Make a Distributed Toolbox with Ra

a Raft implementation

By Team RabbitMQ

Karl Nilsson

Son of nil

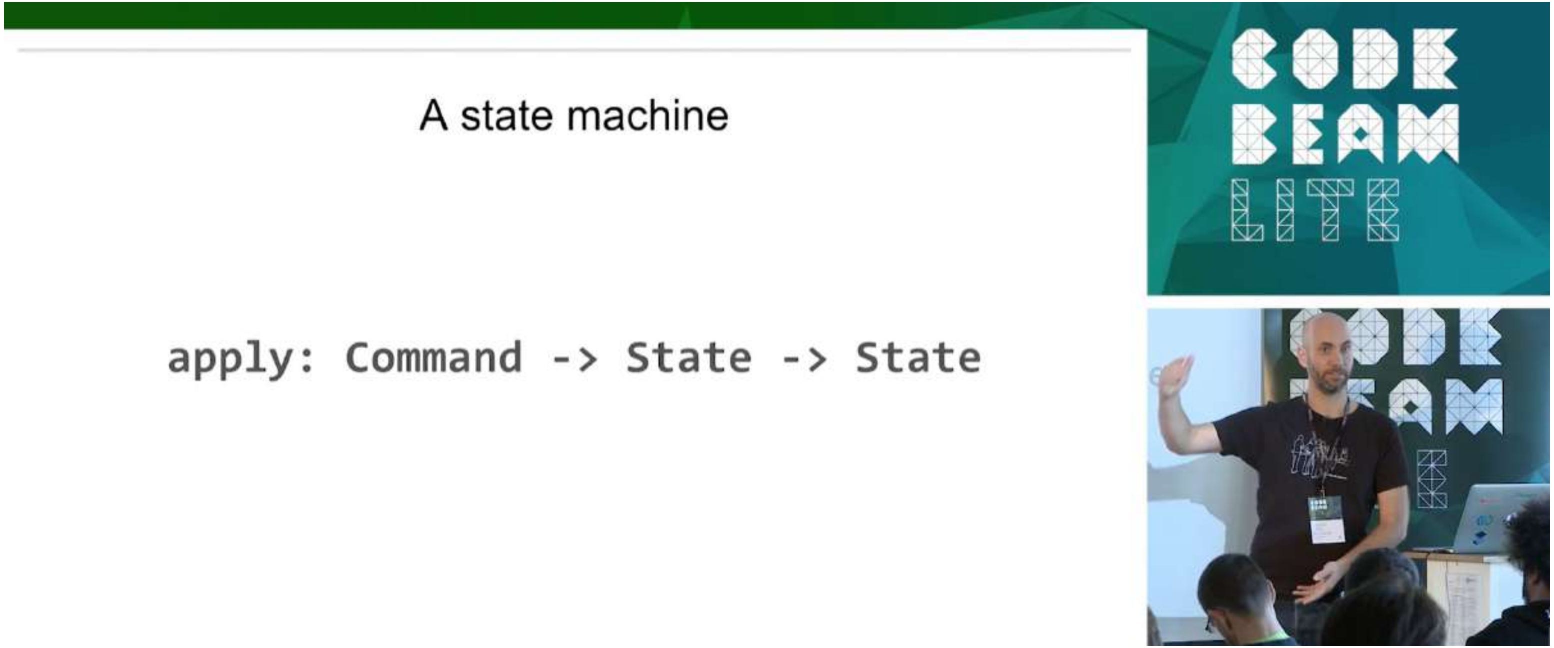
- Past:
  - .NET (C# / F#),
  - Distsys
- Relevant: Fez
  - F# to core erlang compiler
  - https://github.com/kjnilsson/fez
- t: @kjnilsson

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

CodeBEAM Lite Berlin



https://www.youtube.com/watch?v=7NNjjTrBZtw

Ra
Raft
RabbitMQ

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

Agree on ~~a value~~ *an ordered log of commands* in a cluster of processes

A State Machine

```
apply: Command -> State -> State
```

RA Status

https://github.com/rabbitmq/ra

0.9.4 on hex.pm

`ra` module API and `ra_machine` stable

Inside RabbitMQ

- Included in RabbitMQ 3.8
  - Quorum Queue feature
- Maturing consensus implementations is hard
  - Time
  - Testing
  - Application
- Battle testing inside a widely used open source message broker
  - 👌👊💪💗😄🙌👻👽👍💣

