Riak PG
Distributed Process Groups on Dynamo-style Distributed Storage

Christopher Meiklejohn
Basho Technologies, Inc.
Erlang Workshop ’13
The Goal
The Goal

Build a highly-available, fault-tolerant registry.
The Goal

Understand the tradeoffs.
The Problem
The Problem

Highly-available distributed process groups.
The Problem

Examples: pg2, gproc
The pg2 Problem
Reappearing groups; synchronous global writes.
The gproc Problem

Election deadlocks; conflicts; dynamic clusters.
Regarding the gen_leader fork above, it does have basic support for netsplit situations merged from Andrew Thompson's (of Basho) fork. There has not been any rigorous model checking that I'm aware of. My own testing has shown that it basically works, with 2 caveats (so far): 1) the process is periodically unavailable for a few seconds at a time doing election related things (even on stable networks), so frequent requests like "who is the leader" can sometimes take a while, 2) if 2 gen_leader instances are started during a netsplit, they will eventually deadlock. I'm hoping to take a look at these issues over the next few weeks.

--Garret Smith
A more serious problem with using gen_leader is that it requires advance knowledge of all candidate nodes.

“Extended Process Registry for Erlang”, Ulf T. Wiger, Erlang Workshop ’07
Perhaps even more serious is gen_leader’s lack of support for dynamically reconfigured networks, and for de-conflicting the states of two leaders (which is presumably the most difficult part of adding nodes on the fly).

“Extended Process Registry for Erlang”, Ulf T. Wiger, Erlang Workshop ’07
The Challenges
The 3 Challenges
Dynamic **addition** and **removal** of nodes.
The 3 Challenges
Coordination of state mutation.
The 3 Challenges
Resolution of conflicting values.
Riak PG
Dynamic membership through virtual nodes.
Riak PG

Replicated state; quorum reads and writes.
Riak PG

Conflict-free resolution with CRDTs.
Eventually consistent; harvest vs. yield tradeoff.
Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

“Eventual Consistency”, Wikipedia
Our approaches tolerate partial failures by emphasizing simple composition mechanisms that promote fault containment, and by translating possible partial failure modes into engineering mechanisms that provide smoothly degrading functionality rather than lack of availability of the service as a whole.

“Harvest, Yield, and Scalable Tolerant Systems”, Fox and Brewer
The Requirements
The Requirements
Structured names.
Multiple non-unique names per process.

The Requirements
The Requirements
Dynamic cluster membership.
The Requirements
Partition tolerance and conflict resolution.
The Applications
The Applications

Service lookup pattern; publish and subscribe.
The Applications
Trade consistency for availability.
The Background
Riak Core; CRDTs
Riak Core
Riak Core
Erlang implementation of Dynamo.
Riak Core
Consistent hashing.
Riak Core
Hash-space partitioning.
Riak Core
Dynamic membership.
Riak Core
Replication factor.
Observed-Removed Set
Observed-Removed Set
CvRDT; bounded join-semilattice.
Observed-Removed Set
Set; with merge function computing a LUB.
Observed-Removed Set

Two $G$-Sets; preserves monotonicity.
|\[ [{1, a}], [] \] | \[ [{1, a}], [] \] |
|\[ [{1, a}], [{1, a}] \] | \[ [{1, a}], [{1, a}] \] |
The Implementation
The Implementation
Same as pg2; create, join, leave, and members.
The Implementation
Extended with connected members.
The Implementation

Membership vnode stores registrations.
The Implementation
Conflict-free resolution with OR-set.
The Implementation
Process pruning; lack of monitors.
The Implementation

Code examples.
The Virtual Node
% @doc Respond to a join request.
handle_command({join, {ReqId, _}, Group, Pid},
    _Sender,
    #state{groups=Groups0, partition=Partition}=State) ->
  %% Find existing list of Pids, and add object to it.
Pids0 = pids(Groups0, Group, riak_dt_vvorset:new()),
Pids = riak_dt_vvorset:update({add, Pid}, Partition, Pids0),

  %% Store back into the dict.
Groups = dict:store(Group, Pids, Groups0),

  %% Return updated groups.
{reply, {ok, ReqId}, State#state{groups=Groups}};

