

Test-First Construction of Distributed Systems

Erlang Factory SF
March 2012

Joseph Blomstedt (@jtuple)
Basho Technologies
joe@basho.com



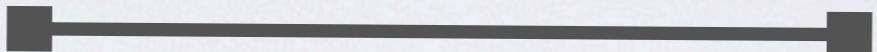
Basho makes



a distributed, scalable, and highly-available datastore store.

Basho is a start-up



Ship Quickly  Ship Correctly

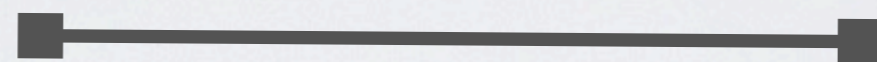
Start-up
Iterate
Agility

Highly Available
Fault Tolerant
Enterprise



Ship Quickly

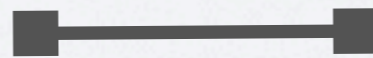
Start-up
Iterate
Agility



Ship Correctly

Highly Available
Fault Tolerant
Enterprise

Strive to reduce gap



Erlang Is Indispensable

- Built-in concurrency and distributed programming
- Fault-tolerant just-crash / supervisor mentality
- Ability to inspect VM state
- Hot load code loading

Result?



7



Majority of bugs are concurrent logic errors



Testing Tools

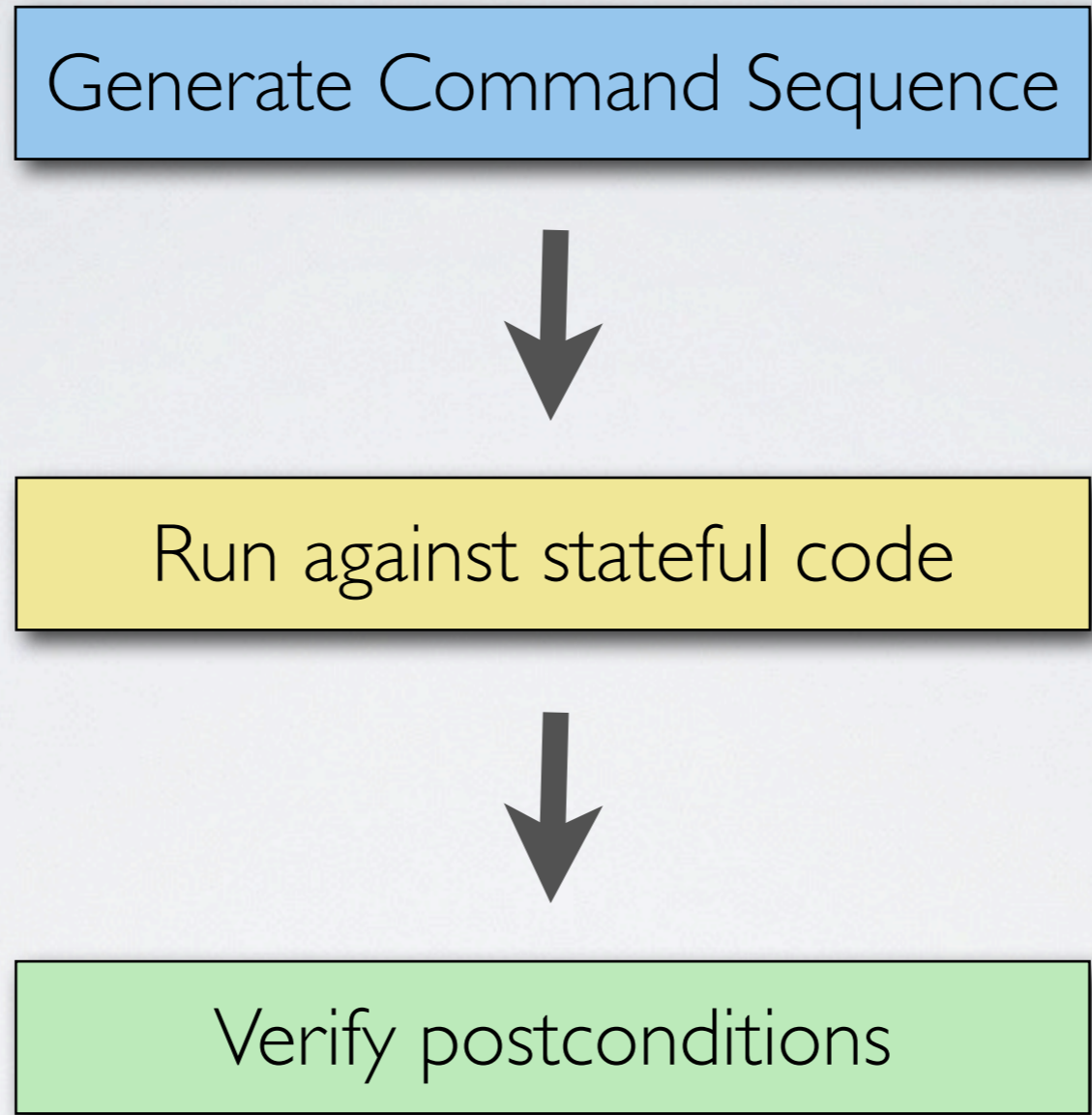
- Quickcheck
Property-based testing
- Pulse
Randomizing Erlang scheduler
- McErlang / Concuerror
Model checkers