Just enough Raft

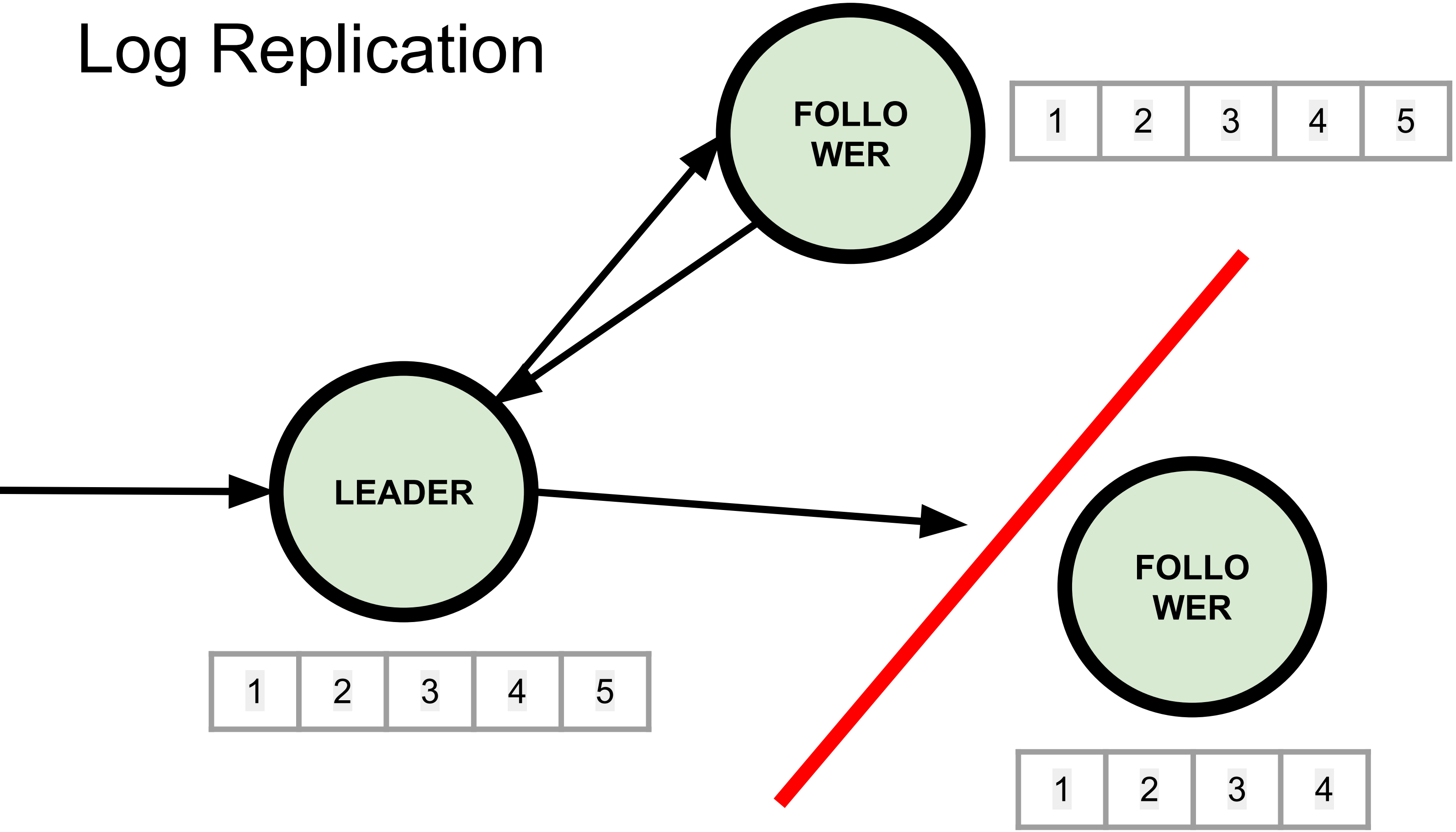
In two minutes

Raft is a consensus algorithm

Anatomy of a Raft Member [condensed]

