The nine nines

mats.cronqvist@klarna.com

Ruminations on tools and strategies.

With boring anecdotes!

rumination

n 1: a calm lengthy intent consideration [syn: contemplation, reflection, reflexion, musing, thoughtfulness]
2: (of ruminants) chewing (the cud); "ruminants have remarkable powers of rumination"
3: regurgitation of small amounts of food; seen in some infants after feeding

this talk

- the nine nines
- debugging in the telecom world
- debugging Erlang

the claim

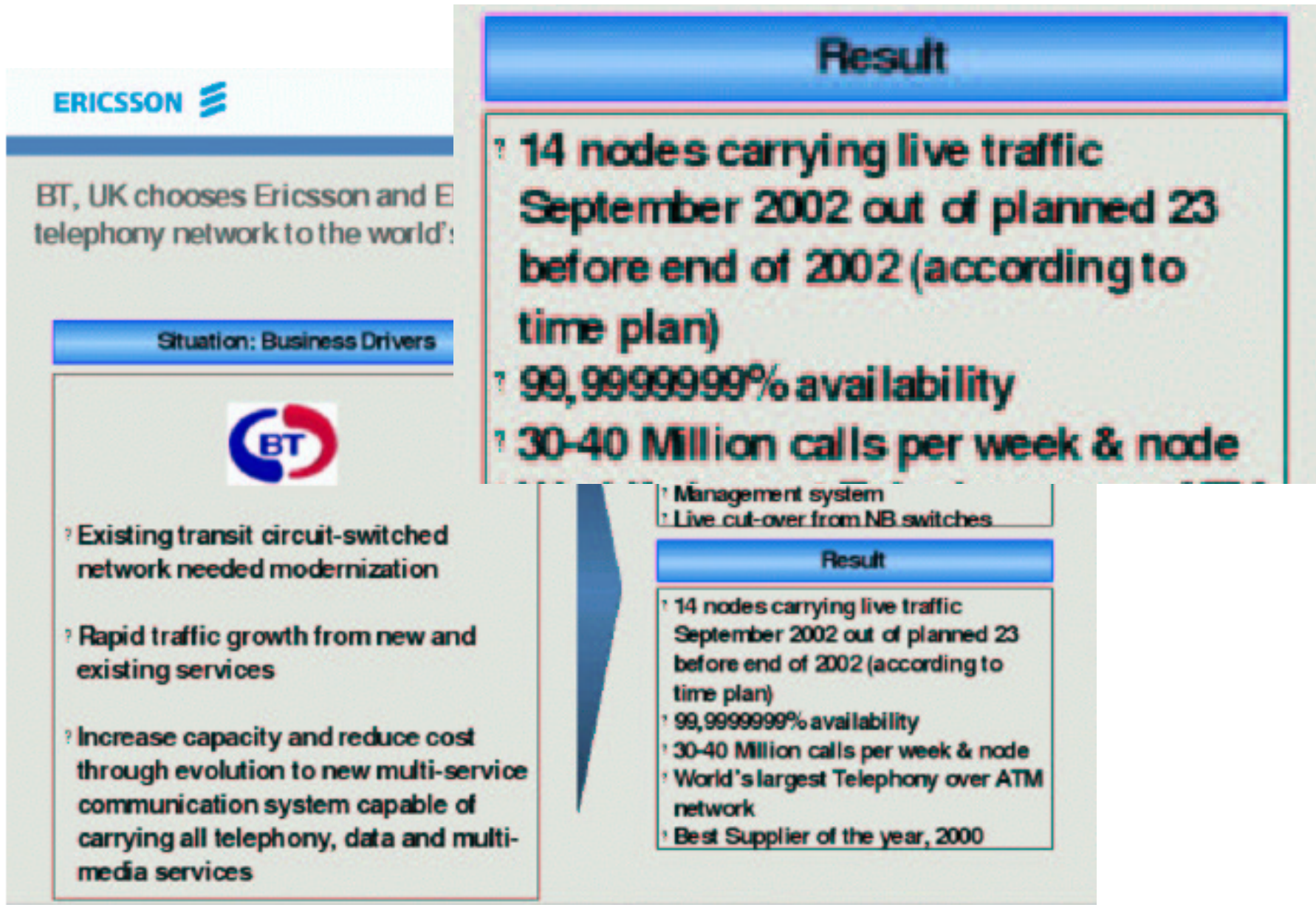
"99.9999999% reliability (9 nines) (31 ms. year!)"

