MODELING EVENTUAL CONSISTENCY DATABASES WITH QUICKCHECK

Jon Meredith Basho Technologies

WHERE IS THIS FROM?

- Quickcheck Guru in a NoSQL world?
- A week of consulting with a real guru, John Hughes.
- With acolytes Scott Fritchie, Dave "Dizzy" Smith and me.

basho

riak

Thanks for coming.
Today I'm going to talk about our experience using QuickCheck to test Riak.
Back in January we spent a week working on how

WHO AM I?

Jon Meredith

- Learned Erlang to work on a Dynamo clone 3 years ago.
- Learned QuickCheck soon after.
- Now working on Riak at Basho Technologies.

GOAL

Test Riak works correctly during failure.

Failure: Node death

Also: Network partitions

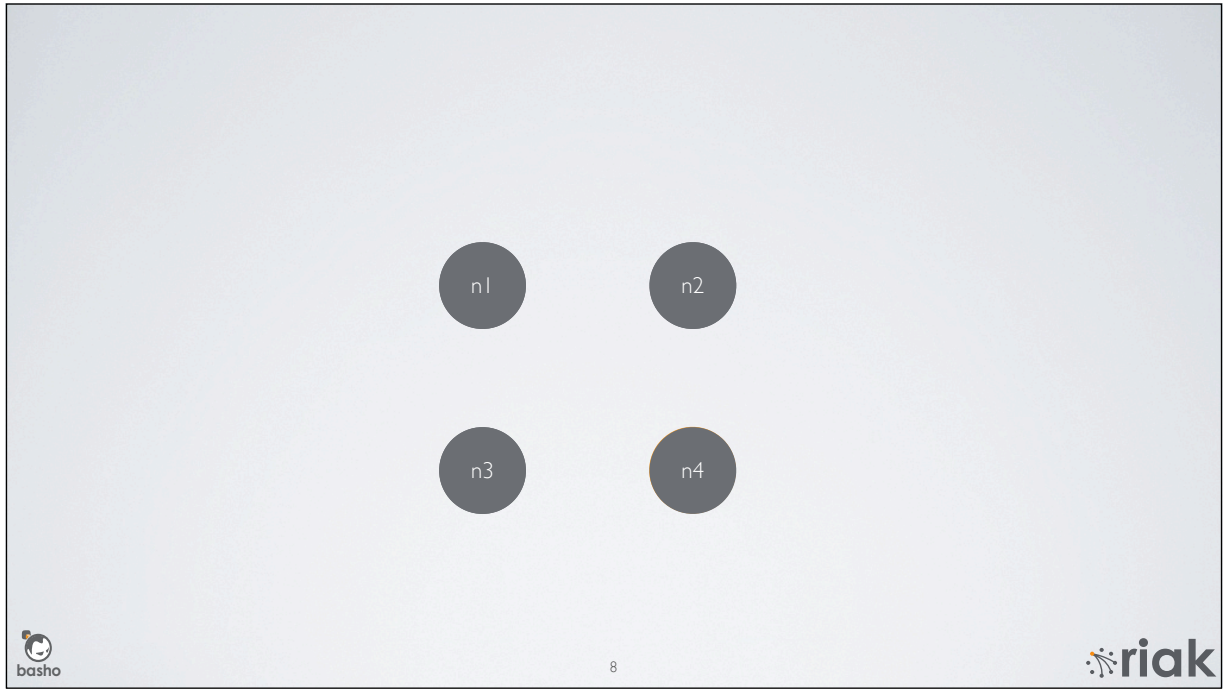
Riak is
a scalable, highly-available, networked
key/value store.

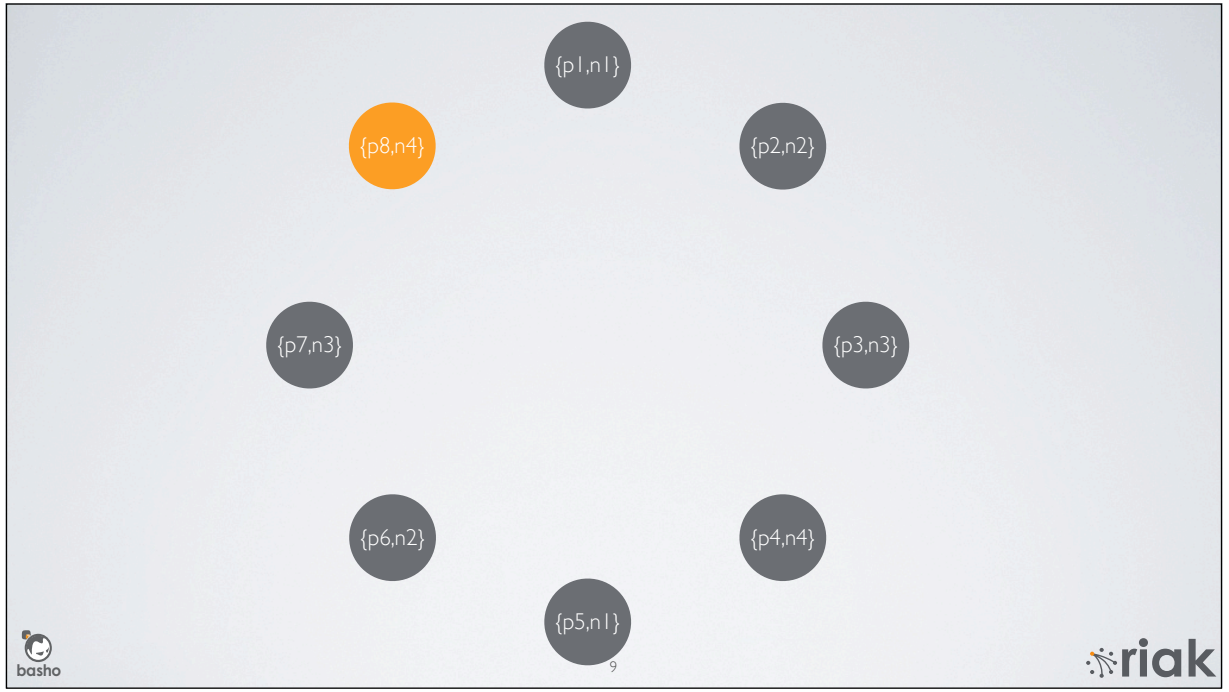
JUST ENOUGH RIAK

- Inspired by Amazon's Dynamo.
- Key/Value store + Metadata
- Riak chooses Availability and Partition tolerance over Consistency.
- Instead provides eventual consistency.

DISTRIBUTED BY DESIGN

- Designed to run on a cluster of nodes.
- Keys are hashed onto a 160-bit hash ring.
- Ring is divided into Q equal partitions.
- Each partition is owned by a vnode.
- Physical nodes run multiple vnodes.