- LOG
  - Indexed log of entries
- MEMBERS
  - The cluster configuration
- APPLY FUN
  - Pluggable state machine transition logic
- STATE
  - The current state of the state machine
- COMMIT INDEX
  - The index into the log which the server knows has “achieved consensus”
  - The state machine can be run up to this index

Log Replication



Every server in a Raft cluster will end up calculating exactly the same state

Ra Provides

- Linearizable storage
  - Ordered set of invocations
  - The foundation for many distributed “tools” (lock servers etc)
- Replication / Persistence
  - Data safety
  - Leader / follower
  - Recoverable State Machine
- Fault tolerance / High availability
  - follower can crash without affecting availability, to a point
  - Leader election
- Dynamic member changes
- Raft as a library

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: Implement `ra_machine` behaviour

Implement the `ra_machine` behaviour (2 required callbacks)

- `init/1`
  - Create the initial state of the state machine
- `apply/3`
  - Apply a command to the state machine and return the new state
  - Must be deterministic
  - No side effects inside `apply/3`! (!, exceptions, ets/dets operations)

Start a Ra cluster

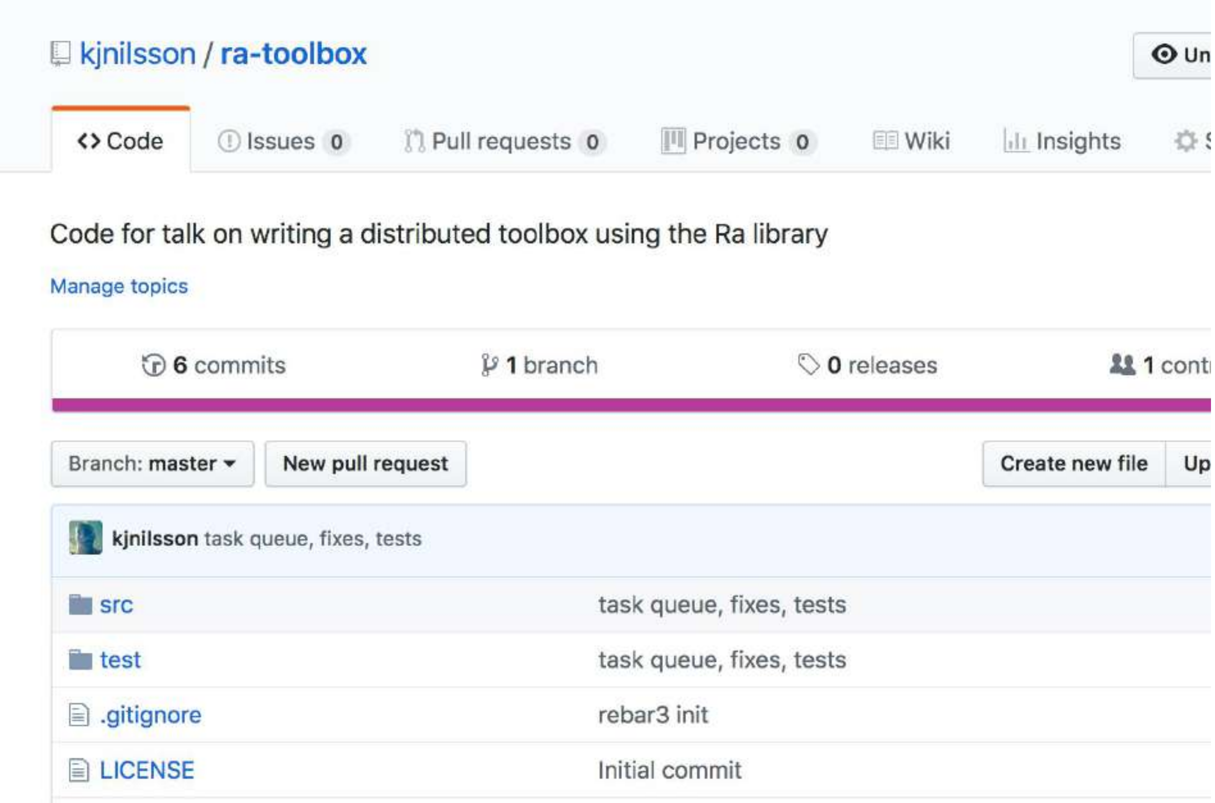
Start a cluster of Ra servers

- `ra:start_cluster/3`
- Ra servers are always named and referred to by their `{Name, Node}`.

