Concurrent and Distributed Erlang

Lectures F6 + F7 (Chapters 11..14)

Joe Armstrong

Ericsson AB

W6/F6 - Concurrent Erlang

- Chapters 11,12.
- Concurrency Intro.
- Concurrency Primitives.
- Client Server.
- Processes are cheap.
- Send/Receive.
- Spawn.
- Registered Processes .

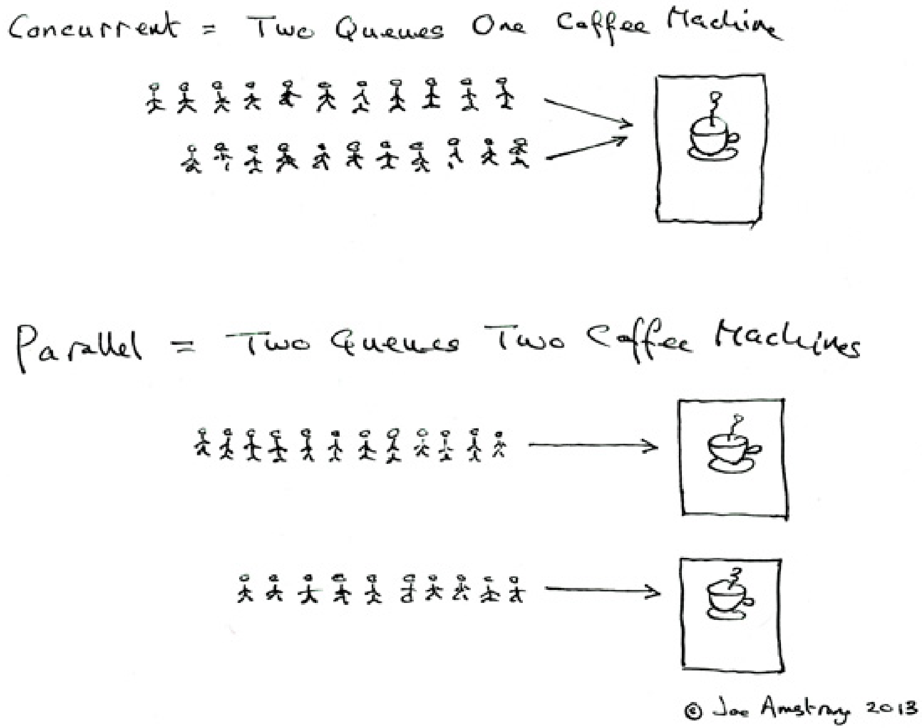
Concurrency Intro

- The World is Parallel.

- Concurrency should be in the programming langauge not the operating system.

- Concurrent is not parallel.

Explain it to a five year old



Concurrent vs Parallelism

- Concurrent = *the illusion of parallelism*.
- Parallel = *really parallel*.
- On a quad core there can only really be 4 things happening at the same time.

  Note that in reality there is more parallelism than is implied by the number of cores:
- Instruction level parallelism (several assembly instructions can execute at the same time).
- Memory fetch/store parallelism (several memory locations can be transfered to registers at the same time).
- Pipeline parallelism (several things can happen in the pipeline at the same time).
- I/O parallelism (SSD's can have several controllers).

  All the above are very difficult to program.

Concurrency Primitives

- spawn, send, receive

```
Pid = spawn(fun() -> ... end)
Pid = spawn(Mod, Func, [Arg1, Arg2, ...])

Pid ! Value

receive
  Pattern1 -> Actions1;
  Pattern2 -> Actions2;
  ...
 end
```

- `receive` is similar to `case`

```
case Value of
  Pattern1 -> Actions1;
  Pattern2 -> Actions2;
  ...
 end
```


Exercise

- Create a registered server called double.
- If you send it an integer it doubles it and sends back the reply.
- It crashes if you send it an atom.
- Make a process that sleeps for a random time and sends a message to the double server and causes it to crash.
- Make a monitor process that detects that the server has crashed. It restarts the server after a random delay.
- Make a client function that sends a request to the server and times out if the request is not satisfied. We can assume the server has crashed. The client should wait a second and then try again.
- Abort the client if it has tried more than ten times.

Client - Server 1

```
start1() ->
   spawn(f6, loop1, []).

loop1() ->
  receive
    {square, X} ->
      print(X*X),
      loop1()
  end.

Pid ! {square, 10}
```

- Run with `f6:start1().`
- Pid always means **Process Identifier**.
- How do we get the result back?