% @doc Return pids from the dict.
-spec pids(dict(), atom(), term()) -> term().
pids(Groups, Group, Default) ->
  case dict:find(Group, Groups) of
    {ok, Object} ->
      Object;
    _ ->
      Default
  end.
handle_command({leave, {ReqId, _}, Group, Pid}, _Sender, #state{groups=Groups0, partition=Partition}=State) ->
  Pids0 = pids(Groups0, Group, riak_dt_vvorset:new()),
  Pids = riak_dt_vvorset:update([{remove, Pid}, Partition, Pids0}),
  Groups = dict:store(Group, Pids, Groups0),
  {reply, {ok, ReqId}, State#state{groups=Groups}};

pids(Groups, Group, Default) ->
  case dict:find(Group, Groups) of
    {ok, Object} -> Object;
    _ -> Default
  end.
The **Write Coordinator**
%% @doc Execute the request.
execute(timeout, #state{preflist=Preflist,
    req_id=ReqId,
    coordinator=Coordinator,
    group=Group,
    pid=Pid}=State) ->
    riak_pg_memberships_vnode:join(Preflist, {ReqId, Coordinator}, Group, Pid),
    {next_state, waiting, State}.

%% @doc Attempt to write to every single node responsible for this group.
waiting([{ok, ReqId}],
    #state{responses=Responses0, from=From}=State0) ->
    Responses = Responses0 + 1,
    State = State0#state{responses=Responses},
    case Responses =:= ?W of
      true ->
        From ! {ReqId, ok},
        {stop, normal, State};
      false ->
        {next_state, waiting, State}
    end.
The **Read Coordinator**
%% @doc Pull a unique list of memberships from replicas, and
%%      relay the message to it.

waiting({ok, _ReqId, IndexNode, Reply},
    #state{from=From,
        req_id=ReqId,
        num_responses=NumResponses0,
        replies=Replies0}=State0) ->
  NumResponses = NumResponses0 + 1,
  Replies = [{IndexNode, Reply}|Replies0],
  State = State0#{
      num_responses = NumResponses,
      replies = Replies},

  case NumResponses =:= ?R of
      true ->
          Pids = riak_dt_vvorset:value(merge(Replies)),
          From ! {ReqId, ok, Pids},

          case NumResponses =:= ?N of
              true ->
                  {next_state, finalize, State, 0};
              false ->
                  {next_state, waiting_n, State}
          end;
      false ->
          {next_state, waiting, State}
  end.

  riak_pg/src/riak_pg_members_fsm.erl
%% @doc Perform merge of replicas.
merge(Replies) ->
    lists:foldl(fun({_, Pids}, Acc) ->
        riak_dt_vvorset:merge(Pids, Acc) end,
        riak_dt_vvorset:new(), Replies).
%% @doc Wait for the remainder of responses from replicas.

waiting_n({ok, _ReqId, IndexNode, Reply},
    state{num_responses=NumResponses0,
        replies=Replies0}=State0) ->
    NumResponses = NumResponses0 + 1,
    Replies = [{IndexNode, Reply}|Replies0],
    State = State0#state{num_responses=NumResponses, replies=Replies},

case NumResponses =:= ?N of
    true ->
        {next_state, finalize, State, 0};
    false ->
        {next_state, waiting_n, State}
end.

riak_pg/src/riak_pg_members_fsm.erl
%% @doc Perform read repair.
finalize(timeout, #state{replies=Replies}=State) ->
    Merged = merge(Replies),
    Pruned = prune(Merged),
    ok = repair(Replies, State#state{pids=Pruned}),
    {stop, normal, State}.

riak_pg/src/riak_pg_members_fsm.erl
%% @doc If the node is connected, and the process is not alive, prune it.
prune_pid(Pid) when is_pid(Pid) ->
    lists:member(node(Pid), nodes()) andalso
    (is_process_alive(node(Pid), Pid) =:= false).

%% @doc Remote call to determine if process is alive or not; assume if the node fails communication it is, since we have no proof it is not.
is_process_alive(Node, Pid) ->
case rpc:call(Node, erlang, is_process_alive, [Pid]) of
    {badrpc, _} -> true;
    Value -> Value
end.

%% @doc Based on connected nodes, prune out processes that no longer exist.
prune(Set) ->
Pids0 = riak_dt_vvorset:value(Set),
lists:foldl(fun(Pid, Pids) ->
    case prune_pid(Pid) of
    true ->
        riak_dt_vvorset:update([{remove, Pid}],
            none, Pids);
    false -> Pids
    end
    end, Set, Pids0).
The Evaluation
The Evaluation
Partitions heal without conflicts.
The Related Work

Howl; CloudI Process Groups; Riak Pipe
The Future Work
The Future Work
CRDT garbage collection.
The Future Work
Active anti-entropy mechanism.
The Conclusion
Thanks! Questions?

http://github.com/cmeiklejohn/riak_pg