http://ll2.ai.mit.edu/talks/armstrong.pdf

“The AXD301 has achieved a NINE nines reliability (yes, you read that right, 99.9999999%). Let’s put this in context: 5 nines is reckoned to be good (5.2 minutes of downtime/year). 7 nines almost unachievable … but we did 9.”

http://www.pragprog.com/articles/erlang

the evidence



the reaction

"Before a power failure drained the USV the server this blog has been running on had a uptime of about 420 days. So it had NO downtime in a year. Does this mean 100% reliability? No."

"So obviously Erlang is not the only thing which makes an AXD 301 tick. I assume there is also a lot of clever reliability engineering in the C code and in the hardware."

"There is no need use 99.9999999 % which ring so hollow."

http://blogs.23.nu/c0re/2007/08/

what I remember (a.k.a. The Truth)

(I was system architect/troubleshooter)

- The customer (British Telecom) claimed nine nines service availability integrated over about 5 node-years.
- As far as I know, no one in the AXD 301 project claimed that this was normal, or even possible.
- For the record, Joe Armstrong was not part of the AXD 301 team.

the claim is pretty bogus...

- there was much more C than Erlang in the system
- there were no restarts and no upgrades
- the functionality was very well defined

nevertheless...

- the system was very reliable
- compared to similar systems, it was amazingly reliable

I have been unable to find any publicly available reference to this.

An ancdote will have to do!

the Dark Side

Embedded system.
Multi-million lines of C++.
The disks were too small for core files.
100s of units deployed.

...but...
The network worked.

why was it so stable?

- high quality programmers? no...
- superior system architecture? no...
- realistic project management? yes.
  - testing and development were close
- highly stable infrastructure? yes.
  - solaris/beam/OTP
- properties of the Erlang language? yes.
  - highly productive (small team)
  - no memory management
  - selective receive
  - debugging

something rotten in Denmark...

- Embedded system running OTP R5.
- Live in Denmark.
- There was no way to log into the CPU.
- There was no way to load new code.
- There was no usable application debugging tool.
- You could physically connect a terminal.

The node got overloaded after 90 days.
A tech traveled there and rebooted every 90 days.

...tracing...

- Wrote a one-liner...
- ...that ran a one-minute trace and wrote to a file.
- Sent it to the Danish tech by mail...
- ...who ran it by pasting it into a shell...
- ...before and after the reboot...
- ...and emailed the files to me (base-64 encoded)

...saves the day!

Wrote a comparison profiler.
Compare the average execution time for each function, before and after the reboot.

ets:lookup/2
was 100 times slower before the reboot.

the answer

The hash function was broken for bignums.

the point...

Debuggability is a property of a system

In a distributed system, fail-stop bugs are easy

the 3 kinds of bugs

- It crashes “randomly”
- It uses too much of a resource
- It gives the wrong answer