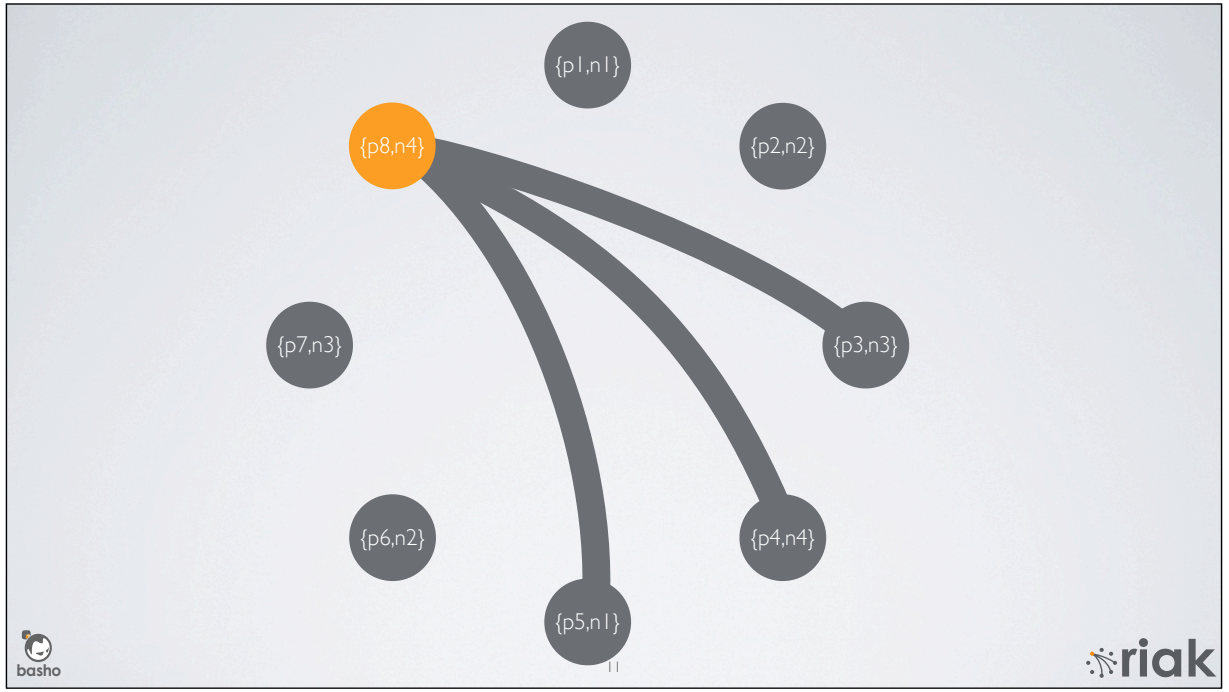

n1
n2
n3
n4

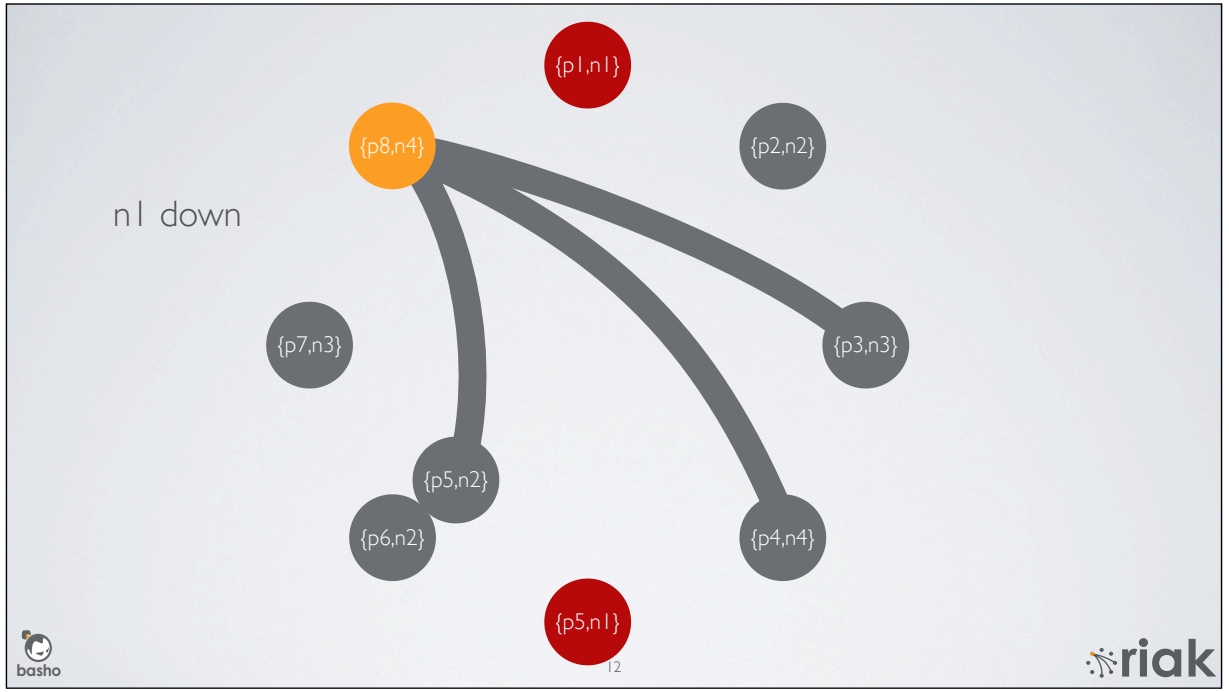

{p1,n1}
{p8,n4}
{p2,n2}
{p7,n3}
{p3,n3}
{p6,n2}
{p4,n4}
{p5,n1}
basho
9
riak

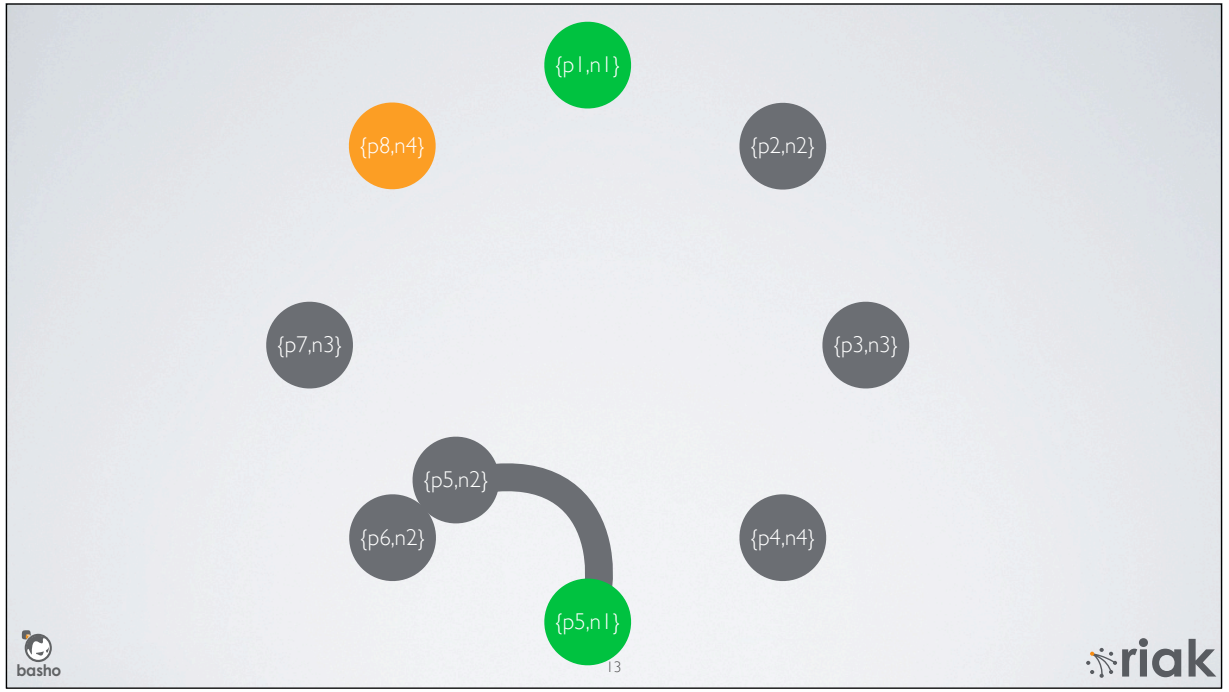
JUST ENOUGH DYNAMO

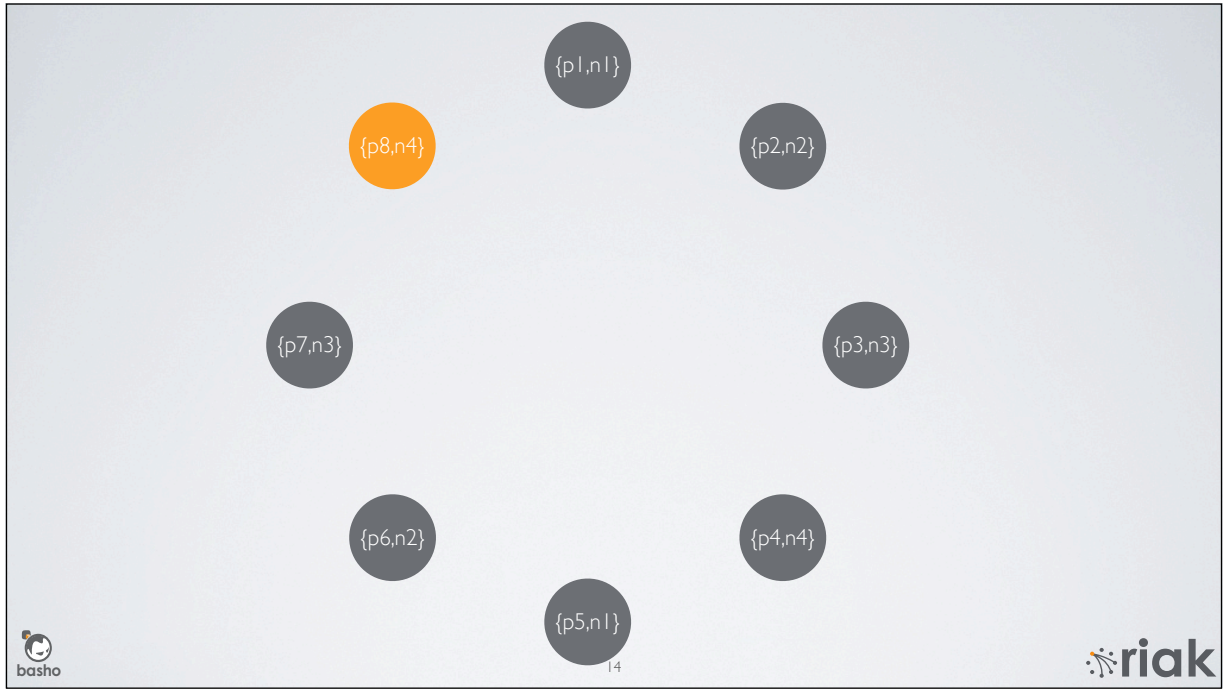
- Each object is replicated to N vnodes.
- Vnodes chosen by hashing key and picking next N partitions - 'preference list'.
- To write, send copies to N vnodes in the cluster.
  - At least 'W' must succeed.
- To read, request copied from the same N vnodes
  - At least 'R' must succeed.