Using Ra: read and write to state machine

- `ra:process_command/2`
  - Synchronously process a command
- `ra:pipeline_command/2|3|4`
  - Async processing. With or without success notification
- `ra:consistent_query/2`
  - Run a query over the state machine state
  - Requires consensus
- `ra:local_query/2`
  - Query the local state of the server being addressed
  - Can return stale results

https://github.com/kjnilsson/ra-toolbox



Key-Value Store

KV Store

```erlang
%% ra_machine implementation

init(_Config) -> #{}.

apply(_Meta, {put, Key, Value}, State) ->
    {maps:put(Key, Value, State), ok};
apply(_Meta, {delete, Key}, State) ->
    {maps:remove(Key, State), ok}.

```

KV Store - Client API

```erlang
%% Client api

put(ServerId, Key, Value) ->
    {ok, Result, _Leader} = ra:process_command(ServerId, {put, Key, Value}),
    Result.

delete(ServerId, Key) ->
    {ok, Result, _Leader} = ra:process_command(ServerId, {delete, Key}),
    Result.

get(ServerId, Key) ->
    QueryFun = fun(State)
                    when is_map_key(Key, State) ->
                      {ok, maps:get(Key, State)};
                  (_State) ->
                      {error, key_not_found}
               end,
    {ok, Result, _} = ra:consistent_query(ServerId, QueryFun),
    Result.
```

Group Membership

Group Membership

- Join / leave named groups
- Crashed processes should automatically be removed
  - monitor effect
- Erlang nodes can come and go but are assumed to come back at some point.

Join / Leave

```erlang
-type key() :: term().
-type group() :: #{pid() => ok}.
-type state() :: #{key() => group()}.

-spec init(map()) -> state().
init(_Config) -> #{}.

apply(_Meta, {join, GroupKey, Pid}, State0) ->
    State = maps:update_with(GroupKey,
                             fun(Group) -> Group#{Pid => ok}  end,
                             #{Pid => ok}, State0),
    Effect = {monitor, process, Pid},
    {State, ok, Effect};
apply(_Meta, {leave, GroupKey, Pid}, State0) ->
    case maps:take(GroupKey, State0) of
        error ->
            {State0, ok};
        {Group0, State} ->
            Group = maps:remove(Pid, Group0),
            {State#{GroupKey => Group}, ok}
    end;
```

What are “Ra Effects”?

Describe side-effects as data

Separate the state machine logic from side effects

Only the leader action the the effects, followers (mostly) throw them away

Ra Effects

- `{monitor, process | node, PidOrNode}`
  - Ask the leader to monitor a process or node
- `{send_msg, Pid, Msg :: term()}`
  - sends a message to a pid
- `{timer, Term, non_neg_integer() | infinity}`
  - Leader commits a timer message
- `{mod_call, Module, Function, Args}`
- https://github.com/rabbitmq/ra/blob/master/docs/internals/INTERNALS.md#effects

Monitor Effect

- The leader will append an entry to the log when a DOWN notification is received
- `{down, Pid, Info}`
  - Info can be the exit reason of the process, noproc or....

`noconnection`

Groups: failure handling

```erlang
    end;
apply(_Meta, {down, Pid, noconnection}, State) ->
    Effect = {monitor, node, node(Pid)},
    {State, ok, Effect};
apply(_Meta, {down, Pid, _Info}, State0) ->
    State = maps:map(fun(_, Group) ->
                            maps:remove(Pid, Group)
                    end, State0),
    {State, ok};
apply(_Meta, {nodeup, Node}, State) ->
    Effects = [{monitor, process, Pid} || Pid <- all_pids(State),
                                          node(Pid) == Node],
    {State, ok, Effects};
apply(_Meta, {nodedown, _Node}, State) ->
    {State, ok}.
```

What if the leader changes?

```
state_enter(leader, State) ->
    %% re-request monitors for all known pids
    [{monitor, process, Pid} || Pid <- all_pids(State)];
state_enter(_, _) ->
    [].
```