strategies

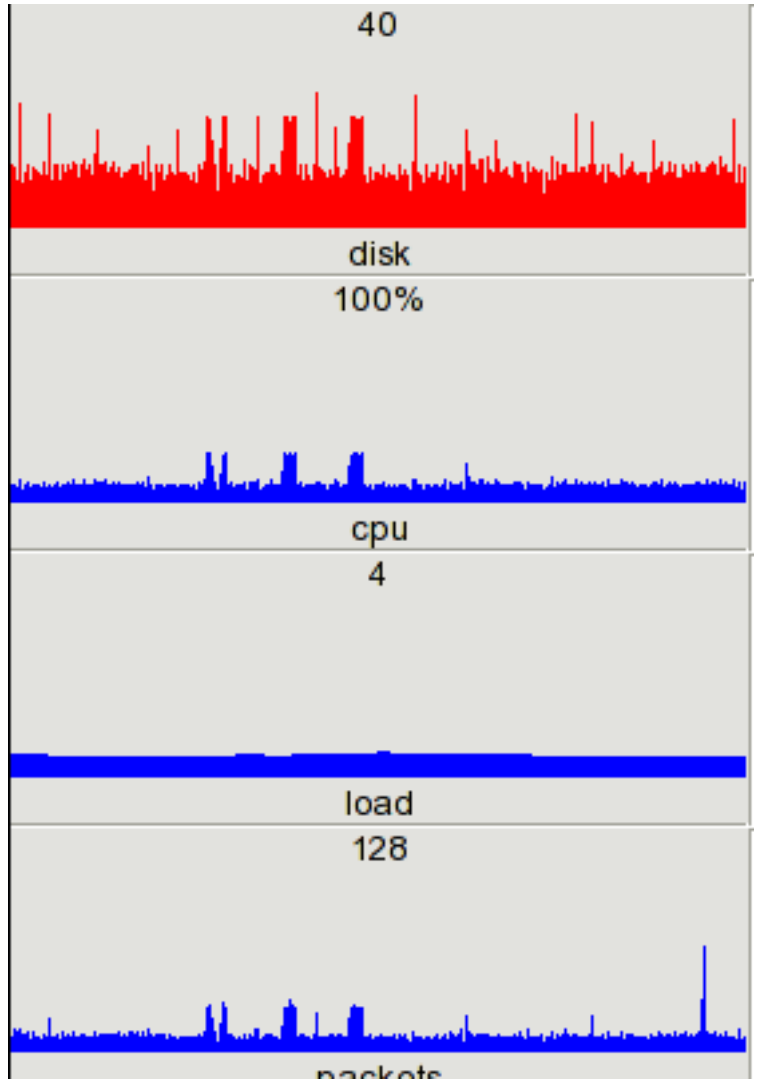
	Monitoring	Narrowing
crashes	logging	context
performance	real-time logging	process → function
wrong result	contracts (test cases)	context