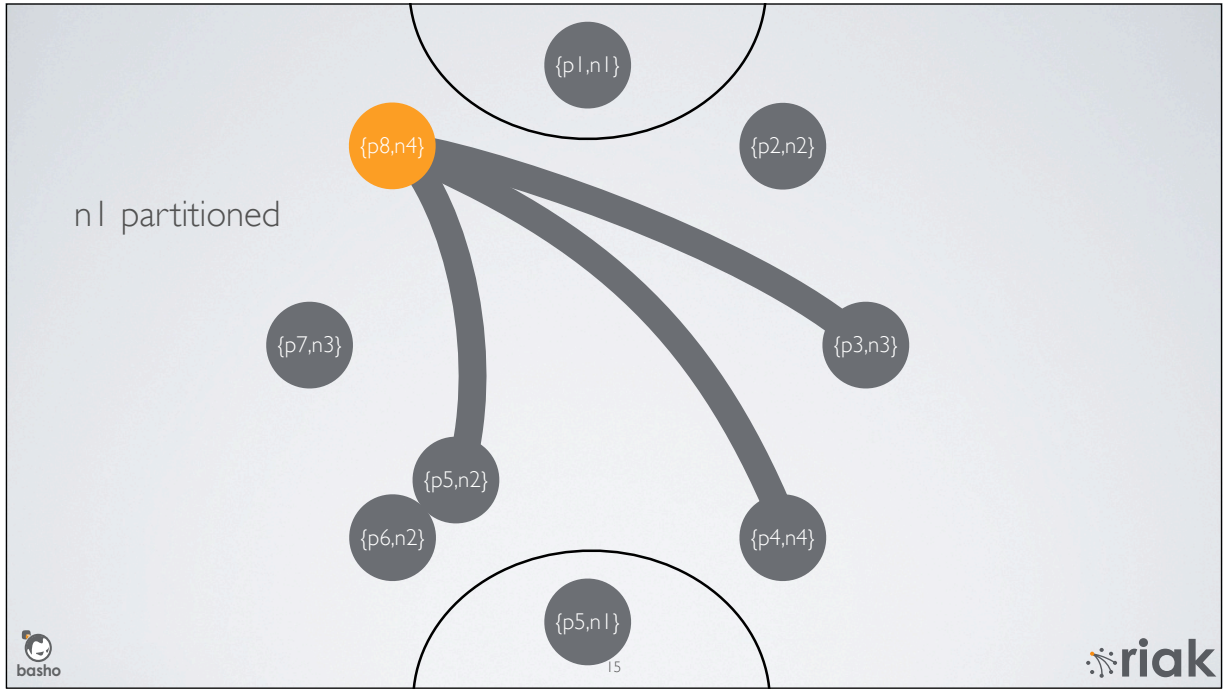
riak


{p1,n1}
{p8,n4}
{p2,n2}
{p7,n3}
{p3,n3}
{p6,n2}
{p4,n4}
{p5,n1}
basho
riak

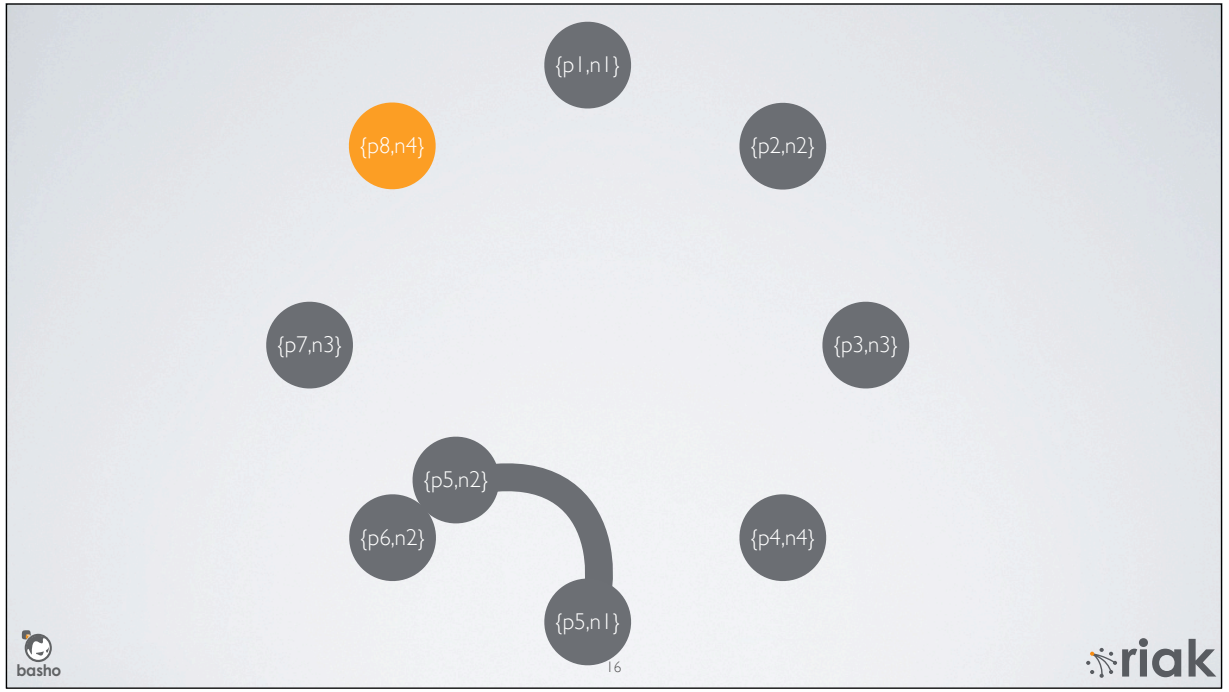

{p1,n1}
{p8,n4}
{p2,n2}
n1 down
{p7,n3}
{p3,n3}
{p5,n2}
{p6,n2}
{p4,n4}
{p5,n1}
basho
12
riak


{p1,n1}
{p8,n4}
{p2,n2}
{p7,n3}
{p3,n3}
{p5,n2}
{p6,n2}
{p4,n4}
{p5,n1}
basho
13
riak


{p1,n1}
{p8,n4}
{p2,n2}
{p7,n3}
{p3,n3}
{p6,n2}
{p4,n4}
{p5,n1}
basho
14
riak



Same mechanism handles network partitions as failures.



Riak has great availability properties.
No locking consitency.
Possible for multiple values to be present per object.
Riak revisions objects so some cases can be handled automatically.
But if multiple writers modify the same object there will be conflicts.

SIMPLE INTEGRATION TEST

- Test get/puts against a cluster of nodes.
- Model multiple clients.
- Stateful problem - eqc_statem to the rescue!

JUST ENOUGH QUICKCHECK

- Erlang QuickCheck (EQC) - Property based testing tool by Quviq.
- You write tests as properties of your system.
- Properties use EQC generators to create parameters.
- Run code on the parameters to verify the property is true.

SIMPLEST EXAMPLE

```
my_test() ->
    eqc:quickcheck(reverse_prop()).

reverse_prop() ->
    ?FORALL(L,
            list(int()),
            begin
                lists:reverse(lists:reverse(L)) == L
            end).
```

SHRINKING

```
bad_prop() -> ?FORALL(L, list(int()),
          begin
             lists:reverse(L) == L
          end).
1> listeg:bad_test().
Starting Quviq QuickCheck version 1.22.2
  (compiled at {{2011,1,13},{22,29,18}})
Licence for Basho reserved until {{2011,3,20},{17,7,46}}
...........Failed! After 13 tests.
[-3,4]
Shrinking...(3 times)
[0,1]
```

riak

JUST ENOUGH EQC_STATEM

- eqc_statem is a framework for testing stateful things
- You provide a command generator
- eqc_statem generates a series of commands
- You check they work
- ... and if they don't, EQC tries to find the minimum sequence for failure.

EQC_STATEM CALLBACKS

- initial_state() -- Initialize a state record
- command(State) -- Generator that returns an MFA
- next_state(State,Result,Cmd) -- Update state after a command
- precondition(S, Cmd) -- Check it is valid to enter a state
- postcondition(S, Cmd, Result) -- Check the command executed correctly

EQC_STATEM

```
my_prop() ->
    ?FORALL(Cmds,commands(?MODULE),
            begin
                {H,S,Res} = run_commands(?MODULE,Cmds,[]),
                ?WHENFAIL(
                   io:format("History: ~p\nState: ~p\nRes: ~p\n",[H,S,Res])
                   Res == ok)
            end).
my_test() ->
  eqc:quickcheck(my_prop())
```

FIRST MODEL - COMMANDS

```
command(S) ->
  oneof([{call,?MODULE,create_client,[]}] ++
        [{call,?MODULE,get,[elements(S#st.clid), gen_bucket(),
                            gen_key()]} || S#st.clients /= []] ++
        [{call,?MODULE,put,[elements(S#st.clviews), binary()]} ||
                                    S#st.clviews /= []]}]).
```

FIRST MODEL - STATE

```
-record(st,{clients=[],
            clviews=[],
            contents=[]}).
```

riak

FIRST MODEL - EXECUTION

```
create_client() ->
    {ok, C} = riak:local_client(),
    C.
get(C, B, K) ->
    C:get(B, K).

put({{C, B, K}, GetResult}, V) ->
    case GetResult of
        {error, notfound} ->
            C:put(riak_object:new(B, K, V));
        {ok, Obj} ->
            C:put(riak_object:update_value(Obj, V))
```

FIRST MODEL - NEXT STATE

```
%% Next state transformation, S is the current state
next_state(S,C,{call,_,create_client,_}) ->
    S#st{clients=S#st.clients ++ [C]};
next_state(S,R,{call,_,get,[_C,B,K,V]}) ->
    S#state{clviews = lists:keystore({C,B,K}, 1,
                                     S#state.clviews,
                                     {{C,B,K},R})};
next_state(S,_R,{call,_,put,[_C,B,K,V]}) ->
    S#st{contents=[{{B,K},V} |
                   lists:keydelete({B,K},1,S#st.contents)]};
next_state(S,_V,{call,_,_,_}) ->
    S.
```

riak

FIRST MODEL - POSTCONDITIONS

```
postcondition(S,{call,_,get,[_C,B,K]},R) ->
    case R of
        {error, notfound} ->
            not lists:keymember({B,K}, 1, S#st.contents);
        {ok, Obj} ->
            GetVal = riak_object:get_values(Obj),
            ExpVal = [V || {{B1,K1},V} <- S#st.contents, B==B1, K==K1],
            if
                GetVal == ExpVal ->
                    true;
                true ->
                    {got, GetVal, expected, ExpVal}
            end
```

WORKS, BUT...

- The model works!
- But it isn't testing
  - Failure
  - Network Partitions

DESPITE THAT, STILL FOUND A BUG!

