Distributed systems and state machines

Stevan Andjelkovic

ZuriHac lightning talk

11th of June, 2023

Motivation

1. There are many implicit state machines in distributed systems;
2. By making them explicit a bunch of things become easier!

Implicit state machines

Verification

- Property-based testing using state machine
- Kyle “aphyr” Kingsbury’s Jepsen testing tool
- FoundationDB’s simulation testing
- Leslie Lamport’s model checking of TLA+
- Martin Kleppmann’s formal verification work in Isabelle

Development

- Joe Armstrong on Erlang’s behaviours (`gen_server` and `gen_statem`)
- Fault-tolerance via replicated state machines
- Martin Thompson’s work on LMAX disruptor and Aeron

Theory

- Yuri Gurevich’s generalisation of the Church-Turing thesis

Explicit state machines

- Some ideas of what we can do if our state machines were explicit
- Rapid overview, slides and links to longer explainations are available

Simulation testing

- `SM state input output = input -> state -> (state, output)`
- Networking interface
- Property-based testing, fault injection, discrete-event simulation
- For more, see https://github.com/stevana/property-based-testing-stateful-systems-tutorial

Time-travelling debugger

- Record all inputs
- Assuming state machine is deterministic we can recompute the state (and logs!) from the inputs
- This gives us a way of replaying the execution of a state machine and see how its state changes over time
- For more, see https://github.com/stevana/armstrong-distributed-systems/blob/main/docs/domain-specific-debuggers.md

Arrow syntax and hot-swappable code

- `instance Arrow (SM state)`
- Conal Elliott's compiling to categories (`Arrow` modulo `arr`)
- CCCs are first-order, i.e. easily seralisable
- So we can send them over the network and upgrade running state machines without downtime!
- For more, see https://github.com/stevana/hot-swapping-state-machines and https://github.com/stevana/smarrow-lang

Pipelining of state machines

- Serving a request typically involves several stages, e.g.:
  1. Read bytes from socket
  2. Parse bytes into structured data
  3. Validate data
  4. Process the data using our business logic and produce some response
  5. Serialise response into bytes
  6. Write response bytes back to socket
- What if each such stage was run on a separate CPU/core? Pipelining!
- Monitor queue lengths of each stage and shard if stages are slow
- What's the relation to dataflow and FRP?
- For more, see https://github.com/stevana/pipelined-state-machines and https://github.com/stevana/elastically-scalable-thread-pools

Modular state machines

- Pipelining is *horizontal* composition, the outputs of one is fed into the another
- What about *vertical* composition?
- State machines inside state machines?
- Hierarchical states?
- Stack of states / pushdown automaton?
- For more, see https://github.com/stevana/armstrong-distributed-systems/blob/main/docs/modular-state-machines.md

Protocols between communicating state machines

- If we think of state machines as black boxes which provide some API via their inputs
- Then protocols between two black boxes are sequences of input-output pairs
- These too can be described using state machines!
- For more, see https://github.com/stevana/armstrong-distributed-systems/blob/main/docs/specification-language.md

Supervisor trees and deployment

- Tree of supervisors in the nodes and state machines in leaves
- Supervisors' job is to do error handling, if one of their children fail
- Jim Grey's idea of failing fast
- Supervisors contain restart strategies, i.e. in which order to restart the children if one fails
- Use restart strategy as a means of deployment (start up order)
- For more, see https://github.com/stevana/supervised-state-machines

Contributing

- Also see https://github.com/stevana/armstrong-distributed-systems for longer elaborations of the above ideas
- If you have any questions or comments, feel free to get in touch!
- Thanks for listening!