polling

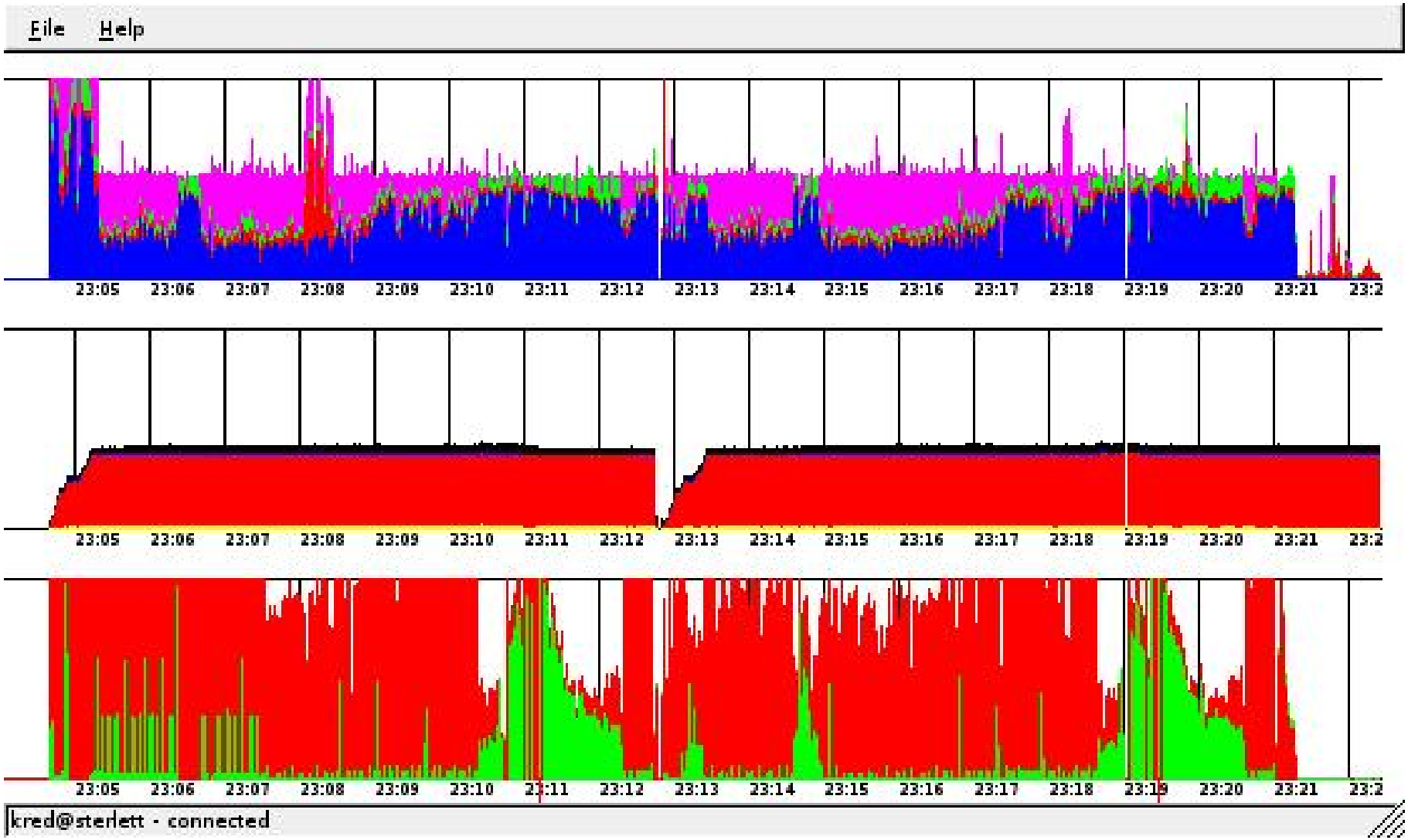
The Erlang VM has many info methods

- erlang:memory
- erlang:system_info
- erlang:statistics
- erlang:process_info
- inet:i().

solaris perfmeter



gperf


File
Help
23:05
23:06
23:07
23:08
23:09
23:10
23:11
23:12
23:13
23:14
23:15
23:16
23:17
23:18
23:19
23:20
23:21
kred@sterlett - connected

unix top

top - 13:54:15 up 24 days, 2:59, 9 users, load average: 0.15, 0.42, 0.49
Tasks: 192 total, 5 running, 185 sleeping, 0 stopped, 2 zombie
Cpu(s): 7.0%us, 3.3%sy, 0.0%ni, 89.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 3106248k total, 2978996k used, 127252k free, 67844k buffers
Swap: 0k total, 0k used, 0k free, 2066100k cached

PID %MEM VIRT SWAP RES CODE DATA SHR nFLT nDRT S PR NI %CPU COMMAND
 8748 4.5 181m 44m 136m 1544 174m 2156   15   0 S 20 0   1 beam.smp
 2842 3.6 307m 196m 110m   44 236m 27m   0   0 R 20 0   7 firefox-bin
29380 3.0 395m 304m 91m   48 225m 30m 1095   0 S 20 0   0 amarok
24748 2.6 171m 93m 77m 31m 113m 12m   7   0 S 20 0   0 chrome

dtop

dtop:start()

```
--------------------------------------------------------------------------
kr@sterlett size: 4(45)M, cpu%: 0(27), procs: 39, runq: 0, 13:57:19
memory[kB]: proc 591, atom 300, bin 40, code 1905, ets 128

pid      name              current             msgq mem cpu
<0.49.0> prfTarg           prfPrc:pidinfo/2       0  31   0
<0.40.0> group:server/3    group:get_line1/3      0  11   0
<0.46.0> dtop              prfHost:loop/1         0  21   0
<0.28.0> user_drv:server/2 user_drv:server_loo    0  17   0
<0.50.0> erlang:apply/2    file_io_server:serv    0   8   0
<0.51.0> erlang:apply/2    file_io_server:serv    0   8   0
```

Interrupts

The Erlang VM has 2 interrupt mechanisms

- erlang:trace/3 (redbug)
- erlang:system_monitor/2 (watchdog)

UNIX strace

STRACE(1) STRACE(1)

NAME
strace - trace system calls and signals

SYNOPSIS
...

