What's the Deal with Erlang?

Introduction to Erlang for Ruby Users of Minnesota

November 26, 2007

Scott Lystig Fritchie

Erlang Hacker

<slfritchie@snookles.com>

Bona Fides

- UNIX sysadmin since 1986
- Hacked UNIX systems since 1988
- Former lives: Desktop app developer, UNIX sysadmin, IP network admin and architect
- Returned to software development: currently using Erlang, C, Tcl, Python, Awk, ...
  - Best tool for the job
  - Distributed, fault-tolerant, highly-available apps running in telcos in Japan, China, the Americas

Don't Know Much Ruby...

- But I've visited Matsumoto's hometown: Matsue, Shimane Prefecture, Japan

Downtown Matsue

What This Talk Isn't About

- Er-Lang Shen, a Chinese god with a truth-seeing third eye.
- Agner K. Erlang, Danish mathematician

What This Talk Isn't About

- Creating a “Web App” in 53 seconds
  - Not a develop-faster-than-Ruby-on-Rails talk.
  - Will focus on reliability and scalability
- Toss Ruby into File 13, Erlang is The Way!
  - Please wear your “right tool for the job” hat
- All-you-need tutorial on Erlang
  - Not enough time
  - Check my post to ruby.mn list for pointers

Erlang History: 1985-1990

- Telephony equipment software sucks
  - Assembly, Pascal, PLEX, EriPascal, ...
  - Too much concurrency, hardware too slow
- Language survey: implement a POTS switch
  - Same control s/w, any language they could find
  - Ada, Chill, Euclid, Smalltalk, PFL, C++ (?), ...
    - Over 20 languages included survey
- Result: None are good enough for future Ericsson telephony products.

Original Requirements

- Handle very large # of concurrent activities
- Soft real-time
- Distribution with heterogenous computers
- Smooth interaction with hardware drivers
- Very large software systems, e.g. > 1M LOC
- Continuous operation: **years**, not days
- Reconfiguration & upgrades without stopping
- Fault tolerance: hardware **and** software faults

Pthreads are evil

- “11 out of 10 people cannot handle threads.”
  - source: Andre Pang
- “If POSIX threads are a good thing, perhaps I don't want to know what they're better than.”
  - source: Rob Pike

My apologies for any misattribution.

Threads Are Evil

- Threads Cannot Be Implemented As a Library
  - Hans-J. Boehm, ACM PLDI '05, Chicago, IL
  - Summary: The POSIX threads spec is fatally flawed and cannot be fixed by any library.
- The Problem with Threads
  - Edward A. Lee, Tech Report UCB/EECS-2006-1
  - Summary: Languages must support concurrency and nondeterminism from the ground level.
- Transactional memory => shared memory!

Shared Memory is Evil

- All research today on shared memory...
  - ... assumes shared memory
- How long can you afford to wait for industry to deliver N cores/machine (then pay for it)?
  - N = 16, N = 64, N = 256, ...
- If you can't wait, distributed processing is your only choice.
  - Software-based shared memory is **hard**.