Locks

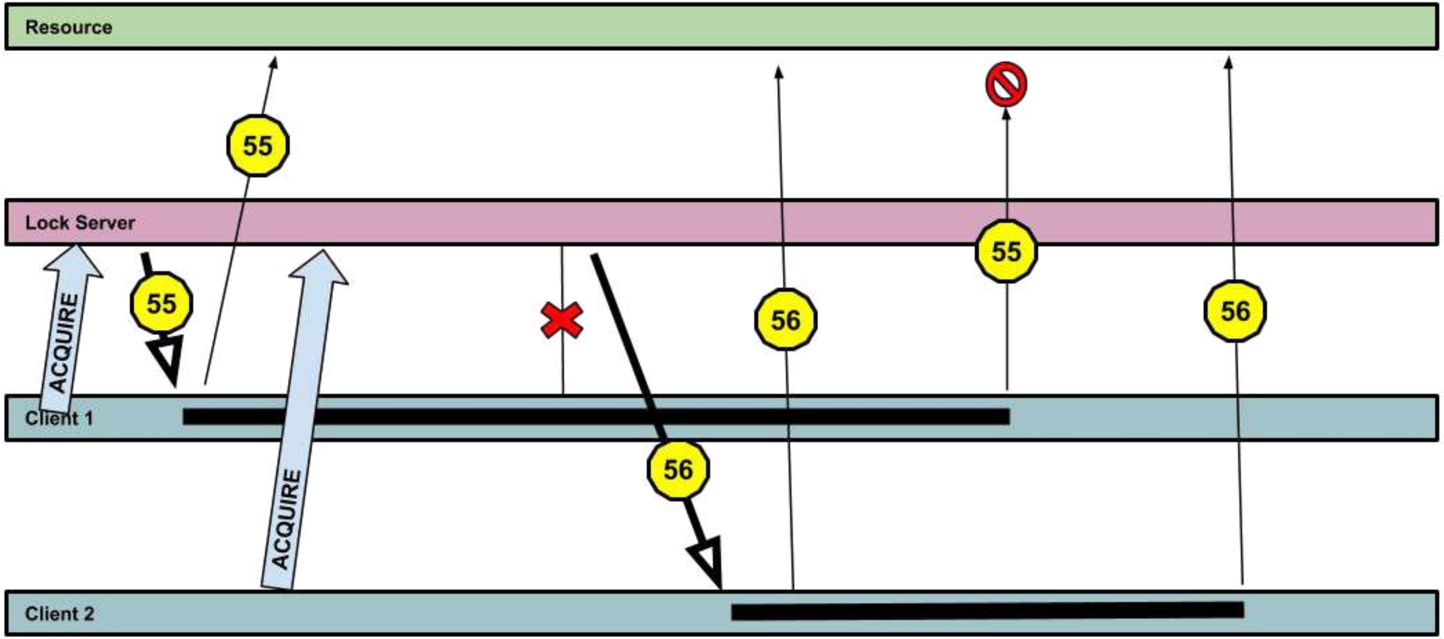
Locks / leader election

Coarse grained locks as could be used for holding leadership

1. Fencing token
2. Session based timeouts
   a. Distributed erlang “session”
      i. monitors
   b. Explicit sessions
3. Follow the leader
   a. Periodic heartbeats to leader to ensure leader is still “active”
   b. Raft allows multiple concurrent leaders (although only one can actually make progress)
4. Ra state machine
   a. Not always easy due to idempotency requirements

Fencing token

- Provides a unique monotonic value (token) with the lock
- The resource will reject any requests from clients with a token lower than the highest seen
- Use the Raft index as fencing token
- Pushes a lot of responsibility onto the resource
- Not all resources may be able to evaluate and maintain fencing token state

