Implementation and Verification of a Consensus Protocol in Erlang
I

about your
Distributed Systems are HARD
Many ways to skin a
TRADITION

Just because you’ve always done it that way doesn’t mean it’s not incredibly stupid.
Asynchronous Replication

client → Master → Slave
Asynchronous Replication

client

Master

Slave

a
Asynchronous Replication

client

Master

Slave

ok
Asynchronous Replication

client

Master

Slave

a
Asynchronous Replication

client

Master

Slave

a

a
Asynchronous Replication

Client → Master → Slave

b
Asynchronous Replication

Client

Master

Slave
Asynchronous Replication

client

Master

ok

Slave

a
Asynchronous Replication
Consistent or Available
if (promote_secondary) {
    stderr(“possible data loss”);
} else {
    stderr(“system unavailable”);
}
Yep, you just traded safety for latency
Synchronous Replication

client → a → Master

Slave
Synchronous Replication

Client

Master

Slave
Synchronous Replication

client

Master

Slave

a
Synchronous Replication

client

Master

Slave

a
Synchronous Replication

client

Master

Slave

ok
Synchronous Replication

client

Master

Slave

ok
Synchronous Replication

client → b → Master

Slave

a

a
Synchronous Replication

- Client
- Master: a, b
- Slave: a
Synchronous Replication
Synchronous Replication

Client

Master

Slave

timeout
Safety vs. Liveness
Goal: Maintain Safety while tolerating failure
Quorums
Node Failures Allowed while Available

Cluster Size
Problems

• Who coordinates writes and reads?
• What interleavings are ‘safe’?
Consensus
RAFT
• Distributed State Machine Replication
• Designed to facilitate understanding
• John Ousterhout and Diego Ongaro
izability
• All nodes agree on an identical sequence of operations

• Monotonic ‘Term’ acts as a logical clock to prevent time from going backwards

• Operations are committed when written to a log on majority of nodes AND the term of the entry is the current term

• Once committed an operation cannot be removed from the log
### Replicated Log

<table>
<thead>
<tr>
<th>Index</th>
<th>Term</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>vl</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
<td>v6</td>
<td></td>
</tr>
</tbody>
</table>

- **Committed**
- **Uncommitted**
<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>v1</td>
<td>v2</td>
<td>v3</td>
<td>v4</td>
<td>v5</td>
</tr>
</tbody>
</table>

Replicated Log

- Committed
- Uncommitted

NEW Leadership™
Leader Election
• All nodes start in follower state

• After a random timeout, one becomes candidate

• Candidate increments term, requests a vote

• Followers vote for the candidate if the candidate log and term are up to date
Majority Rules
Log Replication
• Leader sends Append Entries calls to each follower

• If previous log entry and term agree with the follower log contents, follower replies with success

• Leader keeps track of follower log indexes, decrements index on failure and sends older data

• If Majority replies with success, leader commits entry, responds to client and tells followers the latest commit index on next heartbeat
Heartbeats

:=:=

Append Entries
• Prevent followers from becoming candidates unnecessarily

• Allow followers to detect failed leader or network partition and start a new election

• Leader replays log to followers who are behind due to either prior failure or netsplit
Rafter
• A Library for building strongly consistent distributed systems in Erlang

• Implements Raft in Erlang

• Isolates the application developer from the intricacies of consensus
Why Raft?

• I wanted to fully grok consensus
• Easier to understand than Paxos
• Every day someone tries to implement consensus in an ad-hoc manner
Why Erlang?