Meeting the Requirements 1

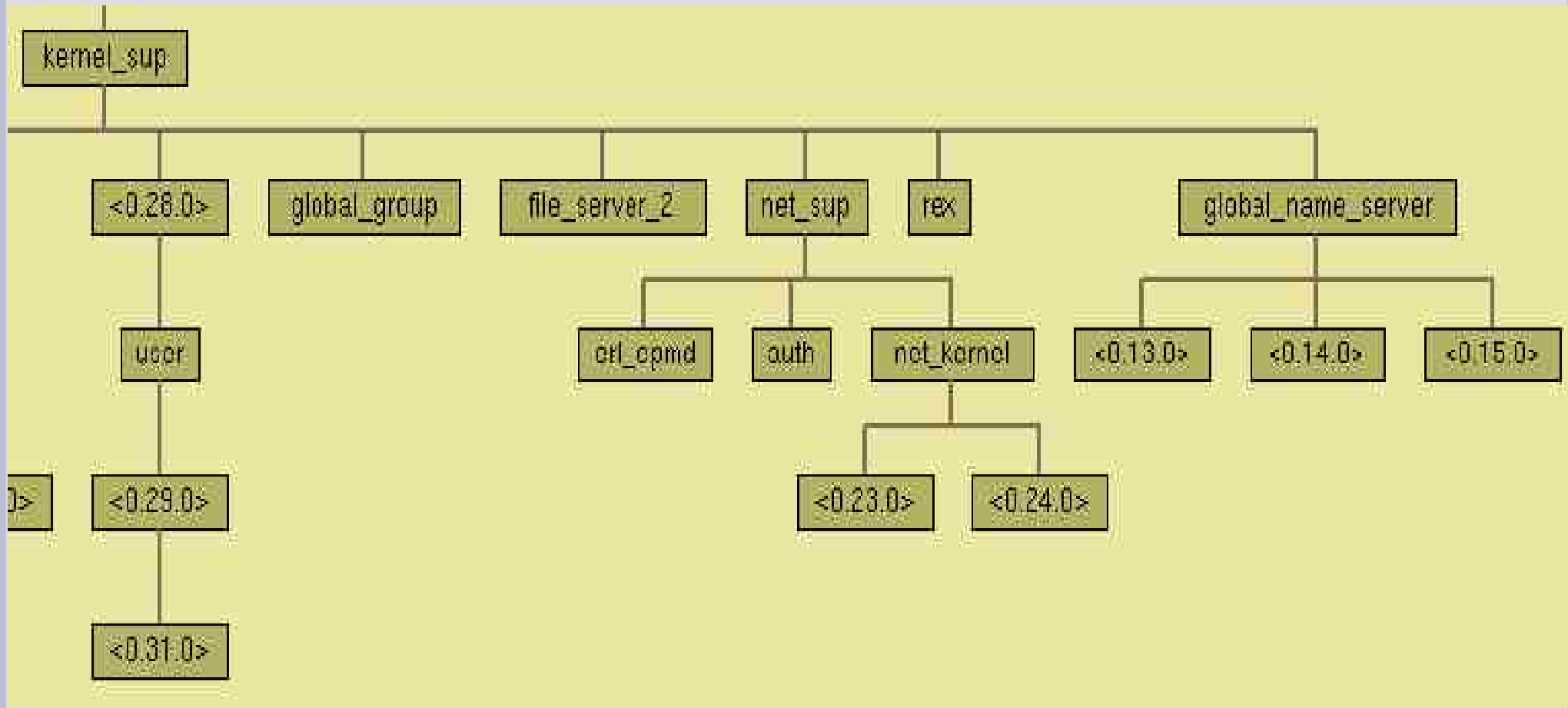
- Concurrency-oriented prog. language (COP)
  - "Thread" thingies are very, very cheap
    - Whenever possible, use 1 thread per concurrent task
  - Process semantics: **shared nothing**
- Preemptive process scheduling
  - Simple round-robin scheduling
  - Nowadays, 1 Erlang process scheduler per CPU or CPU core
- All communication via message passing
  - Message passing demo

Meeting the Requirements 2

- Fault tolerance
  - Process linking: processes share fate
    - Transitive fate-sharing: A <-> B <-> C
      - If A dies, then both B and C die (for the same reason)
    - Or fate monitoring: create supervisor hierarchies
      - Supervisors: start workers, monitor workers, re-start workers
      - Workers: do useful computation
  - Default error handler: fatal crash
    - “Let it crash” paradigm makes coding simpler
    - Someone else will restart you...
  - Message passing: asynchronous model

