

Distributed Erlang – Workshop

Hans Svensson <hanssv@ituniv.se>



- Erlang programs are made of (lots of) **processes**
 - Processes can send **messages** to each other
 - Messages may or may not be **received**, to be sure, one needs a confirmation message
 - Pairs of processes can be **linked** together, if one of the processes dies, the other process is notified of the death, and the reason
-

- Erlang programs are made of (lots of) processes
- Processes can send messages to each other
- Messages may or may not be received, but you are sure one needs a confirmation
- Pairs of processes can be created, and if one of the processes dies, the other is notified of the reason

Concurrency-oriented
programming

- Joe Armstrong



Pid = spawn (Fun)

Creates a fresh process that is going to evaluate Fun. The spawned process is running in parallel with the current process. The returned process identifier (Pid) is unique, and can be used to send messages to the process.

Spawn can also be called with three arguments:
spawn (Module, Function, Arguments)



Pid ! Message

Sends a message (**Message**) to the process identified by **Pid**. Sending a message is asynchronous, i.e. the sender does not wait for the message to be delivered.

The send expression returns **Message**.

The send *operation* never fails.

receive ... end

Receiving a message is ***blocking***, i.e. the receiving process will wait until a ***matching*** message arrives.

```
receive
  Pattern1 [when Guard1] -> Exprs1;
  Pattern2 [when Guard2] -> Exprs2;
  ...
  PatternN [when GuardN] -> ExprsN
end
```



- Patterns and guards are similar to patterns and guards in function clauses
 - Messages are matched against the patterns from top to bottom. (The first message is tried against every pattern before considering a second message).
-



- We will look at some code, which handles geometrical shapes (again)
 - `{square, X}`
 - `{circle, R}`
 - `{rectangle, W, H}`
 - We want to compute area
 - **But this time in a separate process!**
-

- Recall the area/1 function

```
%% Area of a geometrical figure
area({square, X}) ->
    X * X;
area({circle, R}) ->
    R * R * ?PI;
area({rectangle, H, W}) ->
    H * W.
```

```
-export([loop/0]).
```

```
loop() ->
```

```
    receive
```

```
        Shape ->
```

```
            io:format("Area of ~p is ~p~n",  
                      [Shape, area(Shape)]),
```

```
            loop()
```

```
end.
```

Area example

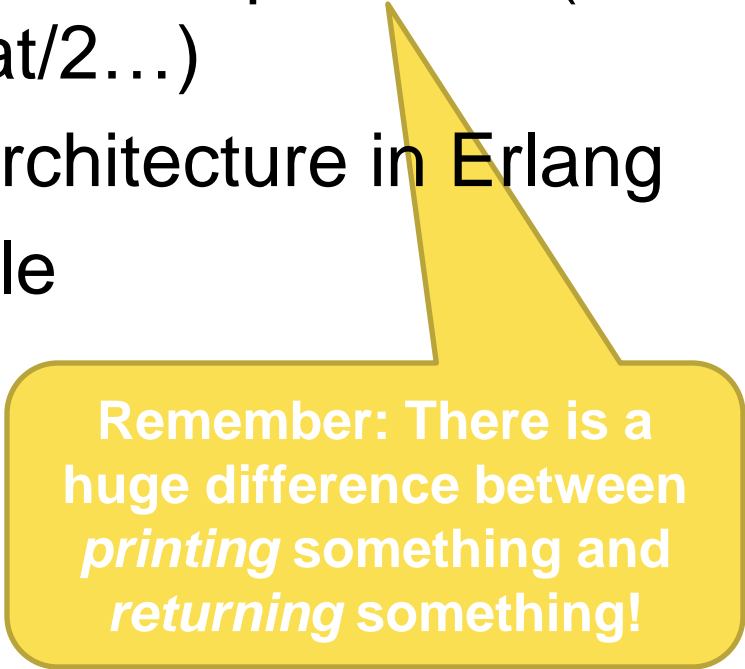


```
1> Pid = spawn(fun
dist_geometry:loop/0) .
<0.47.0>
2> Pid ! {rectangle, 4, 7}.
Area of {rectangle, 4, 7} is 28
{rectangle, 4, 7}
3> Pid ! {circle, 5}.
Area of {circle, 5} is 74.5375
{circle, 5}
4> Pid ! {triangle, 2, 4, 7}.
{triangle, 2, 4, 7}
5> Pid ! {circle, 3}.
{circle, 3}
```

The result of a send operation is the message!