Quickcheck

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

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

Quickcheck eqc_statem



Quickcheck eqc_statem

`command(State) ->`

`%% Commands to run against stateful system
oneof(Cmds).`

`precondition(State, Cmd) ->`

`%% Return true if cmd is valid in current state.`

`next_state(State, Result, Cmd) ->`

`%% Update test state after a given cmd.`

`postcondition(State, Cmd, Result) ->`

`%% Test postconditions.`

Testing Issues

- Building test from implementation often not straightforward
- Testing concurrent interleaving requires a different approach
- Building a great implementation of a broken algorithm is disheartening

Test First Construction



Build testable model



Test
Iterate
Gain Confidence



Convert model into implementation



Verify implementation against model



History

- First built testable model for new clustering subsystem for Riak 1.0
- Model built on top of eqc_statem
- The test itself was the model of the system and tested properties against itself
- Somewhat ad-hoc, but it worked

eqc_system (1/2)

- Refactored the approach into general-purpose framework based on lessons learned
- Events
External events, timers, things you do not care to model
- Calls/Casts
Similar to OTP `gen_server`
- Calls/casts map to simulated receive/reply semantics

eqc_system (2/2)

- Test consists of test module and a set of node modules
- Callbacks
 - handle_event, handle_call, handle_cast
 - after_event, after_call, after_cast
 - post_event, post_call, post_cast, always
- Test module can generate events and test properties against global test state
- Node modules generate events, calls, casts and test local properties

Simple Example

- Nodes join together and form a cluster
- Nodes periodically gossip membership state to other known nodes

Test Module

```
events(#state{nodes=Nodes}) ->  
  ?EVENT(join, [elements(Nodes), elements(Nodes)]).
```

```
precondition(_, S, join, [Node, [OtherNode]]) ->  
  Singleton = S#state.singleton,  
  all([Node /= OtherNode,  
    lists:member(Node, Singleton),  
    (Singleton == S#state.nodes) or  
    lists:member(OtherNode, Singleton)]);
```

```
after_event(_Nodes, S, {join, [OtherNode]}, Node, _NodeState) ->  
  Singleton = S#state.singleton -- [Node, OtherNode],  
  S#state{singleton=Singleton};
```

Test Node Module (1/3)

```
events(Node, #state{members=Members}) ->  
  ?EVENT(gossip, [Node, [elements(Members)]]).
```

```
precondition(S, gossip, [Node, [OtherNode]]) ->  
  all([lists:member(OtherNode, S#state.members),  
       Node /= OtherNode]);
```


