Usually Received, Maybe Late, or Sometimes Dropped

Scott Lystig Fritchie, Basho Technologies, <scott@basho.com>

Erlang Factory London, June 10, 2011

Who is Scott?

Me

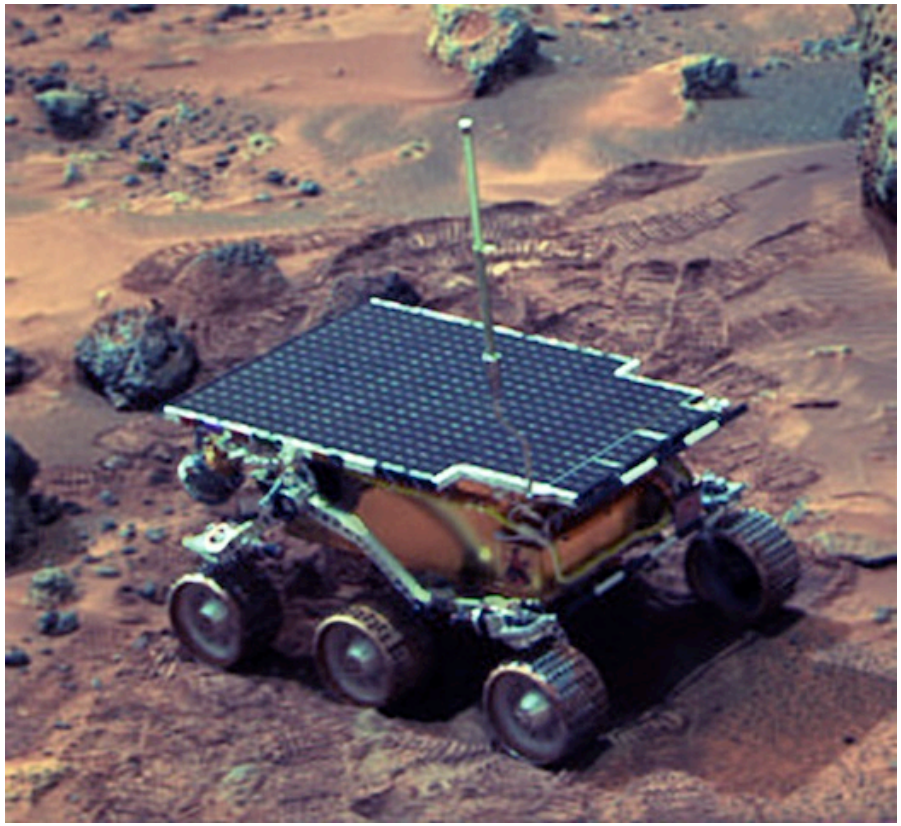
My Employer

basho

Why verify a protocol?

- Bugs are expensive, especially "in the field"
- Expense:
  - *Money*
  - *Time*
  - *Reputation*
  - *Life*

"In the field" / "on another planet"



Credit: NASA

A flawed protocol will *always* be buggy

- If it will never work correctly, why bother?
- Find bugs in your development environment, not "in the field"

Typical protocol testing goals

- Requirements: vague, fuzzy, uncertain generalities
- Does what it is supposed to do?
- Does not do anything that it is not supposed to do?
- ... and then a miracle occurs.

Desirable protocol verification goals

- Can a design requirement be violated?
  - *Find a counter-example*
- Executable
- Verification is independent of execution time
  - *CPU speed, process scheduling, network latency, ...*
- Find error possibility, not probability
- Test software, not hardware (different kettle of fish)

Properties

- Safety: something bad never happens
  - *Example: claim that property X is never violated*
- Liveness: something good will always happen
  - *Example: claim that service Z can always be queried successfully*
- A matter of time
  - *Safety violations happen in finite time*
  - *Liveness violations happen in infinite time*

Verification claims

- About state: a state is reachable or unreachable
  - *Example: bank balance is always greater than 0*
- About execution: an execution path is possible or impossible
  - *Example: if PIN is incorrect, then bank balance is never transmitted*

msgdropsim primary goals 1 of 2

- Test message passing algorithms in concurrent systems
- Support selective receive
- Deterministic process scheduling unfairness
- Deterministic message dropping
  - *The Erlang VM is "too good"*
- Easy to test for safety violations
  - *Program state claims*
  - *Execution path claims*

msgdropsim primary goals 2 of 2

- Methods for testing verification claims
  - *state check (NOTE: not actually implemented yet!)*
  - *execution path check (via trace logs)*
  - *halting/termination check*
- Use QuickCheck Mini
- Use side-effect free code, nothing too Erlang-specific
  - *Technique is feasible in Ruby, Python, ...*
  - *Feasible with QuickCheck-like libraries: Ruby, Python, ...*

msgdropsim secondary goals

- Coding style similar to gen_fsm
  - *Though gen_fsm doesn't support selective receive*
- Don't use any commercial-QuickCheck-only features
  - *i.e., Play well with PropEr*
- Play well with McErlang
- Support liveness property testing (via McErlang)
  - *Indirectly tested via halting/termination check*