Meeting the Requirements 2b

- Part of “kernel” application supervisor tree



Meeting the Requirements 3

- Distributed, Heterogenous, Scalable
  - Virtual machine: UNIX, Win, MacOS, Vxworks
  - Message passing hides CPU endian-ness
  - Intra-VM and inter-VM messaging: same syntax
- Fault tolerance
  - Process isolation
  - Health monitoring via “link” (“monitor” = 1-way)
  - Supervisors provide restart strategies:
    - restart child, restart all children, restart some children

Meeting the Requirements 4

- Continuous operation
  - Imagine your s/w running for the next 6 years...
  - Hot code upgrade
    - VM supports 2 simultaneous running code versions
  - And/or use “controlled crashing” for system reconfiguration
    - Kill a process ... and let the supervisors do their job
  - Elevator demo

Meeting the Requirements 5

- Soft real-time
  - Preemptive scheduling
  - GC on per-process heaps
  - Behind-the-scenes helper threads to avoid blocking I/O (e.g. file system interaction)
  - Optional process priority mechanism
- Prototyping
  - Dynamic type system (contrast to Haskell, ML)
  - Declarative style
  - Runtime linking and module loading

Erlang Suitable For Web Apps?

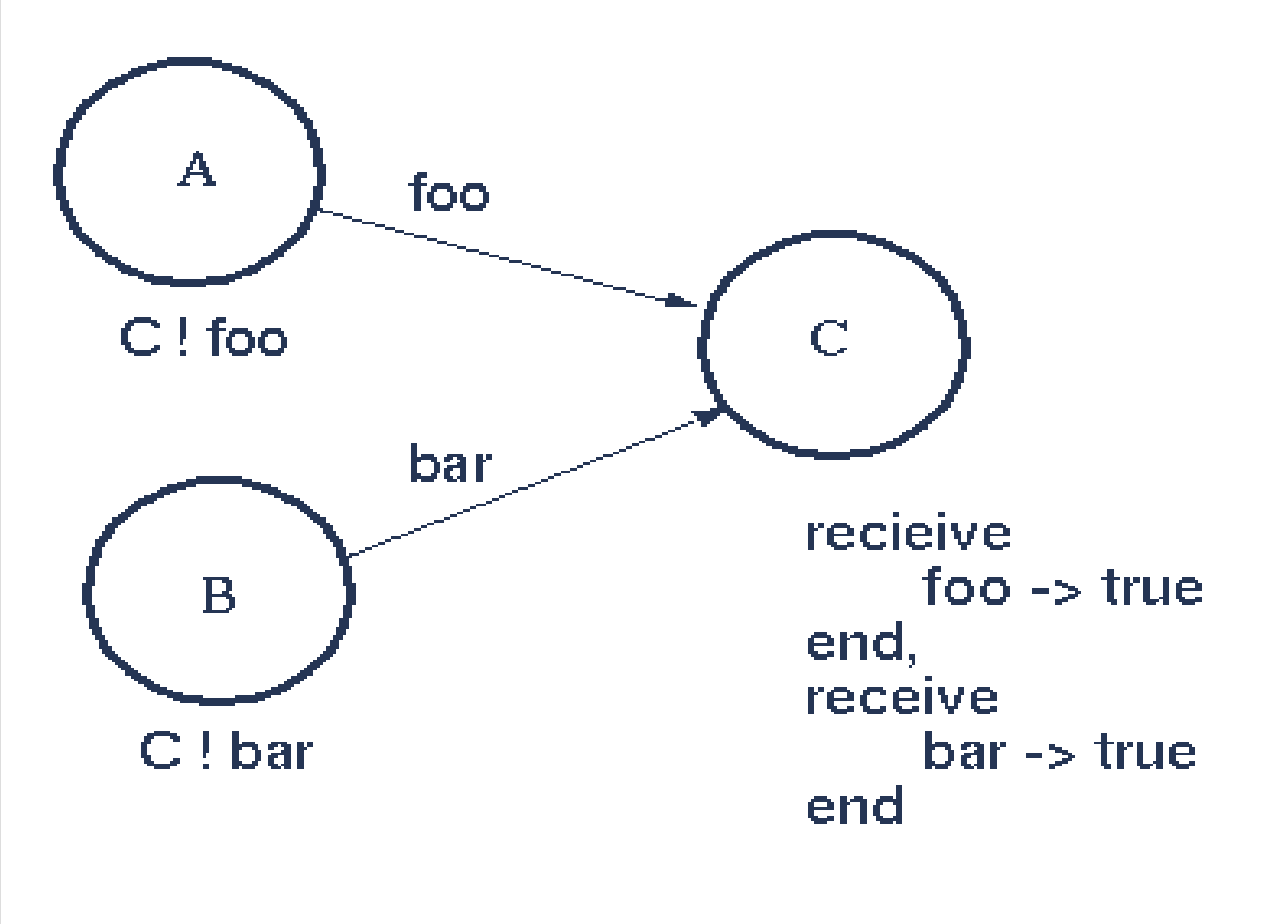
- Certainly.
  - Who doesn't need high availability, fault tolerance, scalability?
- But Erlang's design & history hasn't focused on “the HTTP client/server boundary”.
  - Not my specialty, either, sorry.
- Don't let that stop you.
  - Mnesia, ODBC driver, Yaws, ErlyWeb, ongoing efforts with Ajax'y techniques and Adobe Flex, ...