- Erlang is terrific for building reliable distributed systems
- I currently spend > 90% of my coding time in Erlang
- Consensus is NOT solved in Erlang
- mnesia, gen_leader, gproc don’t tolerate netsplits
Core Abstractions
Peers

- Each peer is made up of 2 supervised processes
  - A gen_fsm implements the raft protocol
  - A gen_server wraps the persistent log
- An API module hides the implementation
-include_lib("rafter/lib/rafter_opts.hrl").

start_node() ->
    Name = peer1,
    Me = {Name, node()},
    Opts = #rafter_opts{state_machine=rafter_backend_ets,
                        logdir="./data"},
    rafter:start_node(Me, Opts).

set_config(Peer, NewServers) ->
    rafter:set_config(Peer, NewServers).

put(Peer, Table, Key, Value) ->
    rafter:op(Peer, {put, Table, Key, Value}).

get(Peer, Table, Key) ->
    rafter:read_op(Peer, {get, Table, Key}).
Replicated Log

• API operates on Log Entries
• Log Entries contain commands
• Commands transparent to rafter
• Cmds encoded with term_to_binary/1
File Header Format
-----------------------
<<Version:8>>

Entry Format
--------------
<<Sha1:20/binary, Type:8, Term:64, Index:64, Size:32, Cmd/binary>>

Entry Trailer Format
----------------------
<<Crc:32, ConfigStart:64, EntryStart:64, ?MAGIC/64>>
Backend State Machine

- OTP behaviour
- Operates on commands via callbacks from consensus fsm
- Callbacks run on each node when commands are committed or read quorum achieved
-module(rafter_backend).

-export([behaviour_info/1]).

behaviour_info(callbacks) ->
    [{init, 0}, {read, 1}, {write, 1}];

behaviour_info(_) ->
    undefined.
read({get, Table, Key}) ->
  try
    case ets:lookup(Table, Key) of
      [{Key, Value}] -> {ok, Value};
      [] -> {ok, not_found}
    end
  catch _:E ->
    {error, E}
  end;

write({put, Table, Key, Value}) ->
  try
    ets:insert(Table, {Key, Value}),
    {ok, Value}
  catch _:E ->
    {error, E}
  end;}
Consensus Module

- Implements Raft protocol in gen_fsm
- 3 states - follower, candidate, leader
- Logs persistent data via rafter_log gen_server
- Pure functions handling dynamic reconfiguration and quorums abstracted out to rafter_config
%% API
-export([start/0, stop/1, start/1, start_link/3,
leader/1, op/2, set_config/2,
send/2, send_sync/2, get_state/1]).

%% States
-export([follower/2, follower/3,
candidate/2, candidate/3,
leader/2, leader/3]).
%% Election timeout has expired. Go to candidate state iff we are a voter.
follower(timeout, #state{config=Config, me=Me}=State) ->
case rafter_config:has_vote(Me, Config) of
  false ->
    Duration = election_timeout(),
    {next_state, follower, State, Duration};
  true ->
    {next_state, candidate, State, 0}
end;

follower({read_op, _}, _From, #state{leader=Leader}=State) ->
  Reply = {error, {redirect, Leader}},
  {reply, Reply, follower, State, ?timeout()};
%% We are out of date. Go back to follower state.

candidate(#vote{term=VoteTerm, success=false}, #state{term=Term}=State)
  when VoteTerm > Term ->
    NewState = step_down(VoteTerm, State),
    {next_state, follower, NewState, NewState#state.timer_duration};

%% Sweet, someone likes us! Do we have enough votes to get elected?

candidate(#vote{success=true, from=From}, #state{responses=Responses, me=Me, config=Config}=State) ->
  NewResponses = dict:store(From, true, Responses),
  case rafter_config:quorum(Me, Config, NewResponses) of
    true ->
      NewState = become_leader(State),
      {next_state, leader, NewState, 0};
    false ->
      NewState = State#state{responses=NewResponses},
      {next_state, candidate, NewState, ?timeout()}
  end.
leader(timeout, State) ->
  Duration = heartbeat_timeout(),
NewState = State#{timer_start=os:timestamp(),
  timer_duration=Duration},
send_append_entries(State),
{next_state, leader, NewState, Duration};

leader({op, {Id, Command }}, From, #state{term=Term}=State) ->
  Entry = #rafter_entry{type=op, term=Term, cmd=Command},
NewState = append(Id, From, Entry, State, leader),
{next_state, leader, NewState, ?timeout()}. 