Next in this talk: msgdropsim workflow

- "Install" msgdropsim and QuickCheck Mini
- Write protocol simulation code
  - *assumptions*
  - *gen_fsm-like style example*
- QuickCheck generates some inputs
  - *Mostly hidden from the user, hooray!*
- Run simulator with all inputs
  - *Process scheduler, trace logs, message sending, selective receive*
- Check results

Install QuickCheck Mini and msgdropsim

- Erlang R13 or R14 is fine
- PropEr should work, but I've not tried it, sorry!
- QuickCheck Mini
  - *http://www.quviq.com/news100621.html*
  - *Follow the directions*
- msgdropsim
  - *https://github.com/slfritchie/msgdropsim*
  - *git clone git://github.com/slfritchie/msgdropsim.git*
  - *See README.md for "How to run simulated protocols"*

Before writing code: some assumptions 1 of 3

- Multiple Erlang-like processes run concurrently
  - *Very familiar to gen_fsm, gen_server, "raw" Erlang users*
- Processes communicate via message passing
  - *Timeouts are supported*
  - *Process linking and monitoring are not supported*
- Two types of processes: clients, servers
- All processes have a registered name

msgdropsim assumptions 2 of 3

- Pure Erlang code plus message passing
  - *Impure = side-effects*
  - *Impure is not impossible, but debugging can be horrible*
- List of operations: "What should a simulation do?"
  - *Your code: make individual operation tuples*
  - *QuickCheck: make random combinations of ops*
  - *Ops are sent as messages at start of simulation*
- A process must receive a message before it can become runnable!
- All processes run FSM-style code
  - *Old state **X** + input message **M** => Do Stuff => new state **Y***

msgdropsim assumptions 3 of 3

- Message receive callback is the scheduler's unit of granularity
  - *No preemption while executing a single callback*
- Scheduler runs until all processes block waiting for messages
- Scheduler maintains two trace logs for property verification
  - *System trace: all scheduling and message events*
  - *Example: c1 receives 'foo' from s2, c1 sends 'foo' to s3*
  - *User trace: events generated by simulation code annotations*
  - *Example: c4 submitted novel to publisher*
- Your verify_property/11 function checks traces for safety violations

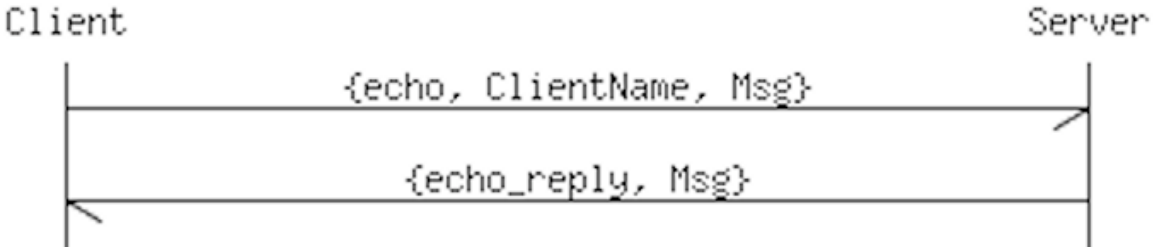
Code: writing callback functions

- `gen_initial_ops(NumClients, NumServers, NumKeys, OptionList)`
- `gen_client_initial_states(NumClients, OptionList)`
- `gen_server_initial_states(NumServers, OptionList)`
- `verify_property/11`
- `all_clients()`
- `all_servers()`
- one function (arity 2) for each FSM state for clients, servers