Joe Armstrong, Guest Speaker

- http://www.esug.org/data/ESUG2006/pres/erlang_in_11_minutes.pdf

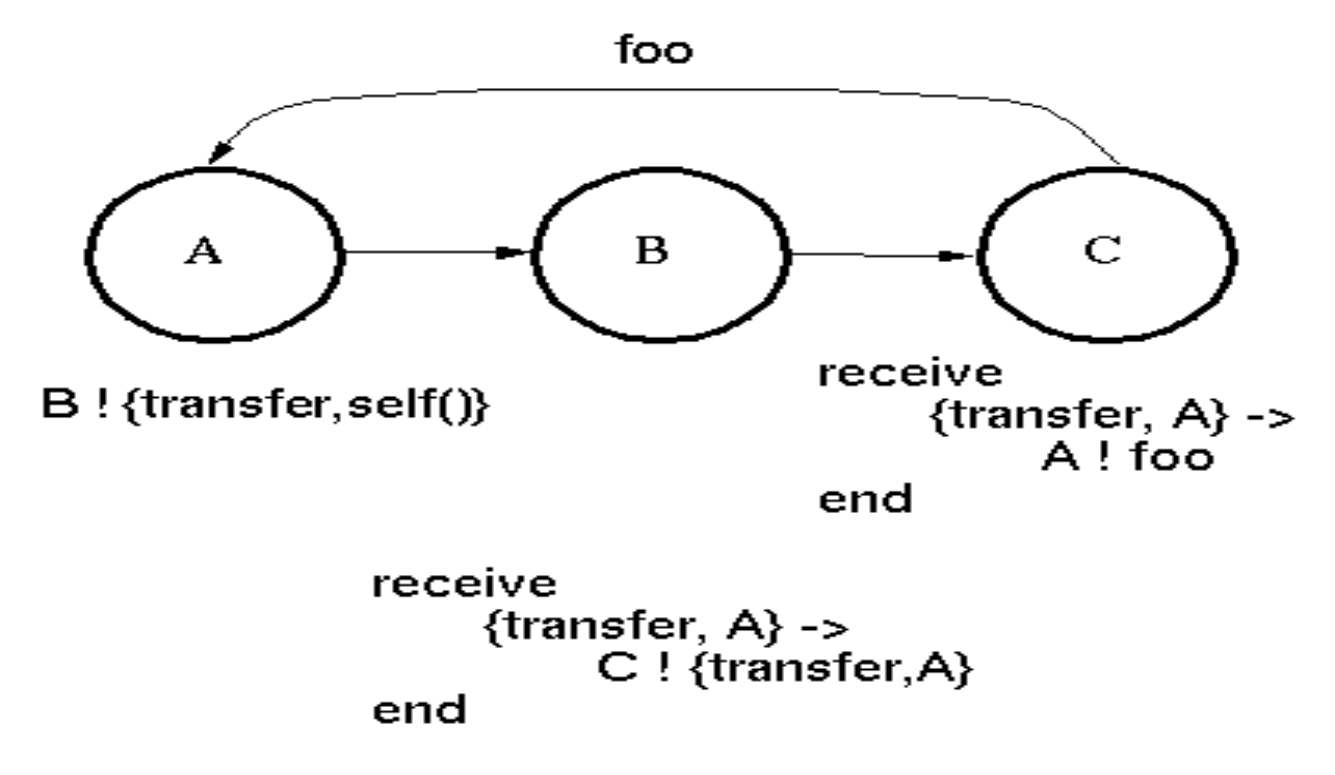
Selective Receive

- Each process has a single mailbox
- Use patterns to choose first message matching the pattern.



Passing a Pid in a Message

- Client/server: the server's reply can be sent by anyone.



Anything Like Ruby Gems?

- Um, er, ... no.
  - Ruby's community shines here
- CEAN: Comprehensive Erlang Archive Network
  - http://cean.process-one.net/
- Faxien and Sinan
  - http://code.google.com/p/faxien
  - http://code.google.com/p/sinan
- Merger in 2008? Maybe, perhaps.

Santa's Concurrency Problem

- Santa sleeps until either:
  - All 9 reindeer return to North Pole from vacation
  - Any 3 of 10 elves request to meet with him
- If awakened by:
  - Reindeer: Go deliver toys, then release reindeer
  - Elves: Attend toy R&D meeting, then dismiss elves
- Santa gives priority to reindeer (who hate snow?)
- Pick a solution language, any language....
- Problem by J.A. Trono, code by Richard O'Keefe
  - http://www.cs.otago.ac.nz/staffpriv/ok/santa/index.htm

Santa's Solution 1

- Module preliminaries, top-level functions.

```
-module(santa).
-author('ok@cs.otago.ac.nz'). % Richard A. O'Keefe
-export([start/0, start/3]).


start() ->
    start(9, 10, 1).

start(NumReindeer, NumElves, WaitSecs) ->
    Santa = spawn_link(fun () -> santa() end),