Implementation Tradeoffs

- Distributed Erlang
- Single FSM for the consensus algorithm
- Separating read path from write path
- Rolling my own log file format
What isn’t done?

• Handling of exactly-once semantics for non-idempotent commands
• Log compaction
• A nice DB built on top of rafter
• More tests, More documentation
• Performance
Testing
Property Based Testing

-include_lib("eqc/include/eqc.hrl").

prop_reverse() ->
  ?FORALL(L, list(int())),
  L == lists:reverse(lists:reverse(L)).

eqc:quickcheck(eqc:numtests(1000, prop_reverse())).
Stateful Property Tests

- eqc_statem behaviour
- Create a model of what you're testing
- Verify that model
eqc_statem callbacks

- initial_state/0
- precondition/2
- command/1
- postcondition/3
- next_state/3
- invariant/1 (optional)
command(_State) ->
  frequency([  
    {10, {call, rafter_backend_ets, write, [{new, table_gen()}]}},
    {3, {call, rafter_backend_ets, write, [{delete, table_gen()}]}},
    {100, {call, rafter_backend_ets, write, [{delete, table_gen()}, key_gen()]似},
    {200, {call, rafter_backend_ets, read, [{get, table_gen()}, key_gen()]似},
    {200, {call, rafter_backend_ets, write,  
        [{put, table_gen()}, key_gen(), value_gen()]}},
    {20, {call, rafter_backend_ets, read, [list_tables]}},
    {20, {call, rafter_backend_ets, read, [{list_keys, table_gen()}]似}}]).
next_state(#state{tables=Tables}=S, _Result, 
   {call, rafter_backend_ets, write, [{new, Table}]})) ->
   S#state{tables={call, sets, add_element, [Table, Tables]}};

postcondition(#state{}, 
   {call, rafter_backend_ets, write, [{new, Table}]}, 
   {ok, Table}) ->
   true;

postcondition(#state{tables=Tables}, 
   {call, rafter_backend_ets, write, [{new, Table}]}, 
   {error, badarg}) ->
   sets:is_element(Table, Tables);
\textbf{invariant}(\textit{State}) \rightarrow
\textit{tables\_are\_listed\_in\_ets\_tables\_table}(\textit{State}) \textbf{andalso}
\textit{tables\_exist}(\textit{State}) \textbf{andalso}
\textit{data\_is\_correct}(\textit{State}).

\textbf{tables\_exist}(\#\text{state}{\text{tables} = \text{Tables}}) \rightarrow
\text{EtsTables} = \text{sets:from\_list}(\text{ets:all}()),
\text{sets:is\_subset}(\text{Tables}, \text{EtsTables}).

\textbf{data\_is\_correct}(\#\text{state}{\text{data} = \text{Data}}) \rightarrow
\text{lists:all}(\text{fun}(\{{\text{Table, Key}, \text{Value}}\}) \rightarrow
\quad [\{\text{Key, Value}\}] =:= \text{ets:lookup}(\text{Table, Key})
\quad \text{end, Data}).
An Actual Bug

%% This should only happen if two machines are configured differently during
%% initial configuration such that one configuration includes both proposed leaders
%% and the other only itself. Additionally, there is not a quorum of either
%% configuration's servers running.
%%
%% (i.e. rafter:set_config(b, [k, b, j]), rafter:set_config(d, [i,k,b,d,o]).
%%   when only b and d are running.)
%%
%% candidate(#vote{term=VoteTerm, success=false},
%%   #state{term=Term, init_config=[_Id, From]}=State) when VoteTerm > Term ->
gen_fsm:reply(From, {error, invalid_initial_config}),
State2 = State#state{init_config=undefined, config=#config{state=blank}},
NewState = step_down(VoteTerm, State2),
{next_state, follower, NewState, NewState#state.timer_duration};
Model Checking

=/=

Proof of correctness
Other Test Tools

- Pulse - http://quviq.com
- Concuerror - http://concuerror.com/
- PropEr - http://proper.softlab.ntua.gr/
That's all Folks!