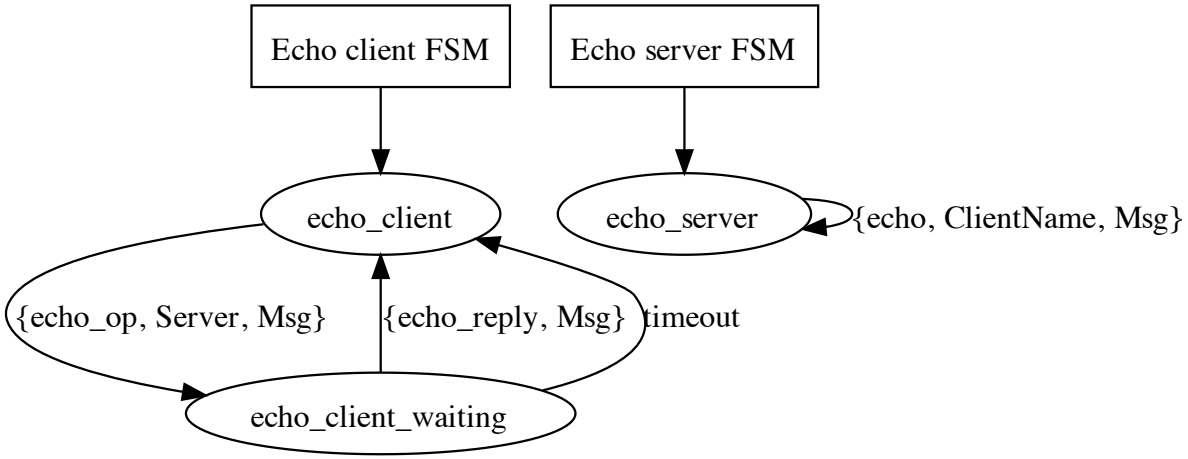
Echo service callbacks, 1 of 4

```
all_clients() ->
  [c1, c2, c3, c4, c5, c6, c7, c8, c9].

all_servers() ->
  [s1, s2, s3, s4, s5, s6, s7, s8, s9].

%% spec (integer(), property_list()) ->
%%      list({atom(), term(), fun() | atom()}).

gen_client_initial_states(NumClients, _OptionList) ->
  Clients = lists:sublist(all_clients(), 1, NumClients),
  [{Clnt, [], fun echo_client/2} || Clnt <- Clients].

gen_server_initial_states(NumServers, _OptionList) ->
  Servers = lists:sublist(all_servers(), 1, NumServers),
  [{Server, placeholder, fun echo_server/2} || Server <- Servers].
```

Echo service message sequence diagram



Echo service FSM state diagram



Echo service callbacks, 2 of 4

```
%% spec (integer(), integer()) ->
%%       list{atom(), term()}).

gen_initial_ops(NumClients, NumServers, _NumKeys, _Props) ->
    list(gen_echo_op(NumClients, NumServers)).

gen_echo_op(NumClients, NumServers) ->
    ?LET({ClientI, ServerI},
         {choose(1, NumClients),
          choose(1, NumServers)},
         begin
           Client = lists:nth(ClientI, all_clients()),
           Server = lists:nth(ServerI, all_servers())
           {Client, {echo_op, Server, int()}}
         end).
```

Echo service callbacks, 3 of 4

```
%% spec (SelectiveReceiveMsg::term(), State::term()) ->
%%       {recv_general, fun() | atom(), NewState::term()} |
%%       {recv_timeout, fun() | atom(), NewState::term()}.

echo_client({echo_op, Server, Key}, ReplyList) ->
    slf_msgsim:bang(Server,
                    {echo, slf_msgsim:self(), Key}),
    {recv_timeout, echo_client_waiting, ReplyList}.

echo_client_waiting(timeout, ReplyList) ->
    NewReplyList = [server_timeout|ReplyList],
    {recv_general, echo_client, NewReplyList};
echo_client_waiting({echo_reply, Msg}, ReplyList) ->
    {recv_general, echo_client, [Msg|ReplyList]}.

echo_server({echo, Client, Msg}, St) ->
    slf_msgsim:bang(Client, {echo_reply, Msg}),
    {recv_general, same, St}.
```

QuickCheck inputs

- First, QuickCheck chooses:
  - *Number of client processes*
  - *Number of server processes*
  - *A key number (usually unused ... code bitrot)*
- Second, QuickCheck chooses:
  - *(cb) Initial operation list*
  - *(cb) Initial state data for client procs*
  - *(cb) Initial state data for server procs*
  - *(int) Scheduler token list*
  - *(int) Network partition list*
  - *(int) Message delay list*

Process runnable states and message handling

- Runnable states:
  - *mbox: Try to receive a message from the inbox*
  - *outbox: Try to send a queued message*
- Message sending is not instantaneous
  - *message may be dropped (network partition)*
  - *message may be delayed (consume extra scheduler tokens)*

The admission token scheduler, network partitions, message delays

- QuickCheck creates a list of tokens to drive scheduler
- 1 token = a process name
  - `[c1, s2, s1, s2, s2, s2]`
- Network partitions and delays
  - `{partition, FromProcs, ToProcs, StartStep, EndStep}`

Run simulator with all the inputs

```
$ cd /path/to/top/of/msgdropsim
$ make
$ erl -pz ./ebin
[...]
> Prop1 = slf_msgsim_qc:prop_simulate(echo_sim, []).
> eqc:quickcheck(Prop1).
```

