

Ra

a Raft implementation

By Team RabbitMQ

Karl Nilsson

Son of nil

- Past:
 - .NET (C# / F#),
 - Distsys
 - Implemented Raft in F#
- Author of fez
 - F# to core erlang compiler
 - <https://github.com/kjnilsson/fez>
- t: @kjnilsson

Pivotal.

 RabbitMQ

Pivotal and RabbitMQ

Invested in the rabbit

- Sponsors RabbitMQ development
- Provides RabbitMQ services as part the Cloud Foundry platform.
 - RabbitMQ “tile”
- Provides commercial support for RabbitMQ

Ra

Raft

RabbitMQ

gen_servers

- **Are:**

- Easy to reason about (linear)
- Simple concurrency model
- Stateful
- Simple API (cast, call)

- **Are not:**

- Not replicated
- Not fault tolerant
- Not persistent

Ra (Raft) allows us to implement persistent, replicated state machines.

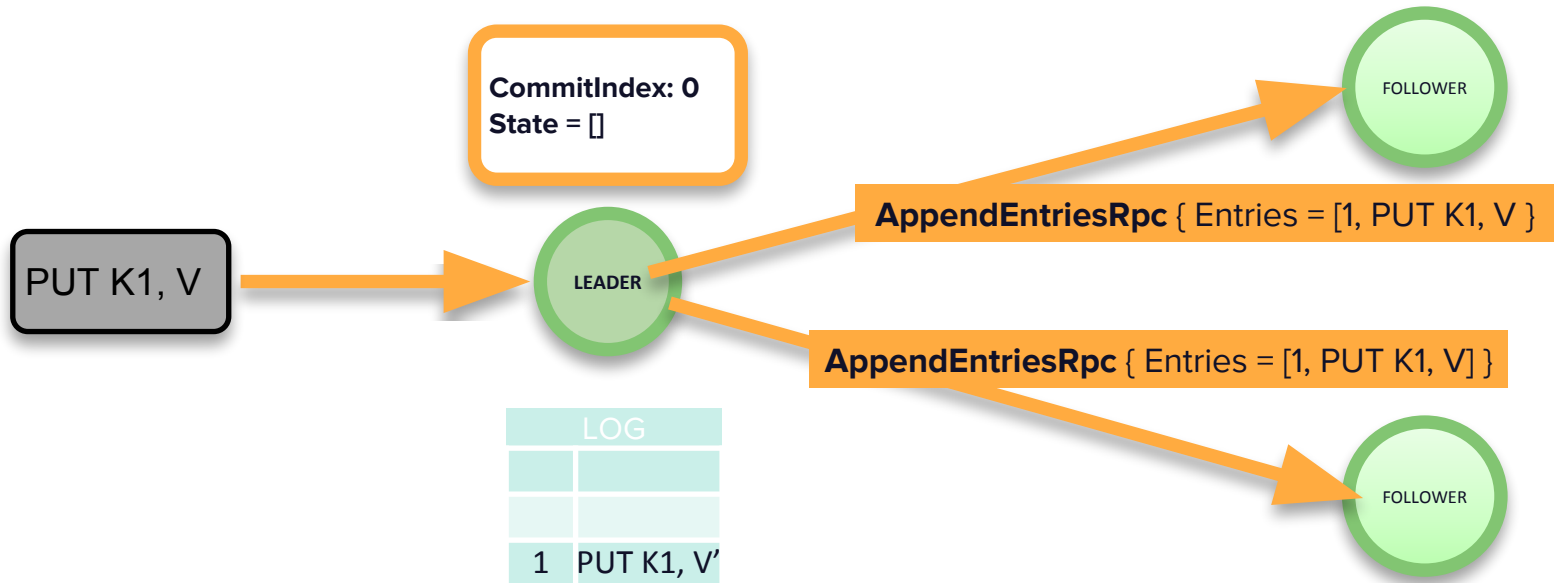
A state machine

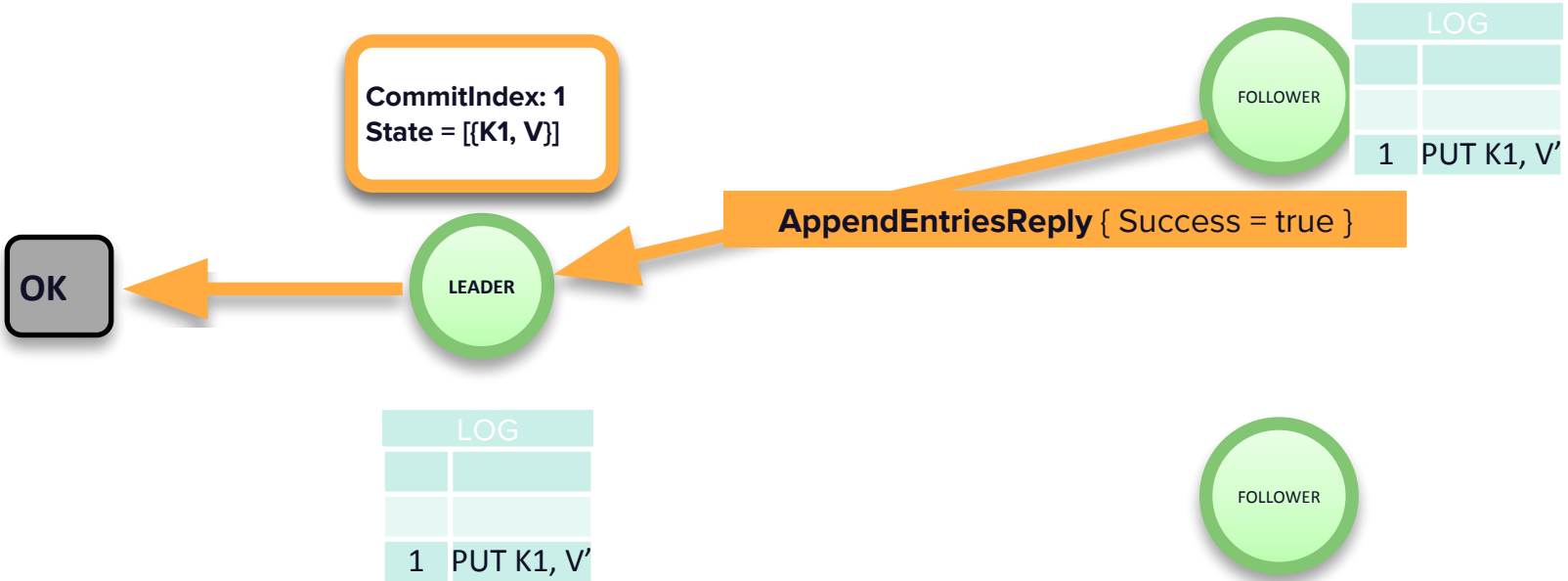
apply: Command -> State -> State

Raft

In two minutes

- Log replication
 - Leader -> follower
 - Persisted to reliable storage (fsync)
 - Quorum ($n/2+1$)
 - Ordered, each log entry is assigned a monotonically incrementing index
- Leader election
 - Safe, based log state
 - Term (epoch) for the time a server is leader for
 - Monotonically incrementing integer





Raft Resources

- The website:
 - <https://raft.github.io/>
- The mailing list:
 - <https://groups.google.com/forum/#!forum/raft-dev>
- The paper:
 - <https://raft.github.io/raft.pdf>
- The thesis:
 - <https://ramcloud.stanford.edu/~ongaro/thesis.pdf>

Using Ra (to make a kv-store)

