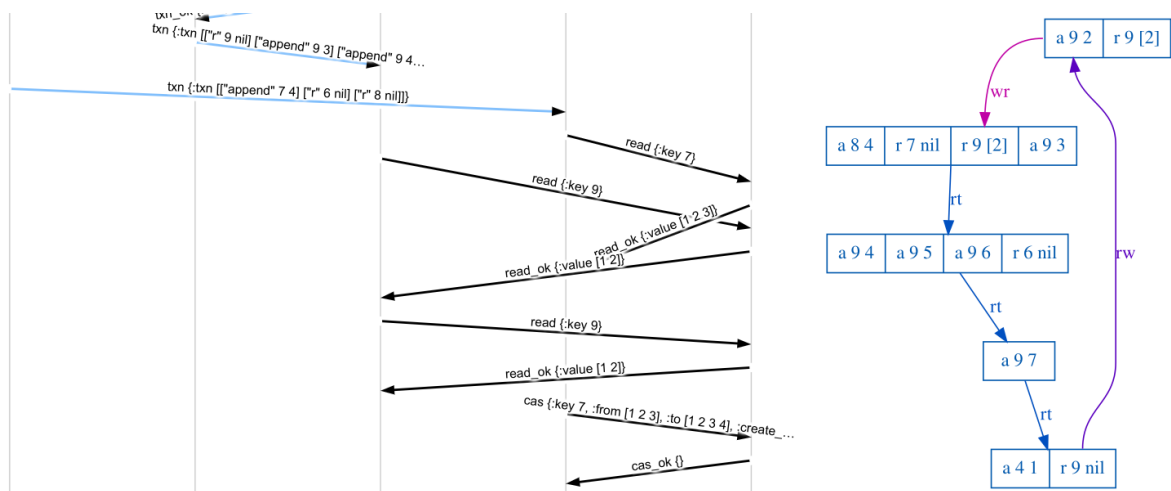

Maelstrom



Maelstrom is a workbench for learning distributed systems by writing your own. It uses the Jepsen testing library to test toy implementations of distributed systems. Maelstrom provides standardized tests for things like “a commutative set” or “a transactional key-value store”, and lets you learn by writing implementations which those test suites can exercise. It’s used as a part of a distributed systems workshop by Jepsen.

Maelstrom provides a range of tests for different kinds of distributed systems, built on top of a simple JSON protocol via STDIN and STDOUT. Users write servers in any language. Maelstrom runs those servers, sends them requests, routes messages via a simulated network, and checks that clients observe expected behavior. You want to write Plumtree in Bash? Byzantine Paxos in Intercal? Maelstrom is for you.

Maelstrom’s tooling lets users experiment with simulated latency and message loss. Every test includes timeline visualizations of concurrency structure, statistics on messages exchanged, timeseries graphs to understand how latency, availability, and throughput respond to changing conditions, and Lamport diagrams so you can understand exactly how messages flow through your system. Maelstrom’s checkers can verify sophisticated safety properties up to strict serializability, and generate intuitive, minimal examples of consistency anomalies.

Maelstrom can help you model how a system responds to different cluster sizes, network topologies, latency distributions, and faults like network partitions. Maelstrom also offers simulated services that you can use to build more complex systems.

It’s built for testing toy systems, but don’t let that fool you: Maelstrom is reasonably fast and can handle simulated clusters of 25+ nodes. On a 48-way Xeon, it can use 94% of all cores, pushing upwards of 60,000 network messages/sec.

Documentation

The Maelstrom Guide will take you through writing several different types of distributed algorithms using Maelstrom. We begin by setting up Maelstrom and its dependencies, write our own tiny echo server, and move on to more sophisticated workloads.

- Chapter 1: Getting Ready
- Chapter 2: Echo
- Chapter 3: Broadcast
- Chapter 4: CRDTs
- Chapter 5: Datomic
- Chapter 6: Raft

There are also several reference documents that may be helpful:

- Protocol defines Maelstrom’s “network” protocol, message structure, RPC semantics, and error handling.
- Workloads describes the various kinds of workloads that Maelstrom can test, and define the messages involved in that particular workload.
- Understanding Test Results explains how to interpret the various plots, data structures, and log files generated by a test.
- Services discusses Maelstrom’s built-in network services, which you can use as primitives in building more complex systems.

Design Overview

Maelstrom is a Clojure program which runs on the Java Virtual Machine. It uses Jepsen to generate operations, record a history of their results, and analyze what happens.