Options

```
[{min_clients, N}, {max_clients, M},  % def: N=1, M=9
 {min_servers, N}, {max_servers, M},  % def: N=1, M=9
 {min_keys, N}, {max_keys, M}         % ignore
 disable_partitions,           % disable network partitions
 disable_delays                % disable message delays
 crash_report,                 % enable verbose crash report
 {stop_step, N}]               % stop execution at step N

> Opts = [{max_servers, 2}, disable_partitions],
> eqc:quickcheck(slf_msgsim_qc:prop_simulate(echo_sim, Opts)).
```

A running simulation: system trace log events

```
{bang, Step, Sender, Rcpt, Msg}
{delay, Step, Sender, Rcpt, Msg, {num_rounds, N}}
{drop, Step, Sender, Rcpt, Msg}
{deliver, Step, Sender, Rcpt, Msg}
{recv, Step, Sender, Rcpt, Msg}
```

Implementing selective receive

- Erlang VM implements SR deep in the virtual machine
- We need to fake it.

"Impurity in the defense of liberty is no vice." -- Barry Codewater

- Selective receive has side-effects (duh!)
- Faking it is ugly
  - *A monad would be helpful*
- Use process dictionary

Checking results: verify_property/11

- Arguments:
  - *NumClients, NumServers, OptionList*
  - *QuickCheck-generated inputs: Ops list, partitions list, delays list*
  - *Starting simulator state*
  - *Ending simulator state*
  - *System trace list*
  - *User trace list*

Echo service callbacks, 4 of 4

```
verify_property(NumClients, NumServers, _Props, F1, F2,
                Ops, _Sched0, Runnable, Sched1,
                Trc, _UTrc) ->
  Clients = lists:sublist(all_clients(), 1, NumClients),
  Predicted = predict_echos(Clients, Ops),
  Actual = actual_echos(Clients, Sched1),
  Runnable == false andalso
      exact_msg_or_timeout(Clients, Predicted, Actual).

exact_msg_or_timeout(Clients, Predicted, Actual) ->
  lists:all(
    fun(Client) ->
        Pred = proplists:get_value(Client, Predicted),
        Act = proplists:get_value(Client, Actual),
        lists:all(fun({X, X}) ->               true;
                     ({_X, server_timeout}) -> true;
                     (_)                    -> false
                  end, lists:zip(Pred, Act))
    end, Clients).
```

What if verify_property() fails?

```
%% From echo_bad1_sim.erl

?WHENFAIL(
io:format("Failed:\nF1 = ~p\nF2 = ~p\nEnd = ~p\n"
          "Runnable = ~p, Receivable = ~p\n"
          "Predicted ~w\nActual ~w\n",
          [F1, F2, Sched1,
           slf_msgsim:runnable_procs(Sched1),
           slf_msgsim:receivable_procs(Sched1),
           Predicted, Actual]),
%% ...
```

verify_property() failure, 1 of 4

```
Failed:
F1 = {1,1,1}
     % 1 server, 1 client, one key
F2 = {[{c1,{echo_op,s1,14}},{c1,{echo_op,s1,0}}],
      % 2 echo ops: echo 14, then echo 0
      [{c1,[],bad1_client}],
      [{s1,placeholder,bad1_server}],
      % client & server initial state
      [s1,c1],
      % scheduler token list
      [{partition,[],[],0,0}],
      []}
      % partition & delay lists: no interference
```

verify_property() failure, 2 of 4

```
End =
{sched,10,99999999999,4,
 [s1,c1],
 % Scheduler token list
 [],
 [{c1,{proc,c1,[14,14],[],{[],[]},mbox,bad1_client,
       undefined}},
  {s1,{proc,s1,14,[],{[],[]},mbox,bad1_server,
       undefined}}],
 % Final process state: private state, msgs, etc.
```

verify_property() failure, 3 of 4

```
[{recv,9,s1,c1,{echo_reply,14}},
 {deliver,8,s1,c1,{echo_reply,14}},
 {bang,7,s1,c1,{echo_reply,14}},
 {recv,7,c1,s1,{echo,c1,0}},
 {deliver,6,c1,s1,{echo,c1,0}},
 {bang,5,c1,s1,{echo,c1,0}},
 {recv,5,scheduler,c1,{echo_op,s1,0}},
 {recv,4,s1,c1,{echo_reply,14}},
 {deliver,3,s1,c1,{echo_reply,14}},
 {bang,2,s1,c1,{echo_reply,14}},
 {recv,2,c1,s1,{echo,c1,14}},
 {deliver,1,c1,s1,{echo,c1,14}},
 {bang,0,c1,s1,{echo,c1,14}},
 {recv,0,scheduler,c1,{echo_op,s1,14}},
 {deliver,0,scheduler,c1,{echo_op,s1,0}},
 {deliver,0,scheduler,c1,{echo_op,s1,14}}],
% System trace list
[],[],[],echo_bad1_sim,[]}
% User trace list, partition & delay specs, etc.
```

verify_property() failure, 4 of 4

```
Runnable = [], Receivable = []
Predicted [{c1,[14,0]}]
Actual [{c1,[14,14]}]
false
```