Client - Server 2

- Getting the result back?

```
start2() ->
   spawn(f6, loop2, [])

loop2() ->
   receive
     {From, {square, X}} ->
        From ! X*X,
        loop2()
   end.

Pid ! {self(), {square, 10}},
receive
   Result ->
     ...
end
```

- How do we know the result we got back was from the server and not from some other process that just happend to send us a message?

Client - Server 3

- Pattern match the reply message to check that the reply comes from the correct process.

```
start3() ->
  spawn(f6, loop3, []).

loop3() ->
  receive
    {From, {square, X}} ->
      From ! {self(), X*X},
      loop3()
  end.

area_square(Pid, X) ->
   Pid ! {self(), {square, X}},
   receive
    {Pid, Area} -> Area
end.
```

- Say something about selective receive ...

Selective Receive

```
receive
   Pattern1 ->
      Actions1;
   Pattern2 ->
      Actions2
end
```

This suspends until a message matching `Pattern1` or `Pattern2` is received. All other messages are queued.

Client - Server 4 (abstract the RPC)

```
%% old

area_square(Pid, X) ->
   Pid ! {self(), {square, X}},
   receive
     {Pid, Area} -> Area
end.

%% refactored

area_square(Pid, X) ->
   rpc(Pid, {square, X}).

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


Client - Server 5 (Tagged replies)

```
Pid = spawn(fun() -> loop() end)

loop() ->
  receive
    {From, Tag, {square, X}} ->
      Result = X*X,
      From ! {Tag, Result},
      loop()
  end.

rpc(Pid, Query) ->
   Tag = erlang:make_ref(),
   Pid ! {self(), Tag, Query},
   receive
     {Tag, Result} ->
       Result
   end.
```


Timeouts

```
receive
   Pattern1 ->
      Actions1;
   Pattern2 ->
      Actions2;
   ...
after Time ->
   Actions
end.
```

Client - Server 6

- We detect that the server has not replied with a timeout...

```
rpc(Pid, Query) ->
   Tag = erlang:make_ref(),
   Pid ! {self(), Tag, Query},
   receive
      {Tag, Result} ->
         {ok, Result}
      after 1000 ->
         {error, timeout}
   end.
```

Client - Server 7

- Umm ....

```
rpc(Pid, Query) ->
   Tag = erlang:make_ref(),
   Pid ! {self(), Tag, Query},
   receive
      {Tag, Result} ->
         Result
      after TIME ->
         DO SOMETHING
   end.
```

- What is DO SOMETHING?
- What is TIME?
- Idempotence.
- Getting DO SOMETHING and TIME right is incredibly difficult.

Why is this difficult?

- We send a message to a server.
- We do not get a reply

  So:

- Either the server has crashed, or,
- The communication channel is broken.

  Recovering from this is very difficult (in many cases it is impossible).

Exercise (reminder)

DO SOMETHING means:

- Try again N times with a random delay and then give up.
- Write some code to randomly crash the server.
- Write some code to restart the server if it crashes.

What really happens - the mailbox

- Each process has a mailbox.
- Send causes a message to be added to the mailbox.
- When a process message is added to a mailbox the process is scheduled for execution.
- When the process next executes it checks if the new mails match any of the receive patterns.
- If the message does not match the process suspends.

The scheduler

- Processes run for 1000 reductions and are then suspended. They stay in the run queue.
- Processes waiting for a message are removed from the run-queue.
- When a message is added to the mailbox we add it to the run queue (if it is not in the run queue).
- There is one sheduler per core *not really true – can be two or more*.
- Processes can be moved between schedulers.

The scheduler

Client Server patterns

```
Pid = spawn(fun() -> loop(State) end)

loop(State) ->
  receive
    {From, Pattern1} ->
      State1 = ...
      Result = ...
      From ! {self(), Result},
      loop(State1);
    {From, Pattern2} ->
      ...
  end.

func1(Pid, Args) -> rpc(Pid, Args).

rpc(Pid, Args) ->
   Pid ! {self(), Args},
   receive
      {Pid, Ret} -> Ret
   end.
```


A Stateful counter

```
Pid = spawn(fun() -> counter(0) end)

counter(N) ->
  receive
    {From, {add,K}} ->
      From ! {self(), ok},
      counter(N+K)
  end.

add(K) -> rpc(Pid, {add,K}).

