Robust Erlang
(PFP Lecture 11)
John Hughes
Genesis of Erlang

• **Problem:** telephony systems in the late 1980s
  – Digital
  – More and more complex
  – Highly concurrent
  – Hard to get right

• **Approach:** a group at Ericsson research programmed POTS in different languages

• **Solution:** nicest was *functional programming*—but not concurrent

• **Erlang** designed in the early 1990s

"Plain Old Telephony System"
Mid 1990s: the AXD 301

- ATM switch (telephone backbone), released in 1998
- First *big* Erlang project
- Born out of the ashes of a disaster!
AXD301 Architecture

Subrack

1,5 million LOC of Erlang
16 data boards
2 million lines of C++

10 Gb/s
• 160 Gbits/sec (240,000 simultaneous calls!)
• 32 distributed Erlang nodes
• Parallelism vital from the word go
Typical Applications Today

Facebook Chat

Invoicing services for web shops—in 6 countries!

Distributed no-SQL database serving e.g. all Denmark’s medicine card data
What do they all have in common?

• Serving *huge* numbers of clients through parallelism

• Very high demands on *quality of service*: these systems should work *all* of the time
AXD 301 Quality of Service

• 7 nines reliability!
  – Up 99,999999% of the time

• Despite
  – Bugs
    • (10 bugs per 1000 lines is good)
  – Hardware failures
    • Always something failing in a big cluster
    • Avoid any SPOF
Example: Area of a Shape

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.

8> test:area({rectangle,3,4}).
12
9> test:area({circle,2}).
** exception error: no function clause matching
test:area({circle,2}) (test.erl, line 16)
10>

What do we do about it?
Defensive Programming

Anticipate a possible error

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y;
area(_) -> 0.

Return a plausible result.

11> test:area({rectangle,3,4}).
12
12> test:area({circle,2}).
0

No crash any more!
Plausible Scenario

• We write lots more code manipulating shapes
• We add circles as a possible shape
  – But we forget to change area!

<LOTS OF TIME PASSES>

• We notice something doesn’t work for circles
  – We silently substituted the wrong answer
• We write a special case elsewhere to ”work around” the bug
Handling Error Cases

• Handling errors often accounts for $>\frac{2}{3}$ of a system’s code
  – Expensive to construct and maintain
  – Likely to contain $>\frac{2}{3}$ of a system’s bugs

• Error handling code is often poorly tested
  – Code coverage is usually $<< 100\%$

• $\frac{2}{3}$ of system crashes are caused by bugs in the error handling code

But what can we do about it?
Don’t Handle Errors!

LET IT CRASH!

Stopping a malfunctioning program  
...is better than ...

Letting it continue and wreak untold damage
Let it crash... locally

• **Isolate** a failure within one process!
  – No shared memory between processes
  – No mutable data
  – One process cannot cause another to fail

• *One* client may experience a failure... but the rest of the system keeps going
A fatal exception 0E has occurred at 0020:C0011E36 in UXD VMM(01) + 00010E36. The current application will be terminated.

* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
We know what to do...

Detect failure

Restart
Using Supervisor Processes

- Supervisor process is *not* corrupted
  - One process *cannot* corrupt another
- Large grain error handling
  - simpler, smaller code
Supervision Trees

- Small, fast restarts
- Large, slow restarts
- Restart one or restart all
Detecting Failures: Links

Linked processes

EXIT signal
Linked Processes

“System” process

EXIT signal

This all works *regardless* of where the processes are running😊
Creating a Link

• link(Pid)
  – Create a link between self() and Pid
  – When one process exits, an exit signal is sent to the other
  – Carries an exit reason (normal for successful termination)

• unlink(Pid)
  – Remove a link between self() and Pid
Two ways to spawn a process

• spawn(F)
  – Start a new process, which calls F().