1. Implement the `ra_machine` behaviour (2 required callbacks)
 - a. `init/1`
 - i. Create the initial state of the state machine
 - b. `apply/4`
 - i. Apply a command to the state machine and return the new state
 - ii. Must be deterministic
 - iii. No side effects inside `apply/4` (!, exceptions, ets/dets operations)
 - c. “Client” api (put, delete, get)
2. Start a cluster of Ra servers
 - a. `ra:start_cluster/3`
 - b. Ra servers are always named and referred to by their `{Name, Node}`.
3. Send commands and interact
 - a. `ra:process_command/2`
 - b. Observe!

Demo! (👤)

Ra provides

- Replication
 - Data safety
 - Primary / replica (leader / follower)
 - Fault tolerance (follower can crash without affecting availability, to a point)
- Persistence
 - Recover state machine state
- Leader election
 - High availability
- Dynamic member changes
- Raft as a library
 - Simple (ish) API

Async API (the not so simple API)

- ra:pipeline_command/2|3|4
- How to ensure messages aren't lost
 - ra:pipeline_command/3
 - ra:pipeline_command(ServerId, Command, Correlation)
 - Correlation
 - Correlation returned as:
 - {ra_event, LeaderId, {applied, [{Correlation, Reply}]}}
 - {ra_event, ServerId, {rejected, {not_leader, LeaderId, Correlation}}}
- Maintaining ordering
 - Use process scoped monotonically incrementing integers as correlations ids
 - Resend when detecting a gap in the correlations returned by the applied event
 - Implement a re-sequencer in the state machine
 - Future: first class sessions in ra?