Test Node Module (2/3)

```
handle_event({join, [OtherNode]}, State) ->
  call(State, OtherNode, get_members,
    fun(Members) ->
      Members2 =
        lists:sort([State#state.id | Members]),
      State2 = State#state{members=Members2},
      {noreply, State2}
    end);
```

```
handle_event({gossip, [OtherNode]}, State) ->
  cast(OtherNode, {gossip, State#state.members}),
  {ok, State}.
```

Test Node Module (3/3)

```
handle_call(get_members, _From, State) ->  
  {reply, State#state.members, State}.
```

```
handle_cast({gossip, OtherMembers}, State) ->  
  Members2 = merge(Members, OtherMembers),  
  {noreply, State#state{members=Members2}}.
```

Example Command Sequence

```
[{init, {sys_state, undefined, undefined, rc, 0, [], undefined, undefined,
      model}}},
 {set, {var, 1}, {call, eqc_sys, init_dynamic, []}},
 {set, {var, 2}, {call, eqc_sys, init_system, [rc]}},
 {set, {var, 3}, {call, rc, join, [3, [1]]}},
 {set, {var, 4}, {call, eqc_sys, rcvmsg, [1, {3, {call, get_members}}]}},
 {set, {var, 5}, {call, rc, join, [2, [1]]}},
 {set, {var, 6}, {call, eqc_sys, rcvreply, [3, {1, [1]}]}},
 {set, {var, 7}, {call, eqc_sys, rcvmsg, [1, {2, {call, get_members}}]}},
 {set, {var, 8}, {call, eqc_sys, rcvreply, [2, {1, [1]}]}},
 {set, {var, 9}, {call, rc_node, send_gossip, [3, [1]]}},
 {set, {var, 10}, {call, rc_node, send_gossip, [3, [1]]}},
 {set, {var, 11}, {call, rc_node, send_gossip, [2, [1]]}},
 {set, {var, 12}, {call, rc_node, send_gossip, [3, [1]]}},
 {set, {var, 13}, {call, eqc_sys, rcvmsg, [1, {3, {cast, {gossip,
[1, 3]}}}]}},
 {set, {var, 14}, {call, eqc_sys, rcvmsg, [1, {3, {cast, {gossip,
[1, 3]}}}]}}
```

Extended Example

- Cluster maintains a weak leader

Lowest node id in the cluster is considered the leader

No actual leader election or failure detection

- Property we care about

At all times, there is only one node that believe it is the leader of a cluster

Extended Node Module (1/3)

```
-record(state, {id, members, leader}).
```

```
handle_event({join, [OtherNode]}, _Node, State) ->  
  call(State, OtherNode, get_state,  
    fun(#state{members=Members, leader=Leader}) ->  
      Members2 =  
        lists:sort([State#state.id | Members]),  
      {noreply, State#state{members=Members2,  
        leader=Leader}}  
    end);
```

```
handle_event({send_gossip, [OtherNode]}, _Node, State) ->  
  cast(OtherNode, {gossip, State}),  
  {ok, State};
```



Extended Node Module (2/3)

```
handle_call(get_state, _From, _Node, State) ->  
  {{reply, State}, State};
```

```
handle_cast({gossip,  
            #state{members=Members, leader=Leader}},  
            _From, _Node, State) ->  
  Members2 = lists:usort(State#state.members ++ Members),  
  case State#state.id == State#state.leader of  
    true ->  
      Leader2 = hd(lists:sort(Members2));  
    false ->  
      Leader2 = Leader  
  end,  
  {noreply, State#state{members=Members2, leader=Leader2}}};
```



Extended Node Module (3/3)

```
get_leader(S) ->  
  S#state.leader.
```

```
get_members(S) ->  
  S#state.members.
```

Extended Test Module

```
always(Nodes, S) ->
  all([begin
    Members = nodecall(Nodes, Node, get_members, []),
    one_leader(Nodes, Members)
  end || Node <- S#state.nodes]).
```

```
one_leader(Nodes, Members) ->
  Leaders = [Leader || Node <- Members,
    Leader <- [nodecall(Nodes, Node,
      get_leader, [])],
    Leader == Node],
  length(lists:usort(Leaders)) < 2.
```