Stats when things are "correct"

```
OK, passed 100 tests

29% at_least_1_msg_dropped
clients     : Min: 1   Max: 9   Avg: 4.84   Total: 484
servers     : Min: 1   Max: 9   Avg: 4.60   Total: 460
echoes      : Min: 0   Max: 10   Avg: 3.07   Total: 307
msgs sent   : Min: 0   Max: 18   Avg: 5.00   Total: 500
msgs dropped: Min: 0   Max: 8   Avg: 0.760   Total: 76
timeouts    : Min: 0   Max: 8   Avg: 0.760   Total: 76
true
```

QuickCheck code for measuring stats

```
classify(NumDrops /= 0, at_least_1_msg_dropped,
measure("clients     ", NumClients,
measure("servers     ", NumServers,
measure("echoes      ", length(Ops),
measure("msgs sent   ", NumMsgs,
measure("msgs dropped", NumDrops,
measure("msgs delayed", NumDelays,
measure("timeouts    ", NumTimeouts,
%% ....
```

McErlang: harder than it looks

- You: write "normal" Erlang
  - *Really, a subset of Erlang ... avoid side-effects!*
- McErlang: full exploration of all possible executions
- *Very* easy to find exponential state growth
- Requires much work to create simple tests that fit in RAM

McErlang and msgdropsim status

- Not well integrated, sorry.
- Exhaustive state testing is *hard* to do correctly.
- Selective receive =/= gen_fsm code, so McErlang's gen_fsm support does not help.
- Must convert msgdropsim-callbacks to "raw" Erlang
  - *A parse transform could be a big help, not done yet.*

msgdropsim vs. "raw" Erlang

msgdropsim style:

```
client_waiting({echo_reply, Msg}, St) ->
    {recv_general, client_init, [Msg|St]};
client_waiting(timeout, St) ->
    {recv_general, client_init, [server_timeout|St]}.
```

"raw" Erlang:

```
echo_client_waiting(St) ->
    receive
        {echo_reply, Msg} ->
            client_init([Msg|St])
    after ?SOME_TIME ->
            client_init([server_timeout|St])
    end.
```

We have a memory problem....

Using distrib_counter_2phase_vclocksetwatch_sim.erl with message dropping enabled:

```
1 client  x 1 counter op each x 1 server  =         126 states
2 clients x 1 counter op each x 1 server  =      11,939 states
3 clients x 1 counter op each x 1 server  =   1,569,343 states
2 clients x 1 counter op each x 2 servers =  13,140,204 states
2 clients x 1 counter op each x 3 servers = 149,884,834 states
4 clients x 1 counter op each x 2 servers = 387,461,768 states
                                            (5.5 hours)
```

- I recommend "The SPIN Model Checker: Primer and Reference Manual"

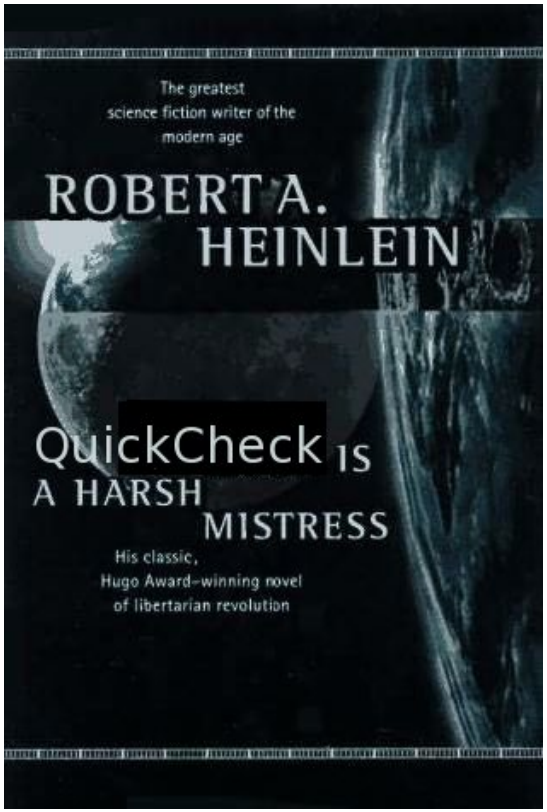
Message dropping and McErlang

```
mc_bang(Rcpt, Msg) ->
    Send = fun() ->
             Rcpt ! Msg
           end,
    Drop = fun() ->
             mce_erl:probe({drop_msg, mc_self(),
                            Rcpt, Msg})
           end,
    mce_erl:choice([{Send, []}, {Drop, []}]).
```

TODO list / future work & wishes

- Emulate BIFs for monitoring and linking
- Emulate gen_fsm/gen_server semantics: test code written for them as-is
- Add state verification: after every execution step, verify state of all processes.
- Failure output: system state dump should be easier to read
- Lots of McErlang integration work remains
- Parse transform to ease McErlang use
- Visualization: draw MSC diagram of failing test case
- Visualization: 2D animation of failing test case
- Implement more protocols: alternating bit, leader election, Paxos, let your imagination run wild....