Making stuff happen in the real world

- Raft state machines are pure functions (no side effects)
- Deterministic
- Replaying the log should result in exactly the same end state
- Effects system models side effects as data.
- Only actioned on the leader

Ra Machine “Effects”

- {send_msg, pid(), Msg}
 - Sends Msg to the pid
 - NB: uses `erlang:send(Pid, Msg, [noconnect, nosuspend])`
- {mod_call, module(), function(), [term()]}
 - Calls an arbitrary function
 - NB: should not block or throw!
- {monitor, process | node, node() | pid()}
 - Ra monitors a process or node on behalf of the state machine
 - Adds a command to the log:
 - {down, pid(), Reason} | {nodeup | nodedown, node()}
- {demonitor, process | node, node() | pid()}
- {release_cursor, RaftIndex, SnapshotState}
 - Allows a state machine to provide a potential snapshot point to ra

Design

Why write our own?

- We wanted to see if it was possible to write an efficient, safe queue implementation
 - Control
 - Experimentation
- Design needed to meet RabbitMQ requirements
 - Many thousands of Ra clusters inside an erlang cluster
 - Distributed erlang
 - Lightweight
 - Minimal number of processes
 - Pure erlang

How did we end up where we are today?

- Initial we wanted it to be an OTP library
 - Embeddable in existing supervision trees.
 - No dependencies
- Initial design
 - Every ra server had their own log that they wrote to
 - Remember fsync
- The initial design provided good throughput
 - But did not scale for multiple processes
 - Fell over after only a dozen or so
 - fsync contention

Centralization

- Access to critical resources needed to be controlled and shared fairly across all the Ra servers running within an erlang cluster
 - Reduce disk contention
 - Dropped OTP library requirement

Log infrastructure

- Write Ahead Log (WAL)
 - All ra servers send async writes to the WAL
 - WAL flushes to disk in batches (`gen_batch_server`) and responds to writers after `fsync`
 - All reads happens from ETS “mem tables”
 - Inspired by LSM Trees (leveldb etc)
- Segment Writer
 - Writes mem tables to per ra-server disk storage
 - No compaction
 - Instead just delete full segments
 - We can do this
- Snapshot Writer
 - Writes snapshots of ra machine state

Fault detection


- Vanilla Raft uses the replication message as heartbeat
- Not feasible for thousands of Ra clusters (too chatty)
- Ra:
 - Erlang monitors (followers monitor leaders and trigger election timers
 - nodedowns, process crashes
 - aten: node failure detector
 - <https://github.com/rabbitmq/aten>
 - Provides timely hints of erlang nodes being slow / unavailable / partitioned
 - If the nodes is the node the leader is running on the follower may start a pre_vote election.

Testing

Jepsen

- ra-kv-store
 - key-value state machine implementation running a similar jepsen test suite to etcd
 - <https://github.com/rabbitmq/ra-kv-store>
 - Found bugs
 - Now passes
- Standard jepsen tests from original RabbitMQ test
 - <https://github.com/rabbitmq/jepsen>
 - Found bugs
 - Now passes
- Gives us confidence we've squashed the most obvious bugs at least!

Inside RabbitMQ

- Also tested indirectly through RabbitMQ
- Maturing consensus implementations is hard
 - Time
 - Testing
 - Application
- Battle testing inside a widely used open source message broker
 - 

Other uses

- Mnevis
 - An experimental replication / transaction layer for mnesia
 - <https://github.com/rabbitmq/mnevis>
 - Implements the mnesia activity API
 - Breaks some of the rules for state machine implementation
 - We're working on verifying soundness of this approach

Ra status

- Current version: v0.9.0
- API settled
 - Major changes are unlikely before 1.0
- Semantic versioning applied to:
 - ra module
 - ra_machine behaviour
 - On-disk data formats
 - Post 1.0
- <https://github.com/rabbitmq/ra>

Ra future

- Multiple WALs
 - Configured to use multiple disk volumes
- Disk-based and mutable state machines
 - Like Mnevis
 - ETS based state machines?
- First class sessions
 - Easier linearizability
- Optimisation
 - Profiling
- More idiomatic elixir API?
 - What would it look like?

Thank you!