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


Extend the server

```
  receive
    ...

    {From, reset} -
      counter(0)
    ...
    {From, read} ->
      From ! {self(), N},
      counter(N);
    ...
  end.

reset(K) -> rpc(Pid, reset).
...
```

- Add extra patterns in the server.
- Add API routines.

Send functions in the messages

```
Pid = spawn(fun() -> loop(State) end)

loop(State) ->
  receive
    {From, F} ->
      {Reply, NewState}= F(State),
      From ! {self(), Reply},
      loop(NewState)
  end.

add(K) -> rpc(Pid,
            fun(State) ->
              {ack, K+State}
            end).
```

Send the server in a message

```
start() ->
   spawn(fun() -> wait() end)

wait() ->
  receive
    {become, F} ->
        F()
  end.

Pid = start(),
...
Pid ! {become, fun() -> loop/1}.

loop(State) ->
   receive
     ...
   end
```


processes are cheap

```
-module(f6).
-compile(export_all).

time(N) ->
   {Time, _} = timer:tc(f6, time_test, [N]),
   Tsec = Time / 1000000,
   {spawned, trunc(N / Tsec), 'processes/sec'}.

time_test(0) ->
   true;
time_test(N) ->
   spawn(fun() -> true end),
   time_test(N-1).
```

- show this.

erl -smp disable

```
erl -smp disable
Eshell V5.10.1 (abort with ^G)
1> f6:time(100000).
{spawned,1027305,'processes/sec'}
2> f6:time(1000000).
{spawned,1212416,'processes/sec'}
```

- 1.2 Million processes/sec.

Registered Processes

- `Pid ! Message` sends a message to the mailbox of the process `Pid`.

- How do we know Pid?

- Only the parent knows Pid

```
start() ->
   Pid = spawn(...),
   Pid ! Message,
   ...
```


Registered Processes

```
start() ->
   Pid = spawn(...),
   register(counter, Pid),
   ...
```

- Now any process can send a message to the process

```
> counter ! {add, 12}
```


Tail recursion

```
start() -> spawn(Mod, loop, [Arg1, ...]).

loop(Arg1, ...) ->
   receive
     Pattern1 ->
        ...
        loop(1);
     Pattern2 ->
        ...
   end
```

- The last thing you do is call yourself.

Non Tail recursion

```
start() -> spawn(Mod, loop, [Arg1, ...]).

loop(Arg1, ...) ->
   receive
     Pattern1 ->
        ...
        loop(1, ..), %% NO NO NO NO
        ... <-- Don't call stuff after ... ; the call to loop
     Pattern2 ->
        ...
   end
```


Tail recursion (again)

- Co-routines.
- Continuation passing style.

```
state1(...) ->
   receive
     Pattern1 ->
        ...
        state2(1);
     Pattern2 ->
        ...
   end.

state2(...) ->
   receive
     Pattern1 ->
       ...
       state3(...);
     ...
   end
```

- If something never returns, it must be the last thing you call.

What does Mod:Func really mean?

- What's the difference between `loop` and `loop1`?

```
-module(foo).

loop(State) ->
   receive
     Pattern1 ->
        ...
        loop(NewState)
   end.

loop1(State) ->
   receive
     Pattern1 ->
        ...
        foo:loop1(NewState)
   end.
```

- Mod:Func calls the latest version.

Spawn MFA - or fun

```
start1() ->
   spawn(Mod, Func, [Arg1, Arg2, ..., ArgN])

start2() ->
   spawn(fun() -> ... end)
```


W6/F7 - Error and Distributed Programming

- Chapters 13,14.
- Links.
- Monitors.
- The error model.
- Firewall.
- Generic Allocator.
- Why Distributed Programming?
- Erlang distribution.
- Explicit sockets and protocols.
- Distribution BIFS.
- Nano-twitter.
- Security.