In summary



- Credit: Orb Books cover, 1997 (?)

msgdropsim summary

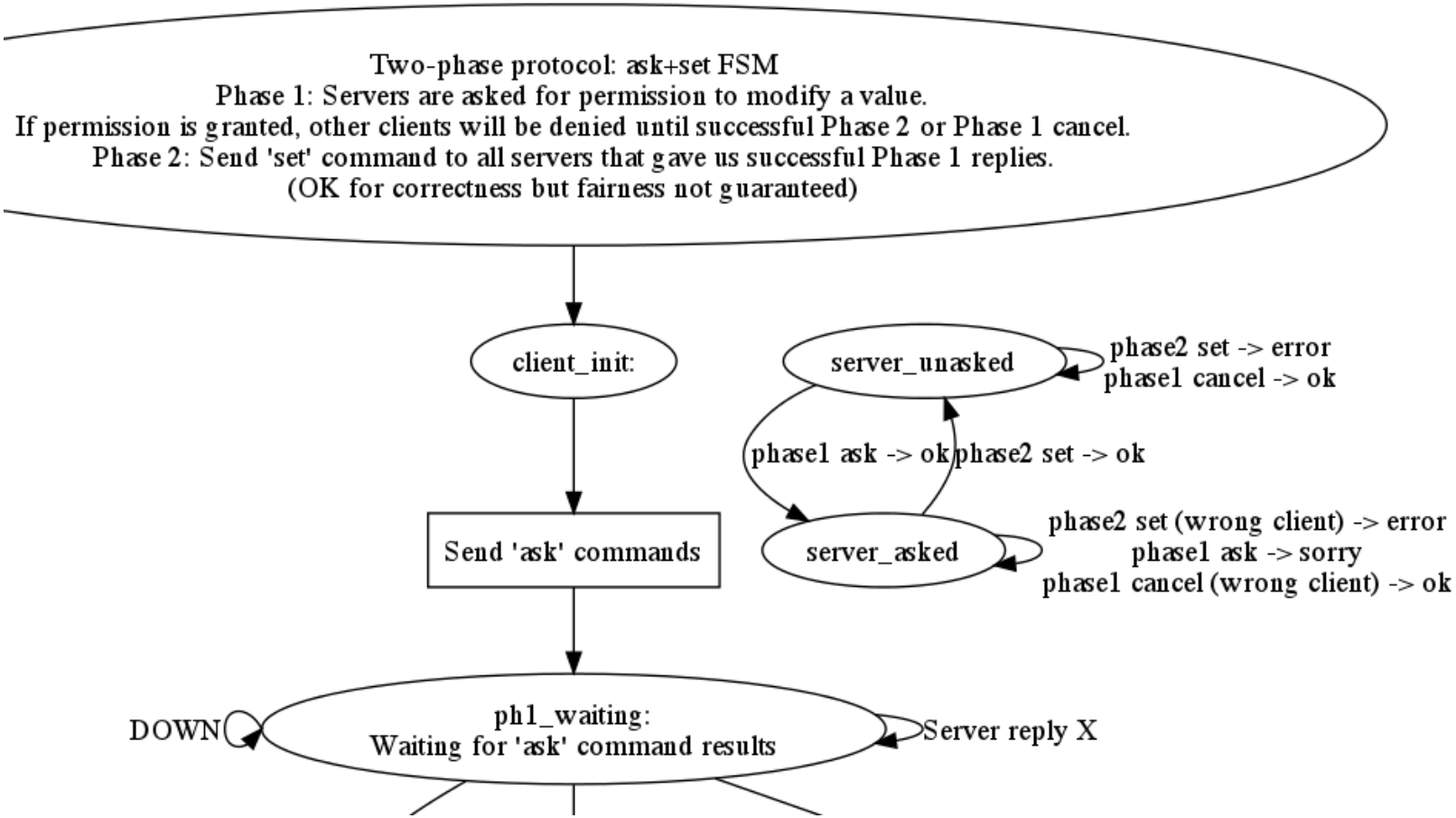
- YES: Test message-passing code with random message drops and scheduling (un)fairness
  - *Quite successful at finding weird corner cases*
- PARTIAL: Integration with McErlang for exhaustive state exploration
  - *If you can set it up correctly...*
  - *... **extremely successful** at finding all bugs*
- msgdropsim has been very helpful in Basho product R&D