- Intermittently saw strange duplicated output

```
Res: {postcondition,{got,
                     [<<"Ý;úw|">>,<<"Ý;úw|">>,<<"ÝÍ">>,<<"ÝÍ">>,
                      <<221,243,200,4,205,205,141,202>>,
                      <<221,243,200,4,205,205,141,202>>],
                     expected,
                     [<<"Ý;úw|">>,<<"ÝÍ">>,
                     <<221,243,200,4,205,205,141,202>>]}}
```

DUPLICATED OBJECTS

- Hard to reproduce - tried ?ALWAYS/?SOMETIMES.
- Enough details in ?WHENFAIL - Scott spotted different timestamps
- Switched from Gregorian seconds to now() - easy reproduction.
- bz://977 in bugzilla & fixed in the product.

IMPROVING MODEL WAS HARD

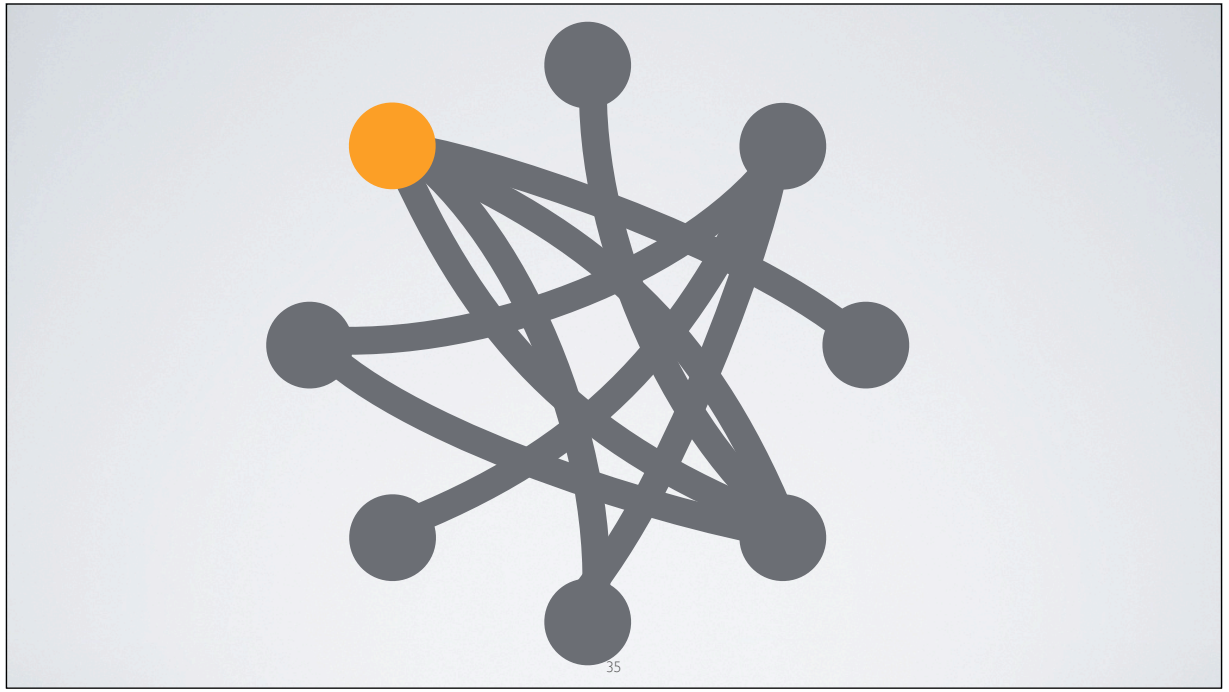
- Added node up / node down commands.
- Resetting a cluster of nodes is fiddly ... and it takes a long time
- Speedups by running on a single node hampered by side-effects.
  ... heroic efforts by Scott on both fronts.
- All of a sudden, very hard to know what the correct postconditions should be.

PAUSE & RETHINK MODEL

- Interested in failure cases.
- Riak is eventually consistent.
- Insight: **eventually** consistent means in the end.

RIAK EVENTUAL CONSISTENCY

- Well behaved clients must get before they put.
- When Riak accepts a write:
  - New value **must** appear eventually unless overwritten.
  - Old values **may** appear again (but do not have to).
- Caveat: as long as all replicas of a key are not destroyed!



All the key changes is the preference list.
These are really just rotations of the same problem.
Save on bookkeeping.
Remind - network partitions and node failure have the same mechanisms.

Riak is
a scalable, highly-available, networked
key/value store.

~~Riak~~ is
a ~~scalable~~, highly-available, ~~networked~~
~~key/~~value store.

EC model tests a highly-available value store.

PAUSE & RETHINK MODEL

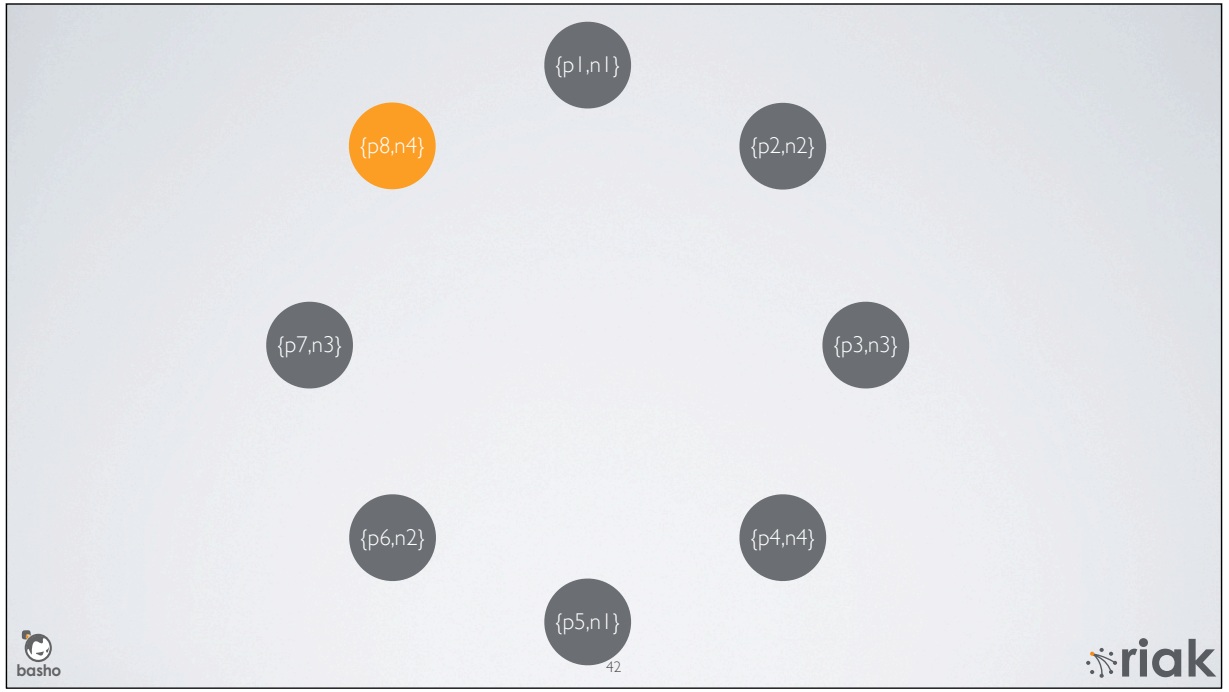
- Model from the point of view of a single key - less bookkeeping.
- Fix the preference list to partitions 1..N - less logic
- Fix the node ownership: node1 owns partition1, node2 owns partition 2.
- Run inside a single VM for speed.

