Erlang

Joe Armstrong
Hi Folks --

Erlang is worth looking at.
Though OOP came from many motivations, two were central. The large scale one was to find a better module scheme for complex systems involving hiding of details, and the small scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether.

...doing encapsulation right is a commitment not just to abstraction of state, but to eliminate state oriented metaphors from programming.

The Early History of Smalltalk
Alan Kay
for \( i \) in \{\text{objects, processes}\}
{
    create very large numbers of \( i \)
    \( i \) work the same way on all OS's
    \( i \)'s are garbage collected
    \( i \) are location transparent
    \( i \) cannot damage other \( i \)
    \( i \) are defined in the language
    creating and destroying \( i \) is light-weight
Erlang is Smalltalk as Alan Kay wanted it

- Niall Dalton
How do we build systems that run forever, are scalable, fault-tolerant, evolve with time and work reasonably well works despite errors in the software?
Difficult
To make a fault-tolerant system you need at least two computers
this is

Distributed Programming
Simplify the problem

no sharing
pure message passing
no locks
This is

Concurrency Oriented Programming

Programming
Concurrency Oriented Programming

- A style of programming where concurrency is used to structure the application
- Large numbers of processes
- Complete isolation of processes
- No sharing of data
- Location transparency
- Pure message passing

My first message is that concurrency is best regarded as a program structuring principle”

Structured concurrent programming
- Tony Hoare
Redmond, July 2001
COP Design Rules

1) Identify the concurrent operations in your problem
2) Identify the message channels
3) Write down the set of message seen on each channel
4) Write down the protocols
5) Write the code

Try to make the design isomorphic to the problem - ie a 1:1 correspondence between the process/message structure in the model and the problem.
Who am I?

Inventor of Erlang, UBF
Chief designer of OTP
Founder of the company Bluetail

Currently
Senior System Architect
Ericsson AB

Current Interests
Concurrency Oriented Programming
Multi-core CPUs
FPGAs
Cats
Motorbikes
How do we correct hardware failures?
Replicate the hardware

How do we correct software errors?
Having two identical copies of the software won't work - both will fail at the same time and for the same reason

Why does your computer crash?
Which fails more often, hardware or software?
History

1986 - Pots Erlang (in Prolog)
1987 - ACS/Dunder
1988 - Erlang -> Strand (fails)
1989 - JAM (Joe's abstract machine)
1990 - Erlang syntax changes (70x faster)
1991 - Distribution
1992 - Mobility Server
1993 - Erlang Systems AB
1995 - AXE-N collapses. AXD starts
1996 - OTP starts
       Bluetail formed
1999 - BMR sold
2000 - Alteon buys Bluetail. Nortel buys Alteon
2002 - UBF. Concurrency Oriented Programming
2003 - Ph.D. Thesis - Making reliable systems
2006 - Multi-core Erlang
How do we make systems?

Systems are made of black boxes (components)

Black boxes execute concurrently

Black boxes communicate

How the black box works internally is irrelevant

Failures inside one black box should not crash another black box
Problem domain

- Highly concurrent (hundreds of thousands of parallel activities)
- Real time
- Distributed
- High Availability (down times of minutes/year – never down)
- Complex software (million of lines of code)
- Continuous operation (years)
- Continuous evolution
- In service upgrade
We start with the `bank_client.erl`

```
-module(bank_client).
-export([deposit/2, withdraw/2, balance/1]).

deposit(Who, X) -> simple_reply([deposit(Who, X)]);
withdraw(Who, X) -> simple_reply([withdraw(Who, X)]);
balance(Who) -> simple_reply([balance(Who)]).

simple_reply(Reply) ->
    case reply_conversion(Reply, 0) of
        ok, Socket =>{ ok, Socket } ->
            send_reply(Socket, {ok, to_binary(Reply)});
        E =>{_, _} ->
            wait_reply(Socket) ->
                receive reply_0(Socket) ->
                    form = binary_to_term(bin);
                    Bank = close(Socket);
                    receive_reply(Bank, Socket) ->
                        end, false
                end, true
    end.
```

This is a simple “no frills” client, that accesses a bank server.

The address of the bank server is "hard wired" into the program at address, hostname and port set.

Since we are not using distributed Erlang we have to do all encoding and decoding of Erlang terms ourselves. This is achieved by using...
Philosophy

Concurrency Oriented Programming

1. COPLs support processes
2. Processes are Isolated
3. Each process has a unique unforgeable Id
4. There is no shared state between processes
5. Message passing is unreliable
6. It should be possible to detect failure in another processes and we should know the reason for failure
System requirements

R1. Concurrency: processes
R2. Error encapsulation: isolation
R3. Fault detection: what failed
R4. Fault identification: why it failed
R5. Live code upgrade: evolving systems
R6. Stable storage: crash recovery
Isolation

Hardware components operate concurrently are isolated and communicate by message passing.
Consequences of Isolation

Processes have *share nothing* semantics and data must be copied

*Message passing is the only way to exchange data*

*Message passing is asynchronous*
GOOD STUFF

Processes

Copying

Message passing
Language

My program should not be able to crash your program.
Need strong isolation and concurrency.

Processes are OK - threads are not (threads have shared resources).