Counterexample

```
[{init, {sys_state, undefined, undefined, rc, 0, [], undefined, undefined, model}},
 {set, {var, 1}, {call, eqc_sys, init_dynamic, []}},
 {set, {var, 2}, {call, eqc_sys, init_system, [rc]}},
 {set, {var, 3}, {call, rc, join, [1, [3]]}},
 {set, {var, 4}, {call, eqc_sys, rcvmsg, [3, {1, {call, get_state}}]}},
 {set, {var, 5}, {call, eqc_sys, rcvreply, [1, {3, {state, 3, [3], 3}}]}},
 {set, {var, 6}, {call, rc_node, send_gossip, [1, [3]]}},
 {set, {var, 7}, {call, eqc_sys, rcvmsg, [3, {1, {cast, {gossip, {state, 1, [1, 3], 3}}}}]}},
 {set, {var, 8}, {call, rc_node, send_gossip, [3, [1]]}},
 {set, {var, 9}, {call, rc_node, send_gossip, [1, [3]]}},
 {set, {var, 16},
  {call, eqc_sys, rcvmsg, [3, {1, {cast, {gossip, {state, 1, [1, 3], 3}}}}]}},
 {set, {var, 18},
  {call, eqc_sys, rcvmsg, [1, {3, {cast, {gossip, {state, 3, [1, 3], 1}}}}]}]}
{postcondition, false}
```

Versioned leader state

- Add version number to gossiped state
- Leader increments version when changed
- Node updates leader only if newer version
- After changes, model passes without issue

Convert to Implementation



Convert to Implementation

- Convert model into actual implementation
- Majority of code reused
eqc_sys designed to mirror OTP code
- Update model if as necessary and reiterate

Test Implementation



Recall model design

- Events

Commands that trigger system transitions

- Calls/casts

Emulated as commands in order for testing purposes

Testing Approach # 1

- Quickcheck generates event sequences, not call/casts
- Events mapped to equivalent implementation constructs
- Erlang tracing used to capture actual call/casts that occurred
- Verify events + observed call/casts against model and final cluster state

Testing Approach #2

- Modify implementation to enable controlling message interleaving
- Implemented as a proxy process that delays forwarding messages until told to do so by test module
- Investigating `parse_transform` option

Interacting with other tools

- Pulse, McErlang, Concuerror
All aimed at concurrency debugging
- Testing approach #1 works well with these tools
Generate event sequences + trace, but allow scheduling tools to force interleavings
- Tested with Pulse and Concuerror
- Even more confidence in model/code

Limitations

- `eqc_sys`
entirely random, may not hit lurking bad interleaving
- `Pulse`
also random
- `McErlang / Concuerror`
state space usually too large

Coq Proof Assistant

- Working on using Coq to prove model
- Coq script similar to Quickcheck model

Represent commands as a list constructed from a generate

Model are functions that operate over list, producing state

Properties checked against state

Prove: Forall commands, properties always hold.

Coq Challenges (1/2)

- Writing Coq scripts

Syntax (Basho is an Erlang company)

Semantics (Mapping Erlang ideas to Coq)

- Working on Erlang to Coq generate that works on subset of Erlang used in my models

Solves syntax issues

Semantics are trickier, but approached as encountered

Coq Challenges (2/2)

- Proving in Coq is not automatic
- Tedious process, not Basho specialty
- Working on domain-specific proof tactic and library of lemmas to enable automated
- Inspired by Professor Chlipala's book
<http://adam.chlipala.net/cpdt>
- Possibly hear more later this year
Personal project, so progress is slow

Model



Test



Implement



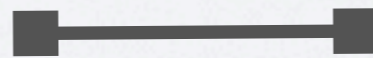
Verify



Ship Quickly ■————■ Ship Correctly



Getting a little closer



Questions?

joe@basho.com
@jtuple