• spawn_link(F)
  – Spawn a new process and link to it atomically
Trapping Exits

• An exit signal causes the recipient to exit also
  – Unless the reason is normal

• ...unless the recipient is a system process
  – Creates a message in the mailbox:
    {’EXIT’, Pid, Reason}
  – Call process_flag(trap_exit, true) to become a system process
An On-Exit Handler

• Specify a function to be called when a process terminates

```erlang
on_exit(Pid, Fun) ->
    spawn(fun() ->
        process_flag(trap_exit, true),
        link(Pid),
        receive
            {'EXIT', Pid, Why} -> Fun(Why)
        end
    end).
```
Testing on_exit

5> Pid = spawn(fun() -> receive N -> 1/N end end).
<0.55.0>
6> test:on_exit(Pid, fun(Why) ->
    io:format("***exit: ~p\n", [Why]) end).
<0.57.0>
7> Pid ! 1.
***exit: normal
1
8> Pid2 = spawn(fun() -> receive N -> 1/N end end).
<0.60.0>
9> test:on_exit(Pid2, fun(Why) ->
    io:format("***exit: ~p\n", [Why]) end).
<0.62.0>
10> Pid2 ! 0.
=ERROR REPORT==== 25-Apr-2012::19:57:07 ===
Error in process <0.60.0> with exit value:
{badarith,[{erlang,'/',[1,0],[[]]}]}
***exit: {badarith,[{erlang,'/',[1,0],[[]]}]}
0
A Simple Supervisor

• Keep a server alive at all times
  – Restart it whenever it terminates

  keep_alive(Fun) ->
  Pid = spawn(Fun),
  on_exit(Pid,fun(_) -> keep_alive(Fun) end).

• Just one problem...

  How will anyone ever communicate with Pid?

Real supervisors won’t restart too often—pass the failure up the hierarchy
The Process Registry

• Associate names (atoms) with pids
• Enable other processes to find pids of servers, using
  – register(Name,Pid)
    • Enter a process in the registry
  – unregister(Name)
    • Remove a process from the registry
  – whereis(Name)
    • Look up a process in the registry
A Supervised Divider

defider() ->
    keep_alive(fun() -> register(divider, self()),
        receive
            N -> io:format("~n\p~n", [1/N])
        end
    end).

4> divider ! 0.
=ERROR REPORT==== 25-Apr-2012::20:05:20 ===
Error in process <0.43.0> with exit value:
{badarith, [{test, '-divider/0-fun-0-', 0,
            [{file, "test.erl"}, {line, 34}]}]}
0
5> divider ! 3.
0.3333333333333333
3
Supervisors supervise servers

- At the leaves of a supervision tree are processes that service requests
- Let’s decide on a protocol

```
{ClientPid,Ref},Request

rpc(ServerName, Request)

{Ref,Response}

reply({ClientPid, Ref}, Response)
```
rpc/reply

rpc(ServerName,Request) ->
  Ref = make_ref(),
  ServerName ! {{self(),Ref},Request},
  receive
    {Ref,Response} ->
    Response
  end.

reply({ClientPid,Ref},Response) ->
  ClientPid ! {Ref,Response}.
Example Server

account(Name,Balance) ->
  receive
    {Client,Msg} ->
      case Msg of
        {deposit,N} ->
          reply(Client,ok),
          account(Name,Balance+N);
        {withdraw,N} when N=<Balance ->
          reply(Client,ok),
          account(Name,Balance-N);
        {withdraw,N} when N>Balance ->
          reply(Client,[error,insufficient_funds]),
          account(Name,Balance)
      end
  end. 
A Generic Server

• Decompose a server into...
  – A *generic* part that handles client—server communication
  – A *specific* part that defines functionality for this particular server

• Generic part: receives requests, sends replies, recurses with new state

• Specific part: *computes* the replies and new state
A Factored Server

server(State) ->
  receive {Client,Msg} -> {Reply,NewState} = handle(Msg,State),
  reply(Client,Reply),
  server(NewState)
end.

displaybox(handle(Msg,Balance) ->
  case Msg of
    {deposit,N} -> {ok, Balance+N};;
    {withdraw,N} when N=<Balance -> {ok, Balance-N};;
    {withdraw,N} when N>Balance  ->
      {{error,insufficient_funds}, Balance}
  end.
)
Callback Modules

• Remember:

- foo:baz(A,B,C)
  - Call function baz in module foo

- Mod:baz(A,B,C)
  - Call function baz in module Mod (a variable!)