The loop-process will crash!

- Ok, that works, but is not very practical. We never get the result of the area-computation! (It is just printed by the `io:format/2...`)
- Client – Server is a central architecture in Erlang
- Let's modify the area example

A yellow callout box with a black border and a tail pointing towards the first bullet point.

Remember: There is a huge difference between *printing* something and *returning* something!

- A server should return the result of the computation, but where? We need an address!

```
-export([loop/0]).  
  
loop() ->  
  receive  
    {From, Shape} ->  
      From ! area(Shape),  
      loop()  
  end.
```

Include a reply-to address
in the message!

Area example – second time



```
1> Pid = spawn(fun dist_geometry:loop/0) .  
<0.43.0>  
2> Pid ! {self(), {rectangle, 4, 7}}.  
{<0.27.0>, {rectangle, 4, 7}}  
3> receive Res -> Res end.  
28
```

The result is now received.

- Works just fine, but there is a lot of typing going on...

- We can wrap the communication in a function

```
area_rpc(Server, Shape) ->  
  Server ! {self(), Shape},  
  receive  
    Result ->  
      Result  
  end.
```

```
1> Pid = spawn(fun dist_geometry:loop/0) .  
<0.43.0>  
2> dist_geometry:area_rpc(  
    Pid, {rectangle, 4, 7}) .  
28
```

The result is received
inside area_rpc/2.

- Fine, but two potential problems:
 - We receive **any** message in area_rpc – next slide
 - We will wait indefinitely if the server crashes – receive-after (in a few slides)
-

- Use a tag to know what message to wait for

```
area_rpc(Server, Shape) ->  
  Server ! {self(), Shape},  
  receive {result, Result} ->  
    Result  
end.
```

We tag the reply with the atom result.

```
loop() ->  
  receive {From, Shape} ->  
    From ! {result, area(Shape)}  
end.
```

We must not forget to add it here as well!



- Erlang processes are extremely light-weight
 - Around 150 bytes in a data-structure
 - No operating system threads, etc. Erlang has its own scheduler.
 - The standard limit is 32767 processes, but that can easily be increased.
 - With 1 GB of memory around 300 000 processes seems to be the limit...
 - The time for a spawn is 5-10 microseconds
-

- Sometimes we do not want to wait forever for a message, we can continue after a timeout

```
receive
  Pattern1 [when Guard1] -> Exprs1;
  Pattern2 [when Guard2] -> Exprs2;
  ...
  PatternN [when GuardN] -> ExprsN
after Time ->
  Exprs
end
```

- Time can be 0 (and infinity!)

```
% A usage of after 0...
flush_mailbox() ->
    receive
        _Any ->
            flush_mailbox()
        after 0 ->
            done
    end.
```



- How receive works in more detail
 - When starting evaluation of receive, if there is an after statement, start a timer.
 - Take the **first** message in the mailbox and match it against the patterns in order (Pattern1, ...) If a match succeeds, the message is removed from the mailbox and the expressions for that pattern are evaluated (using the bindings in the pattern match).
 - If the first message does not match continue with second message, then the third, ...



- cont'ed
 - If none of the messages match, and the timer is not 0, the process is blocked (put in the wait-queue of the scheduler) until a message arrives (in which case the new message is matched against all patterns) or the timer expires in which case the after-expressions are evaluated.
- Try to keep mailbox sizes reasonably small (20-30 messages are ok...) when using selective receive. The algorithm above is expensive



- It is not always practical to keep track of Pids
 - A Pid is unique, and it is (almost) impossible to “guess” a Pid.
 - The Erlang registry
 - register(Name, Pid) – give a name (atom) to Pid
 - unregister(Name) – revoke a name from registry
 - whereis(Name) – lookup a Name
 - It is also possible to send messages directly to a named process: **Name ! Message**
 - Processes are unregistered automatically when they die.
-