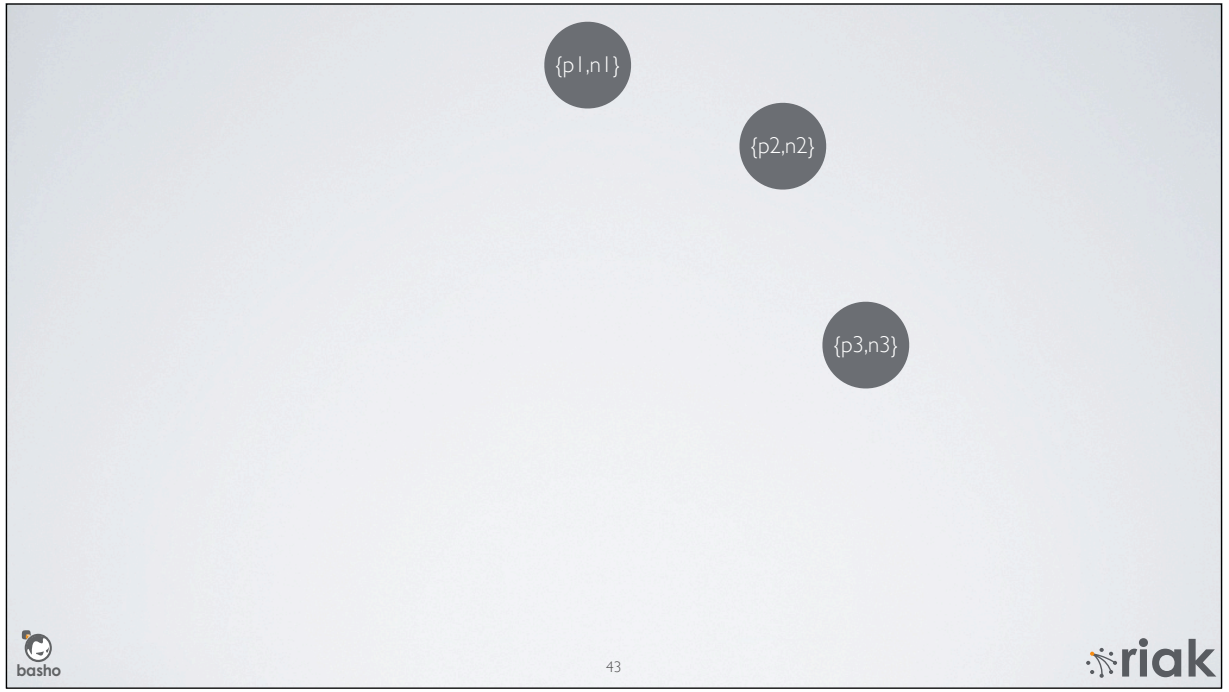
NEW MODEL

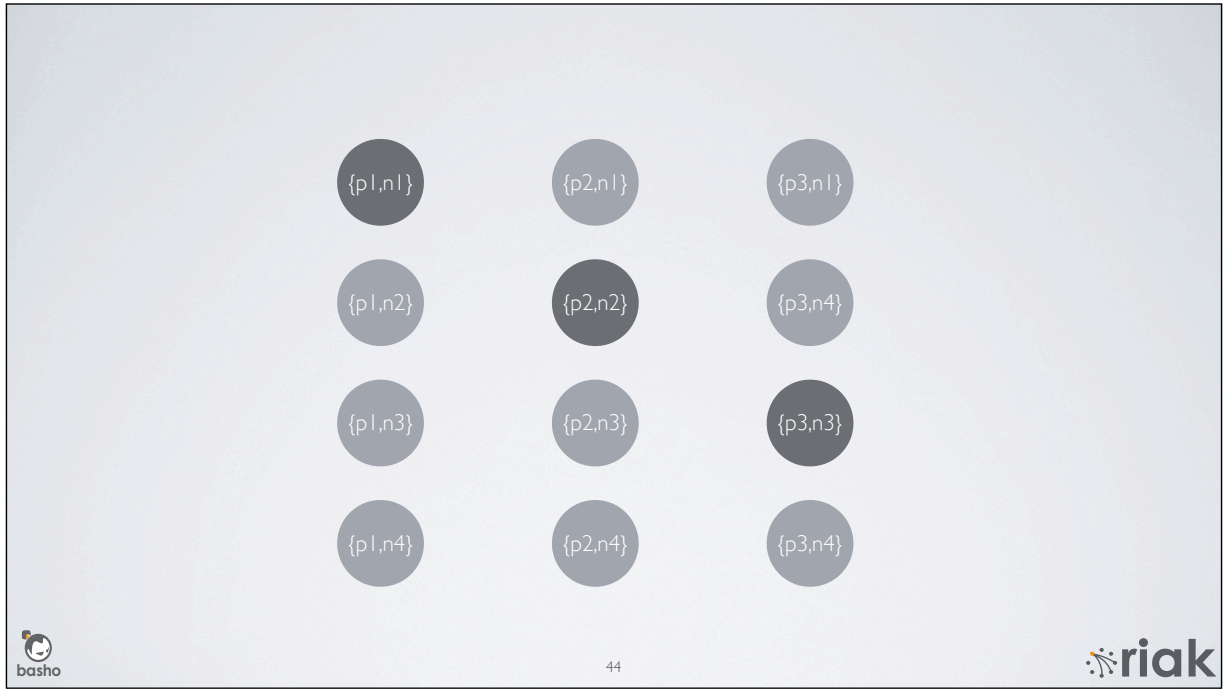
- Phase 1 - Failure, network partitions, chaos etc
- Phase 2 - Recovery

