
Raft

docs passing docs passing crates.io v0.7.0 dependencies 5 of 26 outdated

Problem and Importance

When building a distributed system one principal goal is often to build in *fault-tolerance*. That is, if one particular node in a network goes down, or if there is a network partition, the entire cluster does not fall over. The cluster of nodes taking part in a distributed consensus protocol must come to agreement regarding values, and once that decision is reached, that choice is final.

Distributed Consensus Algorithms often take the form of a replicated state machine and log. Each state machine accepts inputs from its log, and represents the value(s) to be replicated, for example, a hash table. They allow a collection of machines to work as a coherent group that can survive the failures of some of its members.

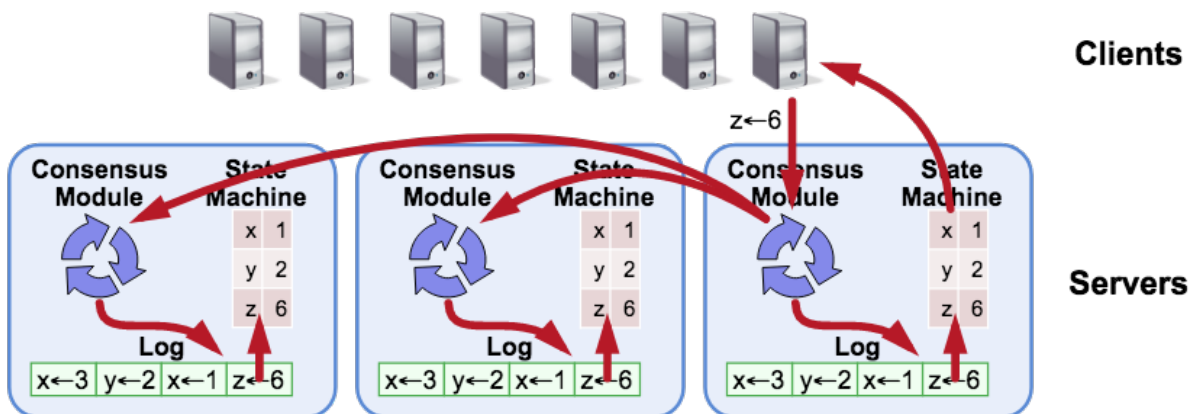
Two well known Distributed Consensus Algorithms are Paxos and Raft. Paxos is used in systems like Chubby by Google, and Raft is used in things like [tikv](#) or [etcd](#). Raft is generally seen as a more understandable and simpler to implement than Paxos.

Design

Raft replicates the state machine through logs. If you can ensure that all the machines have the same sequence of logs, after applying all logs in order, the state machine will reach a consistent state.

A complete Raft model contains 4 essential parts:

1. Consensus Module, the core consensus algorithm module;
2. Log, the place to keep the Raft logs;
3. State Machine, the place to save the user data;
4. Transport, the network layer for communication.



Note: This Raft implementation in Rust includes the core Consensus Module only, not the other parts. The core Consensus Module in the Raft crate is customizable, flexible, and resilient. You can directly use the Raft crate, but you will need to build your own Log, State Machine and Transport components.

Using the raft crate

You can use raft with either rust-protobuf or Prost to encode/decode gRPC messages. We use rust-protobuf by default. To use Prost, build (or depend on) Raft using the `prost-codec` feature and without default features.

Developing the Raft crate

`Raft` is built using the latest version of `stable` Rust, using the 2018 edition. Minimum supported version is `1.44.0`.

Using `rustup` you can get started this way:

```
1 rustup component add clippy
2 rustup component add rustfmt
```

In order to have your PR merged running the following must finish without error:

```
1 cargo test --all && \
2 cargo clippy --all --all-targets -- -D clippy::all && \
3 cargo fmt --all -- --check
```

You may optionally want to install `cargo-watch` to allow for automated rebuilding while editing:

```
1 cargo watch -s "cargo check"
```

Modifying Protobufs

See instructions in the proto subdirectory.

Benchmarks

We use Criterion for benchmarking.

It's currently an ongoing effort to build an appropriate benchmarking suite. If you'd like to help out please let us know! Interested?

You can run the benchmarks by installing [gnuplot](#) then running:

```
1 cargo bench
```

You can check [target/criterion/report/index.html](#) for plots and charts relating to the benchmarks.

You can check the performance between two branches:

```
1 git checkout master
2 cargo bench --bench benches -- --save-baseline master
3 git checkout other
4 cargo bench --bench benches -- --baseline master
```

This will report relative increases or decreased for each benchmark.

Acknowledgments

Thanks etcd for providing the amazing Go implementation!

Projects using the Raft crate

- TiKV, a distributed transactional key value database powered by Rust and Raft.

Links for Further Research

- The Raft site
- The Secret Lives of Data - Raft
- Raft Paper
- Raft Dissertation

-
- Raft Refloated
 - Implement Raft in Rust
 - Using Raft in Rust at RustConf 2018