“I wish I'd known that when I started”
Or
“This is nothing like the brochure :-(("
Who the Hell are you?

- ShoreTel Sky
  - http://shoretelsky.com
- “Enterprise Grade” VoIP
- Elevator Pitch
Our Systems

- >9000 endpoints per server
- >150 calls per minute per server (peak)
- Real-time call control and reporting
- People are used to their computer crashing, but not their phone - very low downtime tolerance
Erlang?

- Simple, powerful syntax!
- Highly concurrent!
- Fault Tolerant!
- Hot code loading!
- We love it!
- I want to help you love it


```
-module(quicksort).
-export([qsort/1]).

qsort([]) -> [];
qsort([Pivot|Rest]) ->
    qsort([ X || X <- Rest, X < Pivot])
    ++ [Pivot]
    ++ qsort([ Y || Y <- Rest, Y >= Pivot]).
```
Highly Concurrent

- Tens of thousands of processes (threads) are no problem
- Each only costs you 1236 bytes of memory
- Primitives for send/receive:

```erlang
NewPid = spawn(?MODULE, f, []),
NewPid ! {message, Message}
...
f() ->
    receive
        {message, M} -> io:fwrite("~p", [M])
    end.
```
Fault Tolerant

- Crashes are localised
- Built in restart/recovery system
- Compare with C/C++ :)


Hot Code Loading

- Umm...I'll get to this later :)

Our Erlang Journey

- “Discovered” it at LCA 2007
- Hacked together a dynamic TFTP server
- Hacked together a soft-phone for automated testing
- Now used as the backbone of our call tracking and billing system
- We're rewriting entire core system using Erlang
Overview - I wish we'd known that...

- Dialyzer should be mandatory
- The VM *can* crash
- Message queues “just work”...except when they don't
- The OTP is invaluable
- Integration as a UNIX-style service is lacking
- Hot code loading is...interesting
- System monitoring is vital
Dialyzer

• Bringing (some) static type-safety to a dynamically typed language

• One run over your code will show you why you need it
How to crash the VM

- Out-Of-Memory
  - Non tail-recursive loops
  - Queue overflow

- Linked-in Drivers or NIFs
Non Tail-Recursive Loops

Good:

```python
main_loop() ->
do_something(),
wait_for_input(),
main_loop(). % Tail-call
```
Non Tail-Recursive Loops

Bad:

```python
main_loop() ->
    do_something(),
    wait_for_input(),
    main_loop(),
    ok.  %  Oops
```
Non Tail-Recursive Loops

Also Bad:

```
loop() ->
  A = do_something(),
  case A of
    done -> 1;
    continue -> 1 + loop() %% Also oops
  end.
```
Non Tail-Recursive Loops

- Bad(!):

```plaintext
foo(X) ->
  try
    case f(X) of
      continue -> foo(A);
      done -> ok
    end
  catch % try-catch must maintain the stack
    _ -> doom()
  end
```
Non Tail-Recursive Loops

• Good:

```
foo(X) ->
  try f(X) of
    % Exceptions thrown here are not caught:
    A -> foo(A); % So the stack is not kept
    _ -> ok
  catch
    _ -> doom()
end.
```
Queue Overflow

• Message queues are simple and powerful
• ...and can get you in very deep trouble
• How do you do it?
  • Outright overload
  • Selective receive
Simple overload

% This is called by lots of threads:
log_msg(Msg) ->
    logger ! {log, Msg}.

% But is all handled by one thread:
logger() ->
    receive
        {log, Msg} -> format_and_write(Msg);
        _ -> ok
    end,
    logger().
Selective Receive

receiver() ->
  % This is O(n):
  receive
    particular_message -> do_lots_of_work()
  end,

% This is O(1):
receive
  OtherStuff -> do_other_work(OtherStuff)
end,
receiver().
Selective Receive

• May not be obvious in your code:
  • \texttt{mnesia:transaction/1}

• Can take hours or even days to cause problems (monitor your system!)

• Somewhat mitigated as of R14 with new reference optimisation
New Reference Optimisation

R = make_ref(),
server ! {R, MyRequest},
receive
  {R, Resp} -> process_response(Resp)
end
New Reference Optimisation

% Compiler marks the queue here
R = make_ref(),
server ! {R, MyRequest},
% And only has to check from that mark
receive
   {R, Resp} -> process_response(Resp)
end
The Open Telephony Platform (OTP)

- Architectural framework for writing robust long running applications
- Forces you to consider process interaction, failure modes, crash behaviour etc
- Possibly overkill for “small” projects
- Definitely **mandatory** for anything else
- Learn it (come to my workshop tomorrow)!
The OTP - Solving problems you didn't know you had

- Making a “call” to another process. 
  First Try:

  ```erlang
  server_proc ! {request, ReqData},
  receive
    {response, RespData} -> RespData
  end.
  ```
The OTP

• But how can you be sure it's the right response?