DESCRIPTION
Strace intercepts and records the system calls which are called by a process and the signals which are received by a process. The name of each system call, its arguments and its return value are printed...

dbg - cons

- dbg is too hard to use correctly
  - very tricky API
  - not safe

needed:

- much simpler API
- safer
  - disallow bad patterns
  - terminate if something bad happens

the Frankfurter

```
Pi = fun(P) when pid(P) -> case process_info(P, registered_name) of[] -> case process_info(P, initial_call) of {_, {proc_lib,init_p,5}}-> proc_lib:translate_initial_call(P); {_, MFA} -> MFA; undefined ->unknown end; {_,Nam} -> Nam; undefined -> unknown end; (P) whenport(P) -> {name,N} = erlang:port_info(P,name), [Hd|_] =string:tokens (N," "), Tl =lists:reverse(hd(string:tokens(lists:reverse(Hd),"/"))),list_to_atom(Tl); (R) when atom(R) -> R; ({R,Node}) when atom(R),Node == node() -> R; ({R, Node}) when atom(R), atom(Node) -> {R,Node} end, Ts = fun(Nw) -> {_,{H,M,S}} =calendar:now_to_local_time(Nw), {H,M,S,element(3,Nw)} end, Munge =fun(I) -> case string:str(I, "Return addr") of 0 -> casestring:str(I, "cp = ") of 0 -> []; _ -> [_, C|_] =string:tokens(I,"()+"), list_to_atom(C) end; _ -> case string:str(I,"erminate process normal") of 0 -> [_, C|_] =string:tokens(I,"()+"), list_to_atom(C); _ -> [] end end end, Stack= fun(Bin) -> L = string:tokens( binary_to_list( Bin),"\n"),{stack,lists:flatten(lists:map(Munge,L))} end, Prc = fun(all) ->all; (Pd) when pid(Pd) -> Pd; ({pid,P1,P2}) when integer(P1),integer(P2) -> c:pid(0,P1,P2); (Reg) when atom(Reg) -> case whereis(Reg) of undefined -> exit ({rdbg, no_such_process, Reg}); Pid when pid(Pid) -> Pid end end, MsF = fun(stack, [{Head,Cond,Body}])-> [{Head,Cond,[{message,{process_dump}}|Body]}]; (return, [{Head,Cond,Body}]) -> [{Head,Cond, [{return_trace}|Body]}]; (Head,[{_,Cond,Body}]) when tuple(Head)-> [{Head,Cond,Body}]; (X,_) ->exit({rdbg,bad_match_spec,X}) end, Ms = fun(Mss) -> lists:foldl(MsF,[{'_',[],[]}], Mss) end, ChkTP = fun({M,F}) when atom(M), atom(F),M/='_', F/='_' -> {{M,F,'_'},[],[global]}; ({M,F,MS}) when atom (M),atom(F), M/='_', F/='_' -> {{M,F,'_'},Ms(MS),[global]};({M,F,MS,local}) when atom(M), atom(F), M/='_', F/='_' ->{{M,F,'_'},Ms(MS), [local]}; ({M,F,MS,global}) when atom(M), atom(F),M/='_', F/='_' -> {{M,F,'_'},Ms(MS),[global]}; (X) ->exit({rdbg,unrec_trace_pattern,X}) end, ChkTPs = fun(TPs) when list(TPs) -> lists:map(ChkTP, TPs); (TP) -> [ChkTP(TP)] end, SetTPs =fun({MFA,MS,Fs}) -> erlang:trace_pattern(MFA,MS,Fs) end, DoInitFun =fun(Time) -> erlang:register(rdbg, self()),erlang: start_timer(Time,self(), {die}),erlang:trace_pattern({'_','_','_'}, false,[local]),erlang:trace_pattern({'_','_','_'},false,[global]) end, InitFun =fun(Time,all,send) -> exit({rdbg, too_many_processes}); (Time,all,'receive') -> exit({rdbg,too_many_processes}); (Time,P,send) -> DoInitFun(Time), erlang:trace(Prc(P),true,[send,timestamp]); (Time, P,'receive') ->DoInitFun(Time), erlang:trace(Prc(P),true, ['receive', timestamp]);(Time,P,TPs) -> CTPs = ChkTPs(TPs), DoInitFun(Time),erlang:trace(Prc(P),true, [call, timestamp]), lists:foreach(SetTPs,CTPs) end, LoopFun = fun(G,N,Out) when N < 1 -> erlang:trace(all, false,[call,send,'receive']), erlang:trace_pattern({'_','_','_'},false, [local]), erlang:trace_pattern( {'_','_','_'},f alse,[global]), io:fwrite("**rdbg, ~w msgs **~n", [length(Out)]), io:fwrite("~p~n", [lists:reverse(Out)]), io:fwrite ("~p~n", process_info(self(), message_queue_len)]); (G,Cnt,Out) -> case process_info(self(),message_queue_len) of {_,N} when N > 100 -exit({rdbg,msg_queue, N}); _ -> ok end, receive {timeout,_,{die}} ->G(G,0,Out); {trace_ts,Pid,send,Msg,To,TS} ->G(G,Cnt-1,[{send,Ts(TS), Pi(To),Msg}|Out]); {trace_ts,Pid,'receive',Msg,TS} ->G(G,Cnt-1, [{'receive',Ts(TS),Msg}|Out]); {trace_ts,Pid,return_from, MFA,V,TS} ->G(G,Cnt-1,[{return,MFA,V}|Out]); {trace_ts,Pid,call,MFA,B,TS} whenbinary(B) -> G(G,Cnt-1,[{Pi (Pid),Ts(TS), {Stack(B),MFA}}|Out]);{trace_ts,Pid,call,MFA,TS} -> G(G,Cnt-1,[{Pi(Pid),Ts(TS),MFA}|Out])end end, Rdbg = fun(Time,Msgs,Proc, Trc) when integer (Time),integer(Msgs) -> Start = fun() ->InitFun(Time,Proc,Trc), LoopFun(LoopFun,Msgs,[]) end, erlang:spawn_link(Start) end.
```

