests and internal communication between state machines)
- Keep a journal/write-ahead-log/event store of all events received by the event loop
  - Used to *replay* a concurrent execution in a deterministic 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, giving us a time traveling debugger a la `rr`
- Can you imagine how all these things together *could* enable convenient debugging of distributed systems?

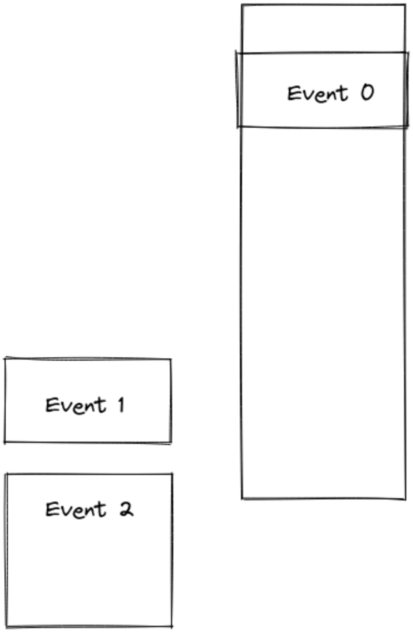
Design of the journal

- Heavily inspired by Martin “LMAX” Thompson et al’s Aeron log buffer
- Circular buffer implemented on top of `mmap`ed byte array
- Three (virtual) files (clean, active, dirty)
  - Analogy: backpacking
  - Doom
- High-performance
  - Contiguous memory/disk (pre-fetching/striding)
  - `recv` zero-copied straight to byte array (and persisted)
  - Lock- and wait-free concurrency