Ref = make_ref(),
server_proc ! {request, Ref, ReqData},
receive
  {response, Ref, RespData} -> RespData
end,
The OTP

- But what if the server process doesn't exist?

```erlang
case whereis(server_proc) of
    undefined -> {error, noproc};
    Pid ->
        Ref = make_ref(),
        Pid ! {request, Ref, ReqData},
        receive
            {response, Ref, RespData} -> {ok, RespData}
        end
end
```
The OTP

• But what if the server process dies after the call?

```erlang
case whereis(server_proc) of
    undefined -> {error, noproc};
    Pid ->
        Ref = make_ref(),
        Pid ! {request, Ref, ReqData},
        receive
            {response, Ref, RespData} -> {ok, RespData}
            after 5000 -> {error, timeout}
        end
end
```
The OTP

- It'd be nice not to have to wait 5 seconds if the process crashed...

```erlang
MRef = erlang:monitor(process, server_proc),
Ref = make_ref(),
server_proc ! {request, Ref, ReqData},
receive
   {response, Ref, RespData} ->
      erlang:demonitor(MRef),
      {ok, RespData};
   {'DOWN', MRef, _, _} -> {error, no_proc};
   after 5000 ->
      erlang:demoniotr(MRef),
      {error, timeout}
end
```
The OTP

• But What if the remote node doesn't support `erlang:monitor`? (C/Java nodes don't).

• Enough! 12+ Lines of code for a simple “call” is already far too much.

```erlang
gen_server:call(server_proc, {request, ReqData})
```
More OTP Stuff

- Supervision Trees
- Event Handlers (subscribe-notify)
- FSMs
Erlang as a UNIX Service

• Erlang has an embedded heritage
• Turn on the device and walk away
• But this can cause trouble in the UNIX world...
Erlang as a UNIX Service

• Usual startup:
  • `erl -noshell -detached -boot myapp.boot`
  • Always returns 0 - success!
• But...what if some part of startup fails?
• Also, `-detached` means no console output
• No feedback => Unhappy sysadmins
.pid Files

• No .pid file - cannot easily find VM process on busy machines. Especially if it moves!
• Naive solution: Just write it from your Erlang code...
• But what if your code never runs?
• That's when you might need the .pid file most of all!
heart to Manage VM Crashes

- heart is a built in VM monitoring program
- A nice idea, but can make shutdown of broken VMs difficult
  - `kill -stop` is helpful
- Great for embedded systems
- Not so much for UNIX services
Log Rotation

- Log rotation is...unusual?
- No way to handle SIGHUP
- All these quirks together make packaging (.deb, .rpm etc) challenging.
Our Solution: erld

- Same basic principle as GNU screen
- Wraps erl and holds its terminal
- Programatically detaches from console
- Logs console output
- Intercepts `SIGHUP` for log rotation
- Returns useful error codes
- Manages crashes/restarts
- Open source (GPL)!
  https://github.com/ShoreTel-Inc/erld
Hot Code Loading

• Great idea!
• Ericsson use it to get insane (reported) uptimes on their AXD 301 switch
• But no other big projects use it on more than a single module basis. Why not?
Hot Code Loading

- It's really, really hard!
- There's no good tools to help
- The documentation is patchy (but improving)
- There's no easy way to integrate with common package management systems
- It's hard to test
System monitoring

- Erlang's VM has lots of great ways to monitor different parts of your system...

- But that's only useful if you use them

- And if you know what you're looking for
Some Key Monitoring Points

• Number of processes
  • \texttt{length(\text{erlang}:\text{processes}())}

• Queue length (esp. for busy processes)
  • \texttt{erlang:process_info(Pid, message_queue_len)}

• Total Memory Use
  • \texttt{erlang:memory/0,1}
Take-Home Messages

- Understand tail-calls
- Keep your message queues short
- Be careful of selective receives
- You will need to work to get your Erlang project to behave as a UNIX service
- Hot code loading is far harder than you think
- Monitor your system
- Use the OTP
- Use Dialyzer
Questions?

bduggan@shoretel.com
Thanks!

- The End.