redbug

```
redbug:start("erlang:now->stack").

09:03:49 <{erlang,apply,2}> {erlang,now,[]}
  {shell,eval_loop,3}
  {shell,eval_exprs,6}
  {shell,exprs,6}
```

redbug - safety

Safety comes from

- turns off if
  - reach timeout
  - reach number of trace message limit
  - trace messages are too large
  - trace messages coming in too fast
- disallows potentially dangerous traces

redbug trace patterns

redbug:start("ets:lookup").

redbug:start("ets:lookup(_,foo)").

redbug:start("ets:lookup(_,X)when X==foo").

09:46:22 <{erlang,apply,2}> {ets,lookup,[inet_db,foo]}

redbug - stack

redbug:start("ets:lookup(_,foo)-> stack").

09:48:03 <{erlang,apply,2}> {ets,lookup,[inet_db,foo]}
{shell,eval_loop,3}
{shell,eval_exprs,6}
{shell,exprs,6}

redbug - return

redbug:start("ets:lookup->return").

09:48:35 <dead> {ets,lookup,[foo,bla]}
09:48:35 <dead> {ets,lookup,2} -> {error,badarg}

redbug opts

time (15000) stop trace after this many ms
msgs (10) stop trace after this many msgs
proc (all) (list of) Erlang process(es)
targ (node()) node to trace on

print_file (standard_io) print to this file
file (none) use a trc file

conclusions

- reliability is easier in Erlang than in C++
- ...but not by any means automatic
- to get high reliability you need testing
- ...and debugging
- debuggability is a core strength of Erlang
- ...especially call tracing

eper.googlecode.com/