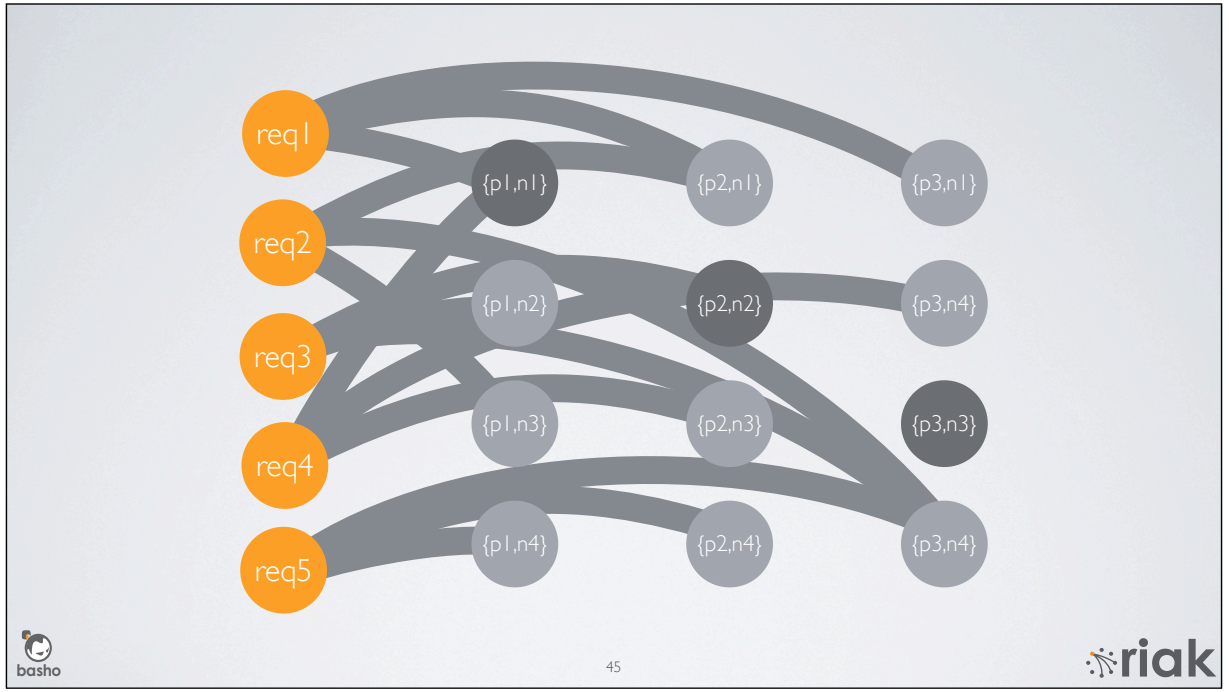
MODEL PHASE 1 - CHAOS

- Create preference lists at random.
  For vnodes 1..N choose a node from 1..M.
- Record the values accepted at put.


{p1,n1}
{p8,n4}
{p2,n2}
{p7,n3}
{p3,n3}
{p6,n2}
{p4,n4}
{p5,n1}
basho
riak


{p1,n1}
{p2,n2}
{p3,n3}


{p1,n1}
{p2,n1}
{p3,n1}
{p1,n2}
{p2,n2}
{p3,n4}
{p1,n3}
{p2,n3}
{p3,n3}
{p1,n4}
{p2,n4}
{p3,n4}


req1
req2
req3
req4
req5
{p1,n1}
{p2,n1}
{p3,n1}
{p1,n2}
{p2,n2}
{p3,n4}
{p1,n3}
{p2,n3}
{p3,n3}
{p1,n4}
{p2,n4}
{p3,n4}

basho

riak

STATE

```
-record(state,{store = [], % [{P,N},VnodeState]
               cids=[],    % client ids
               clviews=[], % client views
               must=[],    % Must appear eventually
               may=[],     % May appear eventually
               value_counter=1
               %% ...
              }.
```

riak

```
command(S) ->
  oneof([{call,?MODULE,new_cid,[S#st.cids]}] ++
        [{call,?MODULE,get,[elements(S#st.cids),
                            gen_pref_list(S),
                            S#st.store]} ||
                                 S#st.cids /= []] ++
        [{call,?MODULE,put,[elements(S#st.clviews),
                            gen_pref_list(S),
                            S#st.val_counter,
                            S#st.store]} ||
```

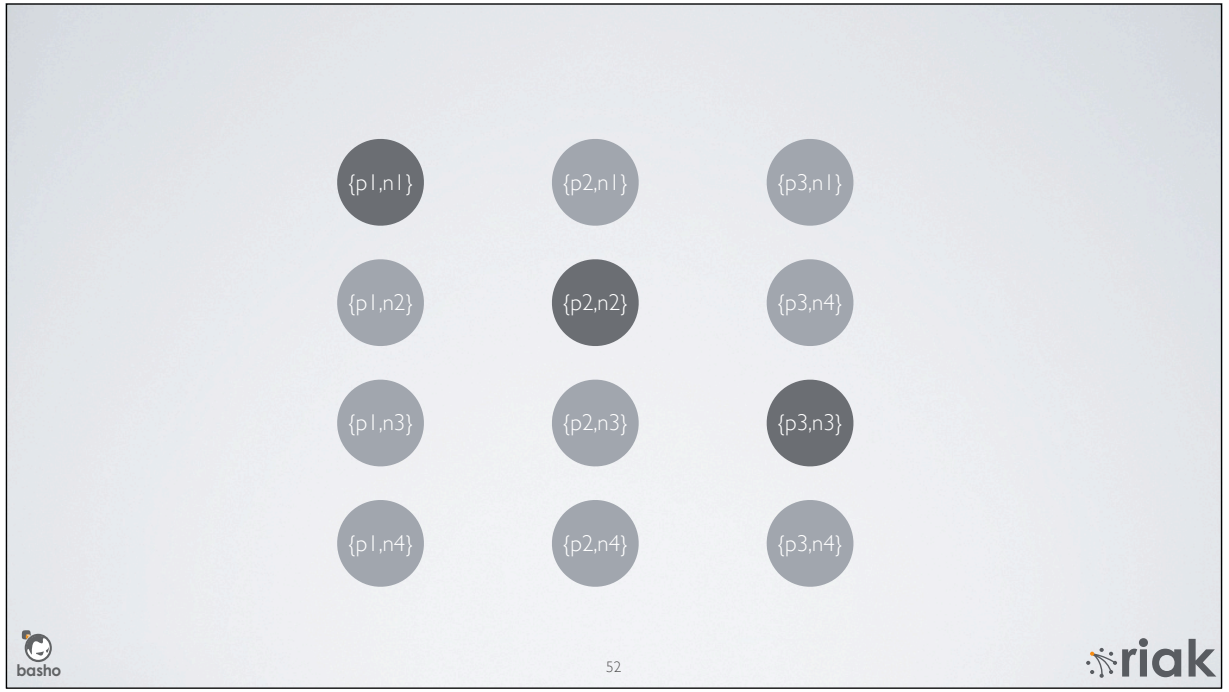
```
%% Next state transformation, S is the current state
next_state(S,Cid,{call,_,new_cid,[_Cids]}) ->
    S#st{cids = [Cid | S#st.cids]};

next_state(S,GetRes,{call,_,get,[Cid, Pl, _Store]}) ->
    S#st{clviews = [{Cid,GetResult,S#st.must} |
                    remove_cid(Cid, S#st.clviews)]};
```

```
next_state(S,NewStore,{call,_,put,
    [{_Cid, _GetResult, MustAtGet}=ClView,
     Pl, Value, _Store]}) ->
   S#st{store = NewStore,
        must = [Value | S#st.must -- MustAtGet],
        may = MustAtGet ++ S#st.may,
        clviews = lists:delete(CV, S#st.clviews),
        value_counter = S#st.value_counter + 1}.
```