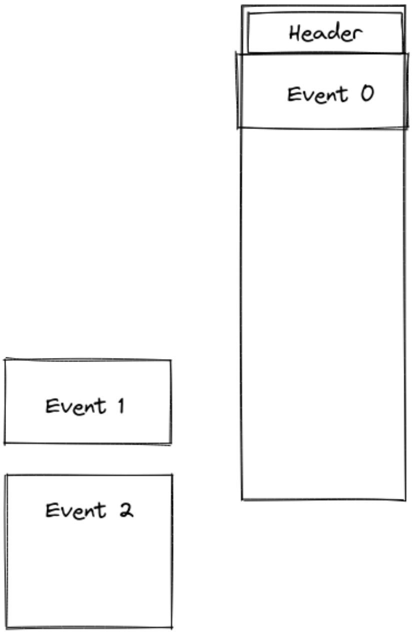
Design of the journal



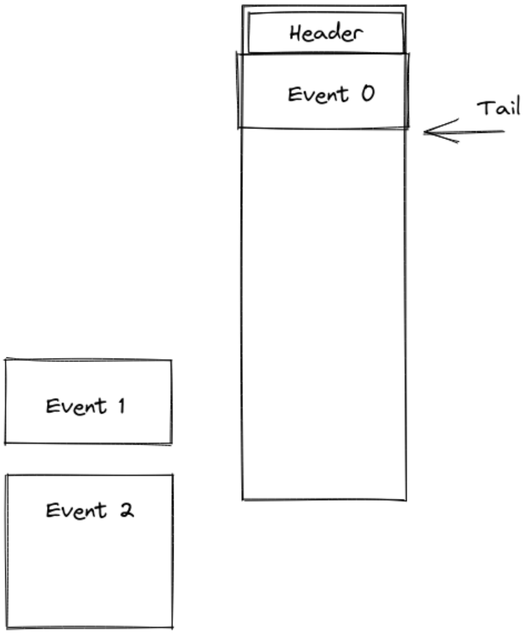
Design of the journal



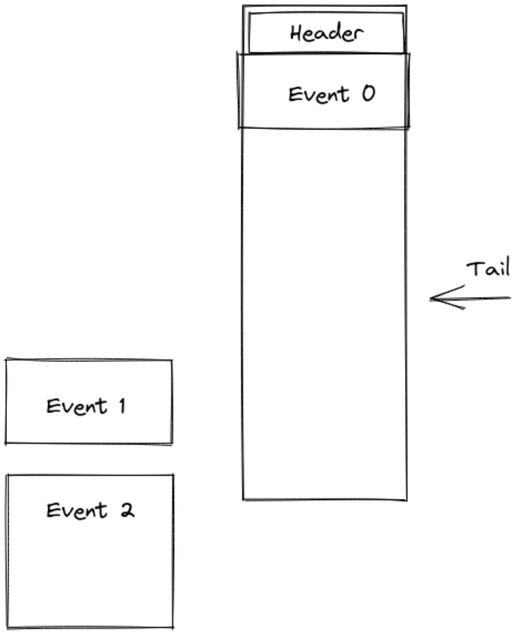
Design of the journal



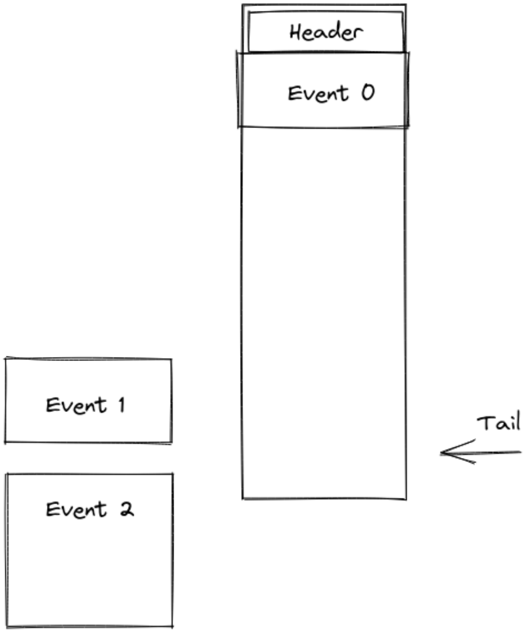
Design of the journal



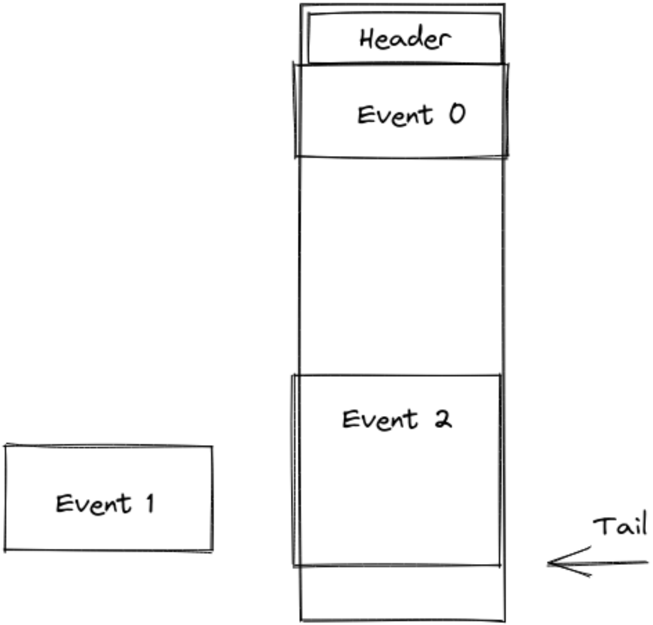
Design of the journal



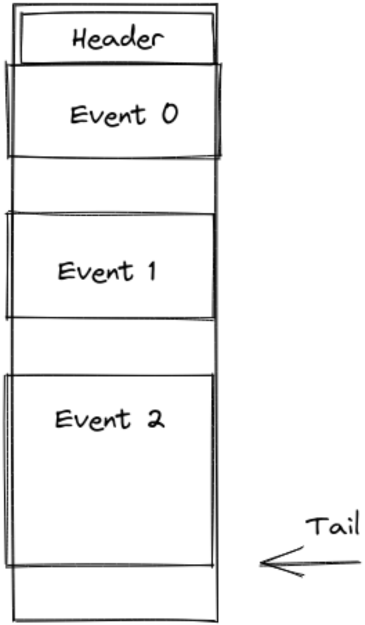
Design of the journal



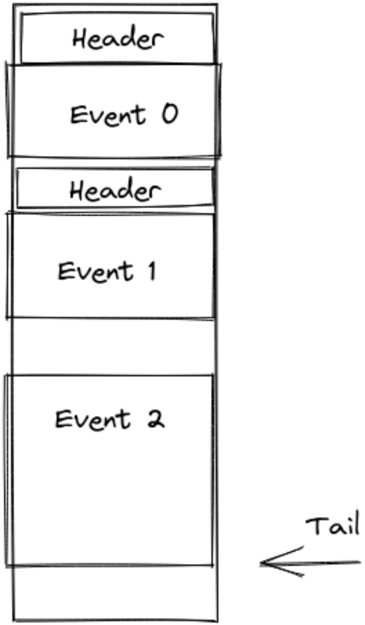
Design of the journal



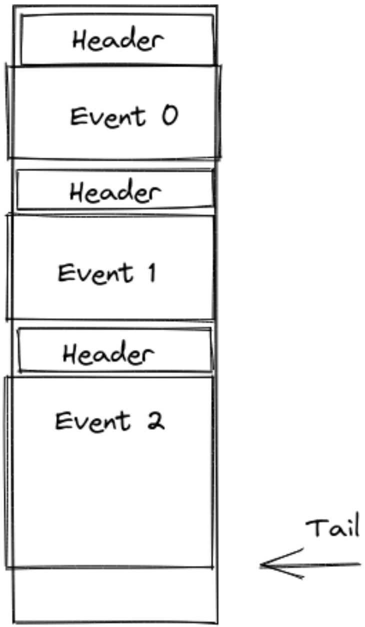
Design of the journal



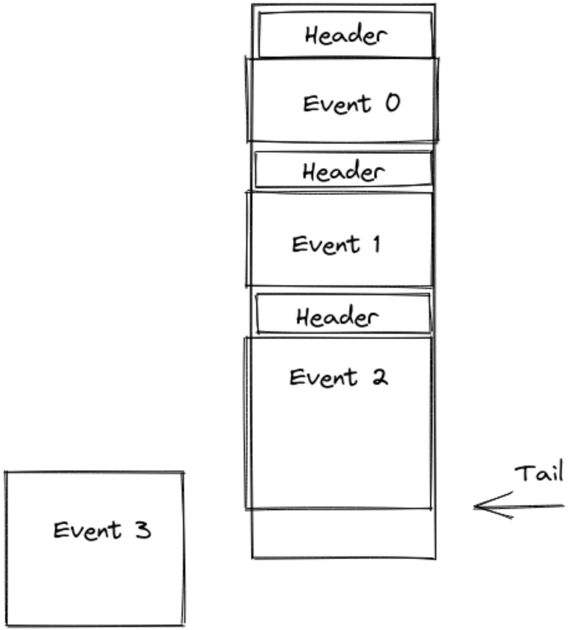
Design of the journal



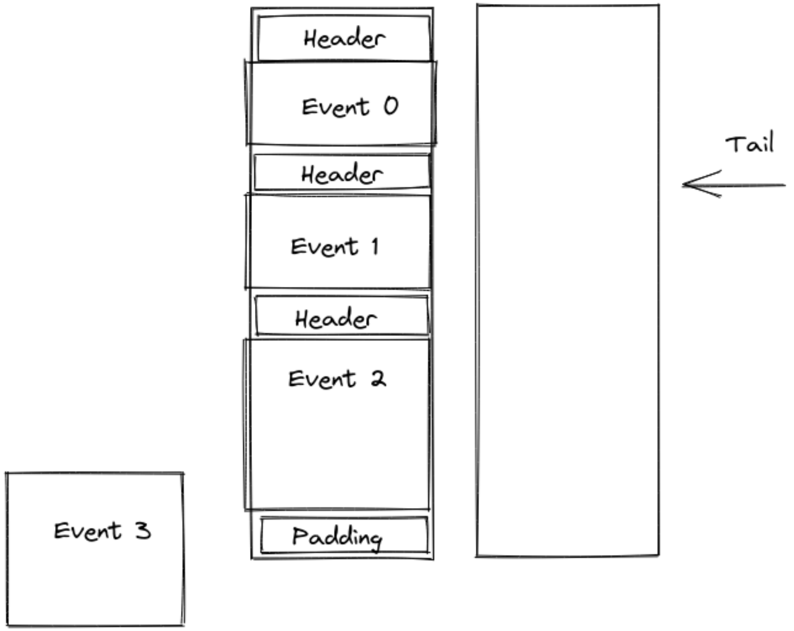
Design of the journal



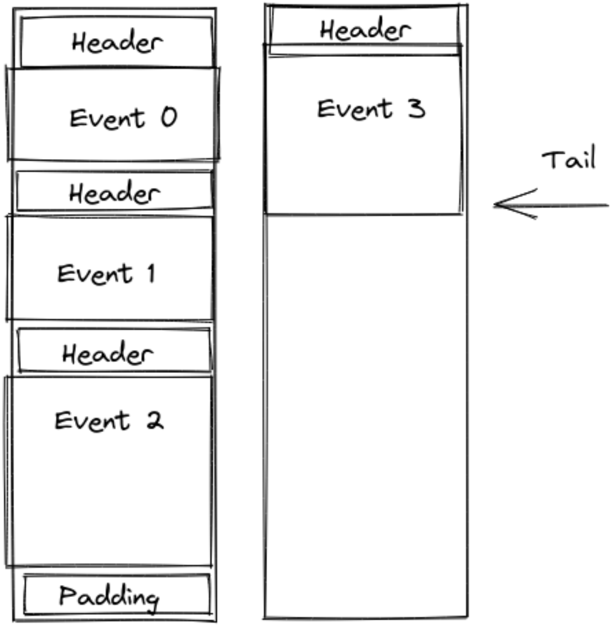
Design of the journal



Design of the journal



Design of the journal



Design of the event loop

```
main() {