Writing “real” distributed systems involves a lot of busywork: process management, networking, and message serialization are complex, full of edge cases, and difficult to debug across languages. In addition, running a full cluster of virtual machines connected by a real IP network is tricky for many users. Maelstrom strips these problems away so you can focus on the algorithmic essentials: process state, transitions, and messages.

The “nodes” in a Maelstrom test are plain old binaries written in any language. Nodes read “network” messages as JSON from STDIN, write JSON “network” messages to STDOUT, and do their logging to STDERR. Maelstrom runs those nodes as processes on your local machine, and connects them via a simulated network. Maelstrom runs a collection of simulated network clients which make requests to those nodes, receive responses, and records a history of those operations. At the end of a test run, Maelstrom analyzes that history to identify safety violations.

This allows learners to write their nodes in whatever language they are most comfortable with, without having to worry about discovery, network communication, daemonization, writing their own distributed test harness, and so on. It also means that Maelstrom can perform sophisticated fault injection and trace analysis.

Maelstrom starts in `maelstrom.core`, which parses CLI operations and constructs a test map. It hands that off to Jepsen, which sets up the servers and Maelstrom services via `maelstrom.db`. Spawning binaries and handling their IO is done in `maelstrom.process`, and Maelstrom's internal services (e.g. `lin-kv`) are defined in `maelstrom.service`. Jepsen then spawns clients depending on the workload (`maelstrom.workload.*`) and a nemesis (`maelstrom.nemesis`) to inject faults.

Messages between nodes are routed by `maelstrom.net`, and logged in `maelstrom.journal`. Clients send requests and parse responses via `maelstrom.client`, which also defines Maelstrom's RPC protocol.

At the end of the test, Jepsen checks the history using a checker built in `maelstrom.core`: workload-specific checkers are defined in their respective workload namespaces. Network statistics are computed in `maelstrom.net.journal`, and Lamport diagrams are generated by `maelstrom.net.viz`.

`maelstrom.doc` helps generate documentation based on `maelstrom.client`'s registry of RPC types and workloads.

CLI Options

A full list of options is available by running `java -jar maelstrom.jar test --help`. The important ones are:

- `--workload NAME`: What kind of workload should be run?
- `--bin SOME_BINARY`: The program you'd like Maelstrom to spawn instances of
- `--node-count NODE-NAME`: How many instances of the binary should be spawned?

To get more information, use:

- `--log-stderr`: Show STDERR output from each node in the Maelstrom log
- `--log-net-send`: Log messages as they are sent into the network
- `--log-net-recv`: Log messages as they are received by nodes

To make tests more or less aggressive, use:

- `--time-limit SECONDS`: How long to run tests for

-
- `--rate FLOAT`: Approximate number of requests per second
 - `--concurrency INT`: Number of clients to run concurrently. Use `4n` for 4 times the number of nodes.
 - `--latency MILLIS`: Approximate simulated network latency, during normal operations.
 - `--latency-dist DIST`: What latency distribution should Maelstrom use?
 - `--nemesis FAULT_TYPE`: A comma-separated list of faults to inject
 - `--nemesis-interval SECONDS`: How long between nemesis operations, on average

For broadcast tests, try

- `--topology TYPE`: Controls the shape of the network topology Jepsen offers to nodes

For transactional tests, you can control transaction generation using

- `--max-txn-length INT`: The maximum number of operations per transaction
- `--key-count INT`: The number of concurrent keys to work with
- `--max-writes-per-key INT`: How many unique write operations to generate per key.

SSH options are unused; Maelstrom runs entirely on the local node.

Troubleshooting

Running `./maelstrom` complains it's missing `maelstrom.jar`

You probably cloned this repository or downloaded the source and didn't compile it. Download the compiled release tarball instead; you'll find it on the GitHub release page.

If you want to run directly from source, you'll need the Leiningen build system. Instead of `./maelstrom ...`, run `lein run ...`.

Raft node processes still alive after maelstrom run

You may find that node processes maelstrom starts are not terminating at the end of a run as expected. To address this, make sure that if the process passed as `--bin` forks off a new process, it also handles the process' termination.

Example In `bin/raft`

```
1 #!/bin/bash
2
3 # Forks a new process.
4 java -jar target/raft.jar
```

In `bin/raft`

```
1 #!/bin/bash
2
3 # Replaces the shell without creating a new process.
4 exec java -jar target/raft.jar
```

License

Copyright © 2017, 2020–2022 Kyle Kingsbury, Kit Patella, & Jepsen, LLC

Distributed under the Eclipse Public License either version 1.0 or (at your option) any later version.