The end

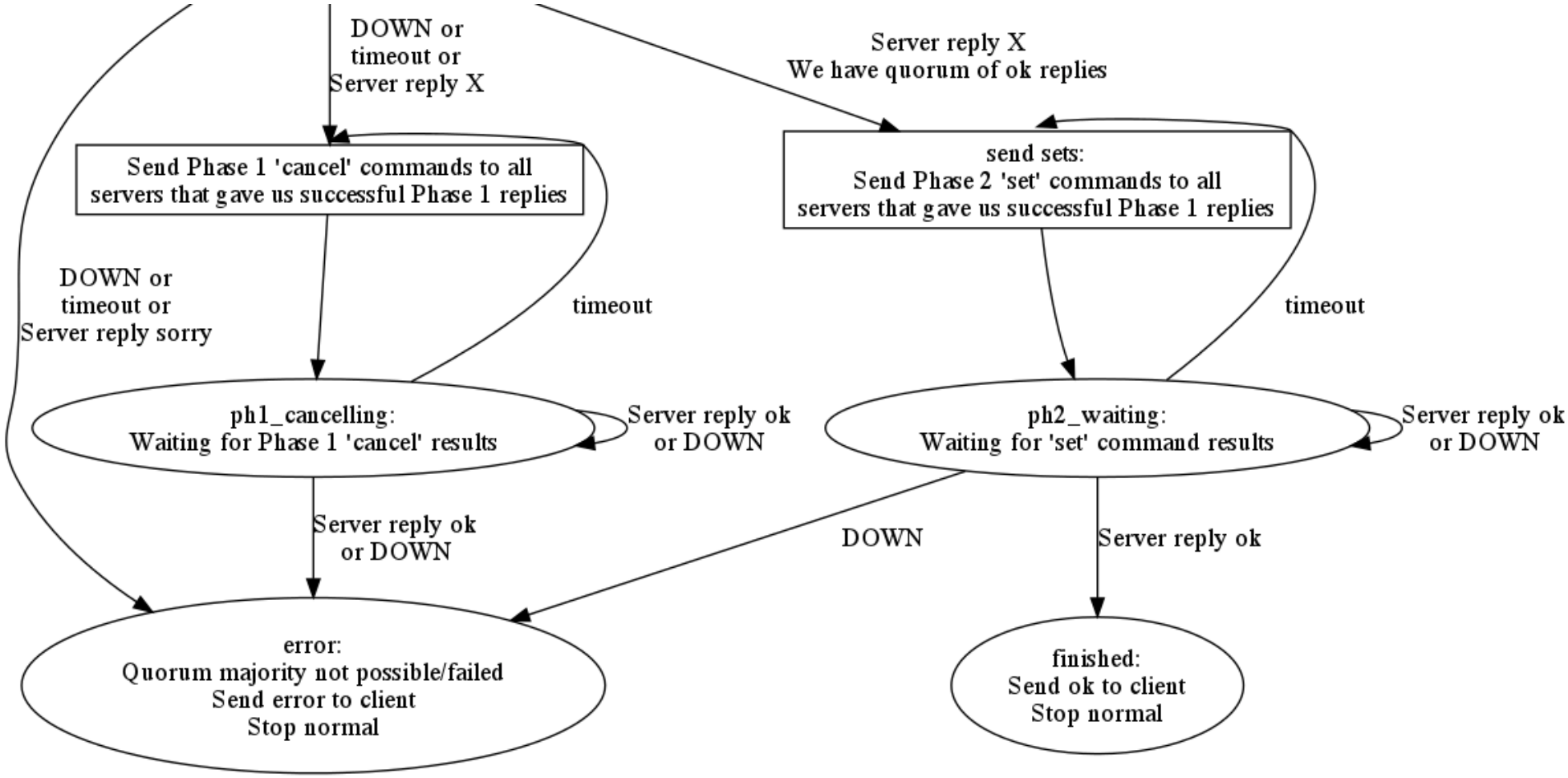
- Any questions?
- https://github.com/slfritchie/msgdropsim
- scott@basho.com

Backup material

Distributed counter protocol FSM 1 of 2



Distributed counter protocol FSM 2 of 2



Distributed counter in action

```
> eqc:quickcheck(eqc:numtests(5000,
      slf_msgsim_qc:prop_simulate(
          distrib_counter_2phase_sim, []))).
[...]
OK, passed 5000 tests

50.96% at_least_1_msg_dropped
clients      Min: 1 Max: 9  Avg: 4.955  Total: 24773
servers      Min: 1 Max: 9  Avg: 5.015  Total: 25075
sched steps  Min: 0 Max: 1959 Avg: 144.6  Total: 722961
crashes      Min: 0 Max: 0  Avg: 0.000e+0  Total: 0
# ops        Min: 0 Max: 17  Avg: 4.184  Total: 20919
# emitted    Min: 0 Max: 16  Avg: 1.278  Total: 6389
# ph1 t.out  Min: 0 Max: 17  Avg: 0.7982  Total: 3991
# ph1 q.fail Min: 0 Max: 16  Avg: 1.983  Total: 9917
# ph2 t.out  Min: 0 Max: 2  Avg: 0.1244  Total: 622
msgs sent    Min: 0 Max: 477  Avg: 52.96  Total: 264822
msgs dropped Min: 0 Max: 1331  Avg: 5.758  Total: 28791
timeouts     Min: 0 Max: 1160  Avg: 4.902  Total: 24510
```