  journalOrChannel := createJournalOrChannel()
  fork (worker journalOrChannel)
  startWebserver (requestHandler journalOrChannel)
}
```

1. Start a webserver where the request handlers have concurrent write access to a shared journal/channel (in the SQLite case);
2. Request handlers merely write the entire request into the journal/channel (this gives us a linearised sequence of requests);
3. A separate “worker” thread reads the journal/channel entries and updates/queries the state of the database (sequential access to db).

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.
- 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).

Demo script

```
# Show Dumblog API
echo hi | http POST :8054 # Append to log, returns index, e.g. 0;

http GET :8054/0          # Read at index from log, returns string,
                          # e.g. "hi".

# Metrics
metrics

# Debugger
dumblog-journal debugfilewatch -- --output /tmp/dumblog.df
debugger watch -- /tmp/dumblog.df

# Snapshot and replay

# Benchmark
bench-journal
```

Benchmarks

```
# Use the performance governor instead of powersave (for laptops).
for policy in /sys/devices/system/cpu/cpufreq/policy*; do
    echo "${policy}"
    echo "performance" | sudo tee "${policy}/scaling_governor"
done

# Disable turbo boost.
echo 1 | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo

# The following run is just a (CPU) warm up, the results are discarded.
bench-sqlite

for i in $(seq 10); do
  for j in $(seq 2 12); do
    bench-journal -- $((2**$j)) >> /tmp/bench-journal-$j.txt
    bench-sqlite  -- $((2**$j)) >> /tmp/bench-sqlite-$j.txt
  done
done
```