Can't use OS processes (Heavy - semantics depends on OS).
Isolation

My program should not be able to crash your program.

This is the single most important property that a system component must have

All things are not equally important
Erlang

Lightweight processes (lighter than OS threads)
Good isolation (not perfect yet ...)
Programs never lose control
Error detection primitives
Reason for failure is known
Exceptions
Garbage collected memory
Lots of processes

Functional

Agner Krarup Erlang (1878-1929)
Erlang in 11 minutes
Erlang

You can create a parallel process

\[
\text{Pid} = \text{spawn(fun() -> ... end}).
\]

then send it a message

\[
\text{Pid} ! \text{Msg}
\]

and then wait for a reply

\[
\text{receive}
\]

\[
\{\text{Pid}, \text{Rely}\} ->
\]

\[
\text{Actions}
\]

\[
\text{end}
\]

It typically takes 1 microsecond to create a process or send a message.
Generalisation

Client
Pid = spawn(fun() -> loop() end)
Pid ! {self(), 21},
receive
    {Pid, Val} -> ...
end

Server
loop() ->
    receive
        {From, X} ->
            From ! {self(), 2*X},
            loop()
    end.

A simple process

Client
Double = fun(X) -> 2 *X end,
Pid = spawn(fun() -> loop(Double) end)
Pid ! {self(), 21},
receive
    {Pid, Val} -> ...
end

Server
loop(F) ->
    receive
        {From, X} ->
            From ! {self(), F(X)},
            loop(F)
    end.

Generalised
A generic server

-module(gserver).
-export([start/1, rpc/2, code_change/2]).

start(Fun) ->
    spawn(fun() -> loop(Fun) end).

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} ->
            Reply
    end.

code_change(Pid, Fun1) ->
    Pid ! {swap_code, Fun1}.

loop(F) ->
    receive
        {swap_code, F1} ->
            loop(F1);
        {Pid, X} ->
            Pid ! {self(), F(X),
                    loop(F);
    end.

Double = fun(X) -> 2*X end,
Pid = gserver:start(Double),
...
Triple = fun(X) -> 3*X end,
gserver:code_change(Pid, Triple)
A generic server with data

-module(gserver).
-export([start/2, rpc/2, code_change/2]).

start(Fun, Data) ->
    spawn(fun() -> loop(Fun, Data) end).

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, X} ->
            {Reply, Data1} = Fun(X),
            Pid ! {self(), Reply},
            loop(Fun, Data1);
        end.

code_change(Pid, Fun1) ->
    Pid ! {swap_code, Fun1}.
Trapping errors

In Pid1 ...
Pid2 = spawn_link(fun() -> ... end).
process_flag(trap_exit, true)
...

receive
  {'EXIT', Pid, Why} ->
    Actions
end.

error detection + reason for failure (slide 10)
Why remote trapping of errors?

To do fault-tolerant computing you need at least TWO computers

Which means you can't share data
Programming for errors

If you can't do what you want to do try and do something simpler

The supervisor monitors the workers and restarts them if they fail
A supervision hierarchy
OTP behaviours

Generic libraries for building components of a real-time system.

Includes

Client-server
Finite State machine
Supervisor
Event Handler
Applications
Systems
case studies

Ericsson AXD301 (in Engine)
Size = 1136150 lines Erlang
Dirty functions = 0.359%
Availability = 99.9999999%

Alteon (Nortel) SSL accelerator
Size = 74440 line Erlang
Dirty functions = 0.82%

Ref: Armstrong Ph.D. thesis
Commercial Successes

Ericsson AXD301 (part of “Engine”)
Ericsson GPRS system
Alteon (Nortel) SSL accelerator
Alteon (Nortel) SSL VPN
Teba Bank (credit card system – South Africa)
T-mobile SMS system (UK)
Kreditor (Sweden)
jabber.org
How do we make systems?

Systems are made of black boxes (components)

Black boxes execute concurrently

Black boxes communicate with defined (universal) protocols

The protocol is checked externally

How the black box works internally is irrelevant
APIs done wrong

+type file:open(fileName(), read | write) ->
  {ok, fileHandle()}
  | {error, string()}.

+type file:read_line(fileHandle()) ->
  {ok, string()} | eof.

+type file:close(fileHandle()) ->
  true.

+deftype fileName()   = [int()]
+deftype string()        = [int()].
+deftype fileHandle() = pid().

silly() ->
  {ok, H} = file:open("foo.dat", read),
  file:close(H),
  file:read_line(H).
APIs with state

+type start x file:open(fileName(), read | write) ->
  {ok, fileHandle()} x ready
  | {error, string()} x stop.

+type ready x file:read_line(fileHandle()) ->
  {ok, string()} x ready
  | eof x atEof.

+type atEof | ready x file:close(fileHandle()) ->
  true x stop.

+type atEof | ready x file:rewind(fileHandle()) ->
  true x ready.

silly() ->
{ok, H} = file:open("foo.dat", read),
  file:close(H),
  file:read_line(H).
Protocols or APIs

How things work inside the black box is irrelevant

Check the protocol at the boundaries to the black box
Finally

My program should not be able to crash your program.

This is the single most important property that a system component must have

All things are not equally important