    Robin = spawn_link(fun () -> secretary(Santa, reindeer, 9) end),
    Edna  = spawn_link(fun () -> secretary(Santa, elves,    3) end),
    [spawn_worker(Robin, "Reindeer ", I, " delivering toys.\n", WaitSecs)
     || I <- lists:seq(1, NumReindeer)],
    [spawn_worker(Edna,  "Elf ",      I, " meeting in the study.\n", WaitSecs)
     || I <- lists:seq(1, NumElves)].
```

Santa's Solution 2

- Worker: elves & reindeer are about the same

```
spawn_worker(Secretary, Before, I, After, WaitSecs) ->
    Message = Before ++ integer_to_list(I) ++ After,
    spawn_link(fun () -> worker(Secretary, Message, WaitSecs) end).


worker(Secretary, Message, WaitSecs) ->
    receive after random:uniform(WaitSecs*1000) -> ok end,  % random delay
    Secretary ! self(),             % send my PID to the secretary
    Gate_Keeper = receive X -> X end,    % await permission to enter
    io:put_chars(Message),          % do my action
    Gate_Keeper ! {leave,self()},   % tell the gate-keeper I'm done
    worker(Secretary, Message, WaitSecs). % do it all again
```

Santa's Solution 3

- The magic sauce: secretary processes

```
secretary(Santa, Species, Count) ->
    secretary_loop(Count, [], {Santa,Species,Count}).

secretary_loop(0, Group, {Santa,Species,Count}) ->
    Santa ! {Species,Group},
    secretary(Santa, Species, Count);
secretary_loop(N, Group, State) ->
    receive PID ->
        secretary_loop(N-1, [PID|Group], State)
    end.
```