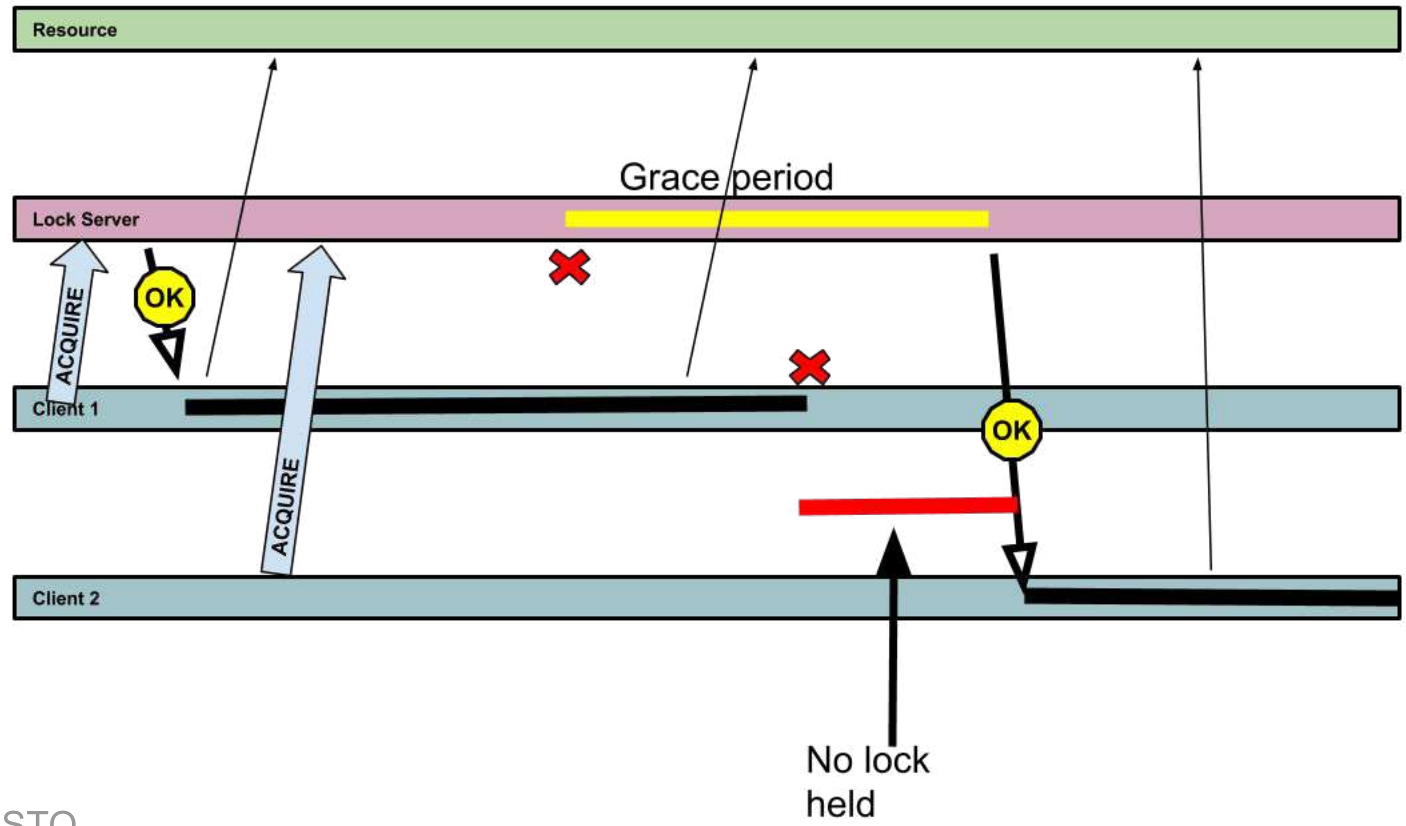

Resource
55
Lock Server
55
55
ACQUIRE
56
56
56
Client 1
56
ACQUIRE
Client 2

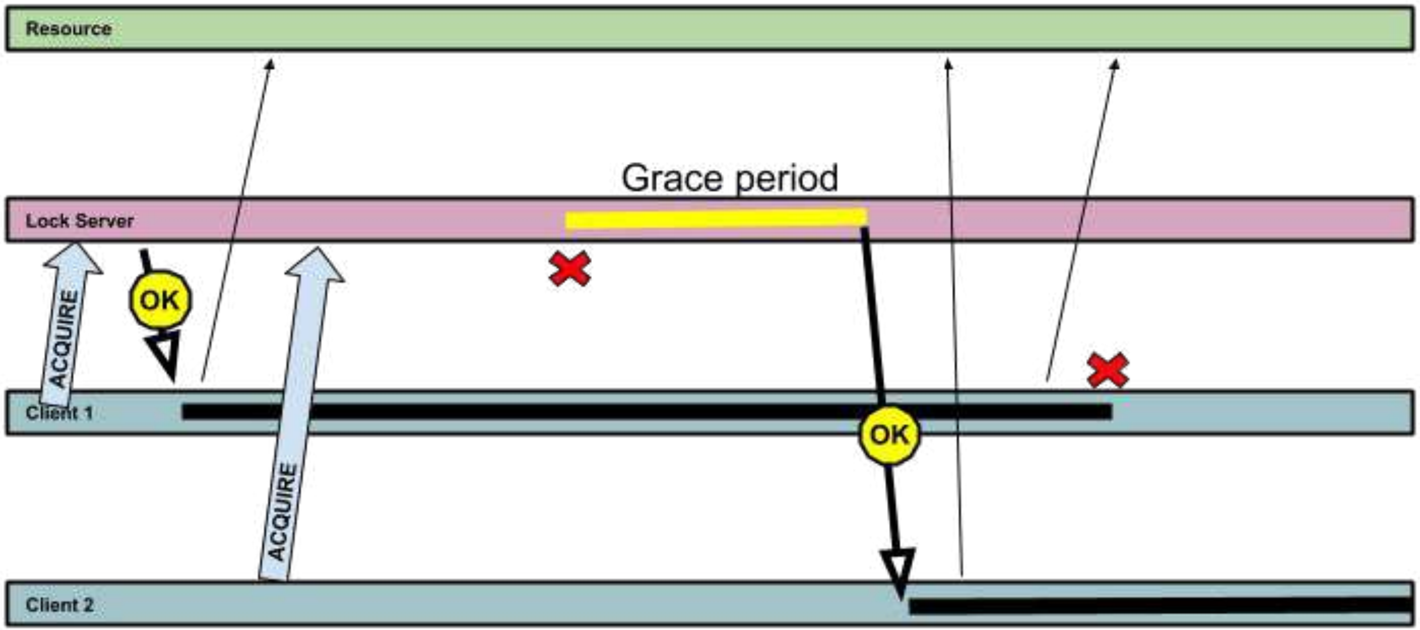
```
-record(lock, {holder :: undefined | pid(),
               waiting = [] :: [pid()]}).
-type state() :: #{term() => #lock{}}.

-spec init(_) -> state().
init(_Config) -> #{}.

-spec apply(meta(), cmd(), state()) ->
    {state(), ok | queued | {acquired, non_neg_integer()}} |
    {state(), ok | queued | {acquired, non_neg_integer()}, effect()}.
apply(#{index := Idx}, {acquire, Key, Pid}, State) ->
    handle_aquire(Key, Pid, Idx, State);
apply(#{index := Idx}, {release, Key, Pid}, State0) ->
    release_lock(Key, Pid, Idx, State0);
apply(#{index := Idx}, {down, Pid, _Info}, State) ->
    handle_pid_down(Pid, Idx, State).
```

Session based / timeouts

- Distributed erlang session / TCP “session”
- Uses time (!)
  - timer effect
- Downsides:
  - Concurrent resource access possible
  - Unavailability
  - Requires careful programming
- Upside:
  - Works with any resource

Time based approaches to locks can never be 100% correct / safe


Resource
Grace period
Lock Server
OK
ACQUIRE
ACQUIRE
Client 1
OK
Client 2
No lock held


Resource
Grace period
Lock Server
ACQUIRE
OK
ACQUIRE
Client 1
OK
Client 2

Follow the Leader

- Spawn a local companion process on becoming leader
  - state_enter(leader, ...
  - `mod_call` effect
  - monitor effect
- Combine with periodic consistent queries to ensure local leader is still the leader
  - If the query times out, exit process
- Grace period on becoming leader before processing starts
  - Needs to be longer than the query period

Ra State Machine

Not all systems can be practically written as a Ra state machine

Other tools

Task Queues

Barrier

2PC

It all looks a bit.. Similar. Can we generalize?

Toolbox or multi-tool?

ZooKeeper API

ZK / Chubby API overview

- Virtual filesystem
  - /node1/node2
- Sessions
- Watchers / ephemeral nodes
- Sequences
  - fencing tokens
- Simple primitives are used by complex clients to implement tools

Ra is designed to support many independent clusters

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

Thank you!