Debugging light-hours away

- Mars Pathfinder, 1997
- "reset bug"
- Process scheduler priority inversion
  - *watchdog bark -> reset -> data loss*
- Debugged using exact same hardware on Earth
- SPIN verification tool: ~25 lines of code

You don't know...

- ... where the flaw is
- ... when the flaw is executed
- ... know when a symptom appears
- ... how much time elapsed between flaw execution & symptom
  - *Insert bad hand grenade analogy*
- ... which log file data to gather
- ... ?

Scheduler code

```
run_scheduler_with_tokens(Tokens, Schedule) ->
    try
        lists:foldl(fun(Name, S) ->
                        consume_scheduler_token(Name, S)
                    end, Schedule, Tokens)
    catch throw:{receive_crash, NewS} ->
            NewS
    end.

consume_scheduler_token(ProcName, S)
  when is_atom(ProcName) ->
    P = fetch_proc(ProcName, S),
    consume_scheduler_token(P, S, 0).
```

Simulator scheduler record

```
-record(sched, {
          step = 0,      % Scheduling step #
          stop_step,     % Debugging: step # to stop
          numsent = 0,   % Number of messages sent
          tokens,        % Process scheduler tokens
          crashed = [], % Processes that have crashed
          procs,         % key=proc name, val=#proc{}
          trace = [],    % trace events
          utrace = [],   % user trace events
          partitions,    % network partition spec
          delays,        % message delay spec
          module,        % implementation module name
          options        % implementation options list
         }).
```

Simulator process record

```
-record(proc, {
          name,            % All procs have a name
          state,           % proc private state
          mbox,            % incoming mailbox
          outbox,          % outgoing messages + delay
          next,            % next execution type
          recv_gen,        % For receive without timeout
          recv_w_timeout   % For receive with timeout
         }).
```

Receive loop: caller and implementation

```
    erlang:put({?MODULE, sched}, S),
    erlang:put({?MODULE, self}, P#proc.name),
    RecvVal = receive_loop(P0#proc.mbox, RecvFun,
                           P0#proc.state),
    %% ....

receive_loop([], _Fun, _St) ->
  no_match;
receive_loop([{imsg, _Sender, _Rcpt, H} = IMsg|T],
             RecvFun, ProcState) ->
  try
      Res = RecvFun(H, ProcState),
      {IMsg, Res}   % Tell caller which imsg we picked
  catch
      error:function_clause ->
          receive_loop(T, RecvFun, ProcState);
      X:Y ->
          {error, H, X, Y}
  end.
```

Receive loop: post-call imperative pseudo-code

```
{IMsg, {ReceiveType, NextStateName, NewProcS}} ->
    %% RecFun0 = name/fun for next state
    %% NewProcS = new process internal state term
    add_trace({recv, Step, message details...}),
    delete_message(IMsg),
    store_next_state(ReceiveType, NextStateName),
    increment_step();
no_match ->
    do_nothing();   % Token will not be consumed
{error, Msg, X, Y} ->
    add_trace({process_crash, ProcName, ...}),
    remove_proc_from_scheduler(),
    increment_step();
```

Sending messages

- Dictionary of {Sender, Recipient} => [{StepStart, StepStop, Delay}]
- One dictionary for network partitions (delay unused)
- One dictionary for message delays

```
if
  is_integer(Delay) ->
    add_trace({delay, ..., {num_rounds, Delay}}),
    queue_message(Msg, {t_delay, Delay});
  DropMessage ->
    add_trace({drop, ...});
  true ->
    queue_message(Msg, t_normal)
end
```

Receiving messages ... what about timeouts?

- Try running scheduler with scheduler token list
- Did scheduler step # advance?
  - *Yes: Run scheduler again (i.e., loop!)*
  - *No:*
    - Let **Ps** = all processes with `recv_timeout` state flavor
    - `[send_timeout(P) || P <- Ps]`

System & user trace lists

- Maintained as simple lists inside the `#sched` record
- System trace log is maintained automaticallly
- User code is annotated, e.g.,
- `slf_msgsim:add_utrace(Meaningful::term())`

* help? contents? restart? slide 2/63