Santa's Solution 4

- Santa is quite straightforward now.

```
santa() ->
    {Species,Group} =
         receive                        % first pick up a reindeer group
            {reindeer,G} -> {reindeer,G} % if there is one, otherwise...
         after 0 ->
            receive                     % wait for reindeer or elves,
                {reindeer, _} = T -> T
              ; {elves,    _} = T -> T
            end
      end,
    case Species
      of reindeer -> io:put_chars("Ho, ho, ho!  Let's deliver toys!\n")
       ; elves    -> io:put_chars("Ho, ho, ho!  Let's meet in the study!\n")
    end,
    [PID ! self() || PID <- Group],% tell them all to enter
    [receive {leave,PID} -> ok end  % wait for each of them to leave
      || PID <- Group],
    santa().
```

Santa's Lessons

- Use one process per concurrent activity.
- Selective message receiving makes life easy.
- Use process links so all processes share fate.
- What if Santa were a Web cache? (think Squid)
  - What if each cached URI/object were a process?
  - What if each cached URI/object could decide for itself to implement its own:
    - Time-to-live/replacement policy? Refresh itself? Disk/RAM?
  - Starts to sound quite object'ish ... but like the original Smalltalk, not like OOP languages today.

Newbie Traps and Pitfalls

- Variables **are not** variable: they're bound once!
- Not using Emacs/ErlIDE/smart indenting editor
- Not using shell for quick edit/compile/test cycle.
- Errors/exceptions/exits in the shell kills stuff.
- Using too few processes.
- Not learning to decipher nested Erlang terms.
  - Error messages usually very helpful, gotta read 'em.
- Not knowing the Erlang/OTP docs structure.
  - Browser demo.

Unit Tests with Erlang Code

- Pattern matching is your friend: code exactly what you expect to get, anything else is a bug.

```
first_256(File) ->
    {ok, F} = file:open(File, [read]),
    {ok, Data} = file:read(F, 256),
    file:close(F),
    256 = length(Data),
    Data.
```

```
4> foo:first_256("/etc/termcap").
"######## TERMINAL TYPE DE [...]"
5> foo:first_256("/etc/motd").
** exited: {{badmatch,eof},              <-- Reason: got 'eof'
            [{foo,first_256,1},         <-- Stack trace (incl.
             {erl_eval,do_apply,5},         shell internal stuff)
             {shell,exprs,6},
             {shell,eval_loop,3}]} **
```

QuickCheck: Testing Done Right?

- Free in Erlang or Haskell, commercial in Erlang (with many enhancements)
  - See also: *Why Programs Fail*, Andreas Zeller
- Specify properties (think invariants), let QuickCheck generate random data to test those properties.

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

Completely Random Test Data Isn't Good Enough

- Fuzz testing (ask Mr. Google) experience says that 100% random data isn't effective.

```
gen_ymd() ->
    ?LET({Year, Mon}, {gen_year(), gen_month()},
         {Year, Mon, gen_dayinmonth(Year, Mon)}).

gen_year() ->
    choose(1500, 2300).

gen_month() ->
    choose(1, 12).

gen_dayinmonth(Y, M) ->
    oneof(
      [choose(1,28)] ++
      [choose(1,31) || lists:member(M,[1,3,5,7,8,10,12])] ++
      [choose(1,30) || lists:member(M,[4,6,9,11])] ++
      %% NOTE: we're trusting calendar:is_leap_year/1
      [choose(1,29) || (M==2) and calendar:is_leap_year(Y)]).
```

Testing a Simple Database Server with QuickCheck

- Simple memcached-like API
- Right now has only one operation: set
- Keys are integers 1..50
- Values are integers 1..10
- Server will “crash” if:
  - magic key Ka = value Ka (e.g. key 3 = val 3)
  - magic key Kb = value Kb
  - magic key Kc is set to anything
- QuickCheck demo with shrinking.