Links

- When an unlinked process dies nobody will know about it.
- When a linked process dies the processes in the link-set of the process will be sent an error signal.
- A normal process receiving a non-normal error signal will die.
- A system process will convert the error signal to an error message and can receive it like any other message.
- A processes that terminates normally sends a normal error siognal to its link set.

signal: {'EXIT', Pid, Why}

Why kan vara: normal

kan också vara: kill

Error handling BIFs

- `link(Pid) unlink(Pid)`
- `process_flag(trap_exit, true)`
- `spawn_link(Fun) spawn_link(M,F,A)`
- `exit(Pid, Why) exit(Pid, kill)`

Trapping exits

```
start() ->
   spawn_link(M, F, A) end.

start() ->
   process_flag(trap_exit, true),
   loop().

loop() ->
   receive
      {'EXIT', Pid, Why} ->
          ...
   end.
```


A process that restarts a failed process

```
start_and_watch() ->
   Pid = spawn(f6, loop1, []),
   spawn(f6, watch, [Pid]),
   Pid.

watch(Pid) ->
   link(Pid),
   process_flag(trap_exit, true),
   print({watching, Pid}),
   receive
      {'EXIT', Pid, Why} ->
         print({ohDear,Pid,crashed,because,Why})
   end.
```


Running the watcher

```
Pid2 = f6:start_and_watch().
{watching,<0.62.0>}
<0.62.0>
9> Pid2 ! {square,abc}.
{ohDear,<0.62.0>,crashed,because,
      {badarith,[{f6,loop1,0,[{file,"f6.erl"},{line,21}]}]}}

=ERROR REPORT==== 31-Jan-2014::10:26:29 ===
Error in process <0.62.0> with exit value:
 {badarith,[{f6,loop1,0,[{file,"f6.erl"},{line,21}]}]}

{square,abc}
```


Spawn and Link race conditions

```
start_and_watch() ->
   Pid = spawn(f6, loop1, []),
   spawn(f6, watch, [Pid]),
   Pid.

watch(Pid) ->
   link(Pid),
   process_flag(trap_exit, true),
   ...
```

- If you are very quick, loop1 might crash *before* the watcher executes the `link` statement.
- That's why we have `spawn_link`.

Monitors

- One directional links.
- A monitor is like two links “back to back”.

```
monitors: ------->

links:        <------->
```

The Worker/Manager model

- Workers do work. They crash if they cannot do what they are supposed to do.
- Managers detect the failure of workers and restart them.
- `start_and_watch` was a very simple Worker/Manager program.
- OTP has supervisors which generalises this idea.
- Akka (Java) is a clone of this idea.

Generic Allocator

- Server links to clients and deallocates respources if the clients crash

```
loop(State) ->
  receive
    {From, Tag, {allocate, X}} ->
       {Resource, State1} = allocate(From, X, State),
       link(From),
       From ! {Tag, Resource},
       loop(State1);
    ...

    {'EXIT', Pid, _} ->
       State1 = dealloc_resources_owned_by(Pid, State),
       loop(State1)
  end.
```


Distributed Programming?

- Fun.
- World is distributed.

Why? ITS FUN =]

Erlang distribution

- Needs only one new idea.

  extra
- `spawn(Node, Mod, Func, Args).`
- Remote Pids work just like local Pids.
- Can test on one machine, deploy on many.

Nano twitter

```
start() ->
   register(twit, spawn(nano_twitter, watcher, [])).

watcher() ->
   receive
      Any ->
         print({tweet, Any}),
         watcher()
      after 5000 ->
            print(yawn),
            watcher()
      end.

connect() ->
   pong = net_adm:ping('twit@joe').

tweet(Msg) ->
   rpc:cast('twit@joe', erlang, send, [twit, Msg]).
```


Running Nano Twitter

In one terminal:

```
erl -noshell -sname twit -s nano_twitter start
yawn
...
```

Someplace else:

```
$ erl -sname one
(one@joe)1> nano_twitter:connect().
pong
(one@joe)2> nano_twitter:tweet('hi joe').
true
```

Back where you started:

```
yawn
{tweet,'hi joe'}
```


Fun Exercise (1)

- No credits.
- Extend `nano_twitter`
- Work in pairs.
- Client on one machine.
- Same LAN.
- Server on a different machine

Fun Exercise (2)

- No credits.
- Server on a different machine with a different OS.
- On a WAN.

Fun Exercise (3)

- No credits.
- Get the entire class running.

Fun Exercise (4)

- Credits.
- $$$$
- Quit school.
- Form a company.
- Connect the world togther.

RTFM

- `http://www.erlang.org/doc/reference_manual/distribute`
- Designed for clusters in the same LAN not wide scale distribution.
- Cookie security.
- Great for tightly connected clusters in the samer administrative domain.
- Code distribution problem not solved (assumes same backend to fetch the code from) - originally all nodes read from a NFS file system.

Next Week

- Doing it with sockets :-)

Have fun