• Passing a module *name* is sufficient to give access to a collection of "callback" functions
server(Mod,State) ->
    receive {Client,Msg} ->
        {Reply,NewState} = Mod:handle(Msg,State),
        reply(Client,Reply),
        server(Mod,NewState)
    end.

new_server(Name,Mod) ->
    keep_alive(fun() -> register(Name,self()),
               server(Mod,Mod:init()) end).
The Bank Account Module

```
handle(Msg,Balance) ->
case Msg of
  {deposit,N} -> {ok, Balance+N};
  {withdraw,N} when N=<Balance -> {ok, Balance-N};
  {withdraw,N} when N>Balance  ->
    {{error,insufficient_funds}, Balance}
end.
init() -> 0.
```

- This is *purely sequential* (and hence easy) code
- This is all the application programmer needs to write
What Happens If...

• The client makes a bad call, and...
• The handle callback crashes?
• The server crashes
• The client waits for ever for a reply

• Let’s make the client crash instead
Erlang Exception Handling

catch <expr>

• Evaluates to V, if <expr> evaluates to V

• Evaluates to {’EXIT’,Reason} if expr throws an exception with reason Reason
server(Mod,State) ->
  receive
    {Pid,Msg} ->
      case catch Mod:handle(Msg,State) of
        {'EXIT',Reason} ->
          reply(Name,Pid, {crash,Reason}),
          server(Mod, State);
        {Reply,NewState} ->
          reply(Name,Pid, {ok,Reply}),
          server(Mod,NewState)
      end.
  end.
end.

rpc(Name,Msg) ->
  ...
  receive
    {Ref, {crash,Reason}} ->
      exit(Reason);
    {Ref, {ok,Reply}} ->
      Reply
  end.

What should we put here?

We don’t have a new state!
Transaction Semantics

• The Mk II server supports transaction semantics
  – When a request crashes, the client crashes...
  – ...but the server state is restored to the state before the request

• Other clients are unaffected by the crashes
Hot Code Swapping

• Suppose we want to *change the code* that the server is running
  – It’s sufficient to change the *module* that the callbacks are taken from

```erl
server(Mod,State) ->
  receive
    {Client, {code_change,NewMod}} ->
      reply(Client, {ok,ok}),
      server(NewMod,State);
    {Client,Msg} -> ...
  end.
```

The State is not lost
Two Difficult Things Before Breakfast

• Implementing transactional semantics in a server
• Implementing dynamic code upgrade without losing the state

Why was it easy?
• Because all of the state is captured in a single value...
• ...and the state is updated by a pure function
gen_server for real

- 6 call-backs
  - init
  - handle_call
  - handle_cast—messages with no reply
  - handle_info—timeouts/unexpected messages
  - terminate
  - code_change

- Tracing and logging, supervision, system messages...

- 70% of the code in real Erlang systems
OTP

• A handful of generic behaviours
  – gen_server
  – gen_fsm—traverses a finite graph of states
  – gen_event—event handlers
  – supervisor—tracks supervision tree+restart strategies

• And there are other more specialised behaviours...
  – gen_leader—leader election
  – ...

Erlang’s Secret

• Highly robust
• Highly scalable
• Ideal for internet servers

• 1998: Open Source Erlang (banned in Ericsson)
• First Erlang start-up: Bluetail
  – Bought by Alteon Websystems
    • Bought by Nortel Networks

$140 million in
<18 months
SSL Accelerator

• "Alteon WebSystems' SSL Accelerator offers phenomenal performance, management and scalability."

  – Network Computing
2004 Start-up: Kreditor

- New features every few weeks—never down
- ”Company of the year” in 2007
- Growth: >13,000% (to over 700 people!)
- Market leader in Scandinavia
Erlang Today

• Scaling well on multicores
  – 64 cores, no problem!

• Many companies, large and small
  – Amazon/Facebook/Nokia/Motorola/HP...
  – Ericsson recruiting Erlangers
  – No-sql databases (Basho, CouchDB, Hibari...)
  – Many many start-ups

• "Erlang style concurrency" widely copied
  – Akka in Scala (powers Twitter), Cloud Haskell...
Requests per month to www.erlang.org

First Intel dual core released
Erlang Events

• Erlang User Conference, Stockholm
• Erlang Factory (multiple tracks)
  – London
  – San Francisco
• Erlang Factory Lite
  – Brisbane, Paris, Munich, Edinburgh, Amsterdam
  – Brussels, Krakow, Zurich, St.Andrews...
• ErlangCamp
  – Chicago, Spain...
Coming up on Thursday...

MAP/REDUCE