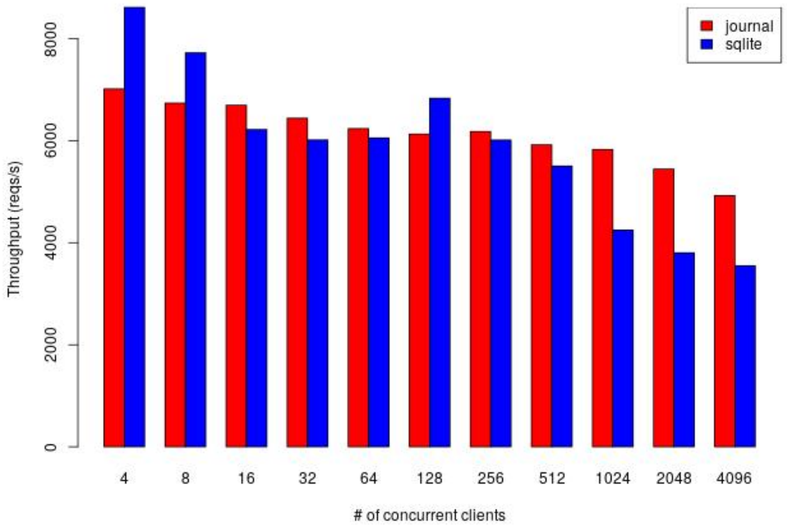
Benchmark statistics

```
Input512=("
  workload                      throughput
  journal-512                   6129.90
  ...
  journal-512                   5756.72
  sqlite-512                    4851.46
  ...
  sqlite-512                    6735.40
")

df512 = read.table(textConnection(Input512),header=TRUE)

t.test(df512$throughput ~ df512$workload, data=df512,
       var.equal=TRUE,
       conf.level=0.95)
```

- Output

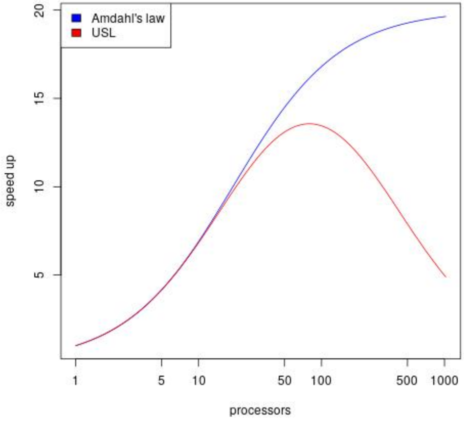
```
mean in group journal-512  mean in group sqlite-512
                 5937.373                  5839.218

t = 0.29646, df = 18, p-value = 0.7703
```

Benchmark plot



Amdahl's law vs Gunther's Universal scalability law

- *C(N) = N / (1 + a(N - 1) + ((b * N) * (N - 1)))*
  - *C* = capacity or throughput
  - *N* = number of processors
  - *a* = contention penalty (time under some kind of lock)
  - *b* = coherence penalty (time to agree, e.g. load the cache line with the shared reference)



Built-in profiler/metrics

- Brendan Gregg's U(tilisation) S(aturation) E(rrors)
- (Atomic) counters (`lock xadd`)
- Histograms (2 + 2ˆ16 counters) uses `log` and `exp` for compression/decompression
- `mmap`ed bytearray for persistence and access for different processes
- No third-party dependencies or extra processes that need to be deployed

Summary

- Recording the inputs and determinism are key to make debugging (and testing) convenient
- Performant journal for recording inputs
  - Could be faster than a database (lock-free, contiguous and zero-copy)
- Several use cases of journal
  - Rich time traveling debugger
  - Together with snapshots we get faster crash recovery
  - Simulation/fast deterministic testing
- Built-in a profiler/metrics and how to use it in statistically sound benchmarks

Future work

- Add ability to download remote nodes' snapshots and journals in the debugger;
- Stash journal prefixes that lead up to crashes
  - Broken analogy: have several black-boxes, one for each crash
- Work out all corner cases with regards to versioning of journal and snapshots, following Chuck's Bandwagon approach;
- Event loop integration.

Even further in the future work

- Zero-copy read path (`sendfile` syscall);
- Use `io_uring` to amortise the cost of syscalls (by batching and doing them async);
- But protocol improvements are likely much more important for performance than these low-level database engine and event loop improvements!

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.