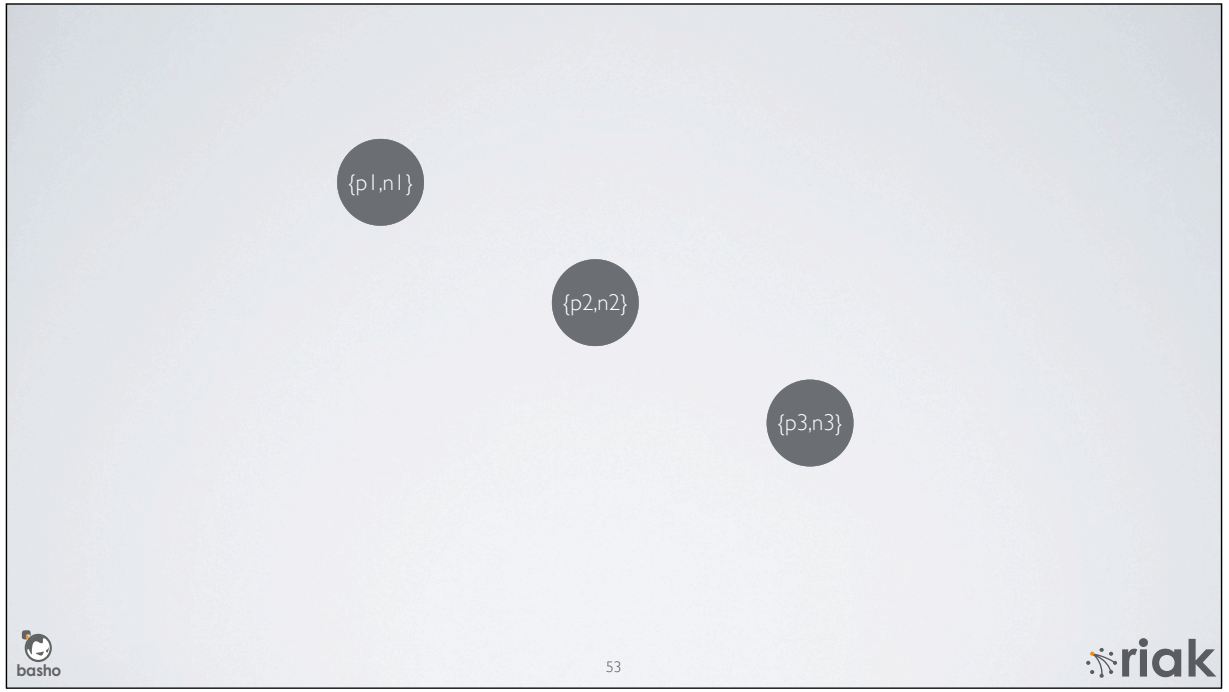
```
get(Cid, PrefList, Store) ->
    VnodeObjs = get_vnodes(PrefList, Store),
    current:get_fsm(VnodeObjs, Cid).

put({Cid,GetRes,_MustAtGet}, PrefList, Value, Store) ->
    VnodePut = current:put_fsm(Cid, GetResult, Value),
    put_vnodes(VnodePut, PrefList, Store).
```

MODEL PHASE 2 - RECOVERY

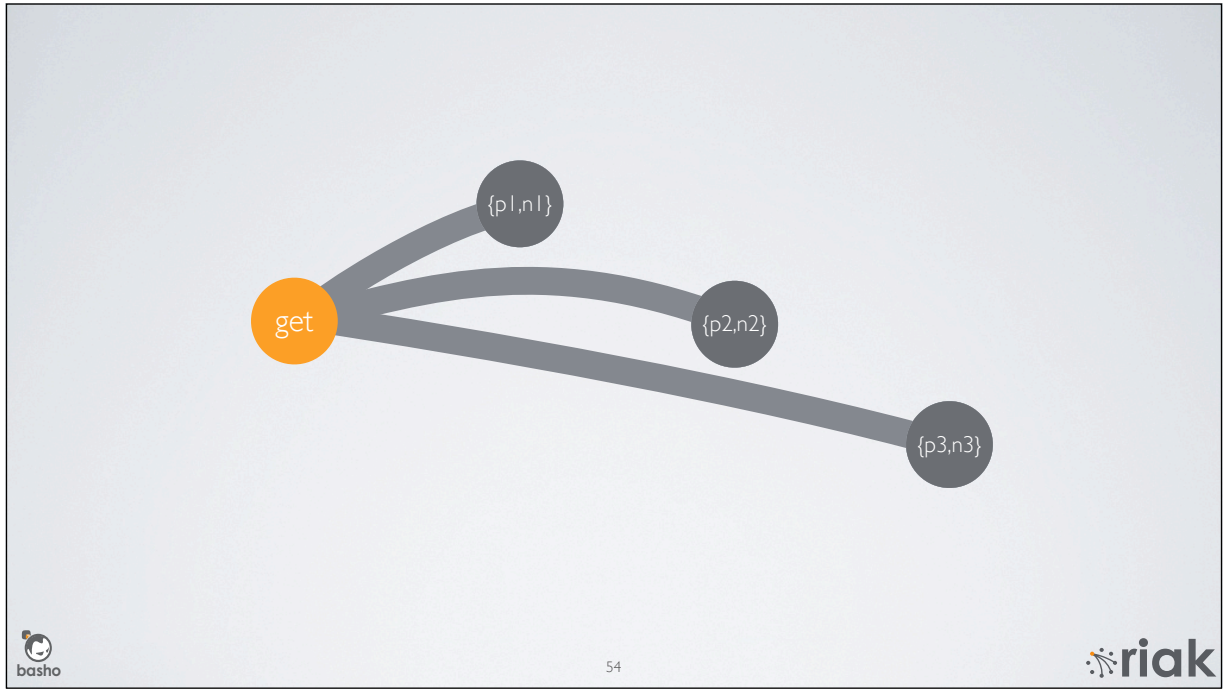
- Execute Riak handoff process against last state (returned by eqc_statem:run_commands).
- Do one last get against the final state.
- Check the returned object contains all the 'must' values.



The N value stores how many replicas of each key are stored.
As nodes may come and go, there is a chance of the cluster containing stale data or conflicts.
Riak handles this by versioning objects with vector clocks and requesting all N objects.
Using the vector clock we can tell if the data is just stale or in conflict.
Riak can handle this with a last timestamp wins strategy or provide the conflicts back to the application.



The N value stores how many replicas of each key are stored.
As nodes may come and go, there is a chance of the cluster containing stale data or conflicts.
Riak handles this by versioning objects with vector clocks and requesting all N objects.
Using the vector clock we can tell if the data is just stale or in conflict.
Riak can handle this with a last timestamp wins strategy or provide the conflicts back to the application.



The N value stores how many replicas of each key are stored.
As nodes may come and go, there is a chance of the cluster containing stale data or conflicts.
Riak handles this by versioning objects with vector clocks and requesting all N objects.
Using the vector clock we can tell if the data is just stale or in conflict.
Riak can handle this with a last timestamp wins strategy or provide the conflicts back to the application.

```
complete_handoff(S) ->
    Fallbacks = [El || {{P1,N1},_O}=El <- S#st.store,
                       P1 /= N1],
    Primaries = S#state.store -- Fallbacks,
    Final =  lists:foldl(fun(Fallback, Store1) ->
                           handoff(Fallback, Store1)
                         end, Primaries, Fallbacks),
    S#state{store = Final}.
```

```
check_consistency(S) ->
    PriPl = [{P,P} || P <- lists:seq(1, S#st.n)],
    case get(new_cid(S#st.cids), PriPl, S#st.store) of
        notfound ->
            S#state.must == [];
        O ->
            GotVals = current:obj_values(O),
            {S#st.must -- GotVals,
             GotVals -- (S#st.must ++ S#st.may)} ==
            {[], []}
```

SUCCESS

- Created a simple model for a complex system.
- While testing extreme failure scenarios.
- Model did not need to know about system details - just a property.

FURTHER WORK

- Had to extract code to make the model - want to run the real code.
- Model lost messages.
- Message delivery order.
- Other operations - e.g list keys.

CLOSING THOUGHTS

- As always with QuickCheck, think about the **properties** of your system.
- No one model to rule them all (or at least not one a mortal like me can understand).

Please visit our website and check us out.
We have a FastTrack section on our wiki to go through all you need to get started.
Thank you for listening.