- Use a start function, and register a known name

```
area_rpc(Shape) ->
  area_srv ! {self(), Shape},
  receive {result, Result} ->
    Result
end.
```

area_rpc now only needs
one argument.

```
loop() ->
  receive {From, Shape} ->
    From ! {result, area(Shape)}, loop()
end.
```

```
start() ->
  register(area_srv, spawn(fun dist_geometry:loop/0)).
```


Area example – fourth time



```
1> dist_geometry:start().  
ok  
2> dist_geometry:area_rpc({rectangle, 4, 7}).  
28
```



- To create fault-tolerant systems, we need to be able to handle crashes
 - Erlang processes can be linked together, a link is always bi-directional
 - The normal behavior when a linked process dies (exits/crashes) is to die as well
 - It is possible to instruct a process to handle crashes in linked processes
-



- When a process exits an 'EXIT' signal is sent to all linked processes
 - To receive exit signals (and not just crash) a process must 'trap exits'
 - The 'EXIT' signal also includes the *reason* of the exit, it can be used to do something clever.
 - In practice there are three strategies...
-

- I do not care if a process I create crash

```
...  
Pid = spawn(fun() -> ... end),  
...
```



- I want to Die if a process I create crashes

```
...  
Pid = spawn_link(fun() -> ... end) ,  
...
```

- `spawn_link` – does a spawn followed by a link to the resulting `Pid`.
 - The process must not trap exits
 - If `Pid` crashes with a non-normal exit the process will also crash
-

- I want to handle errors if a process I create crash

```
...
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
...
loop(...) .

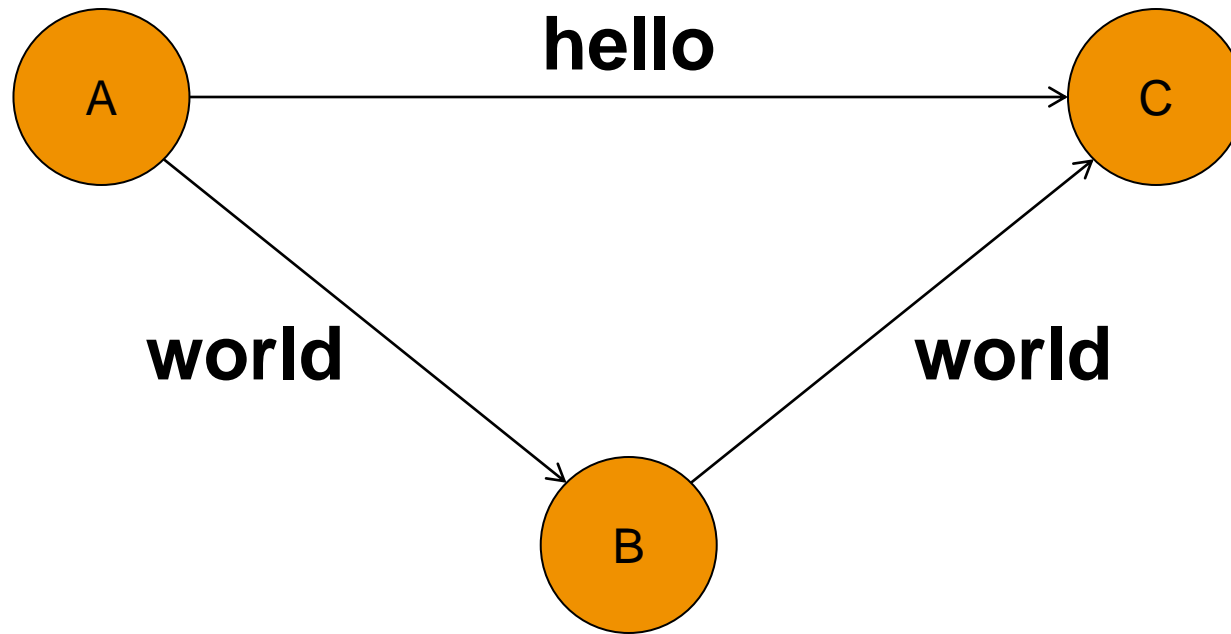
loop(State) ->
  receive
    {'EXIT', Pid, Reason} ->
      %% Do something with error
      loop(State);
  ...
end.
```

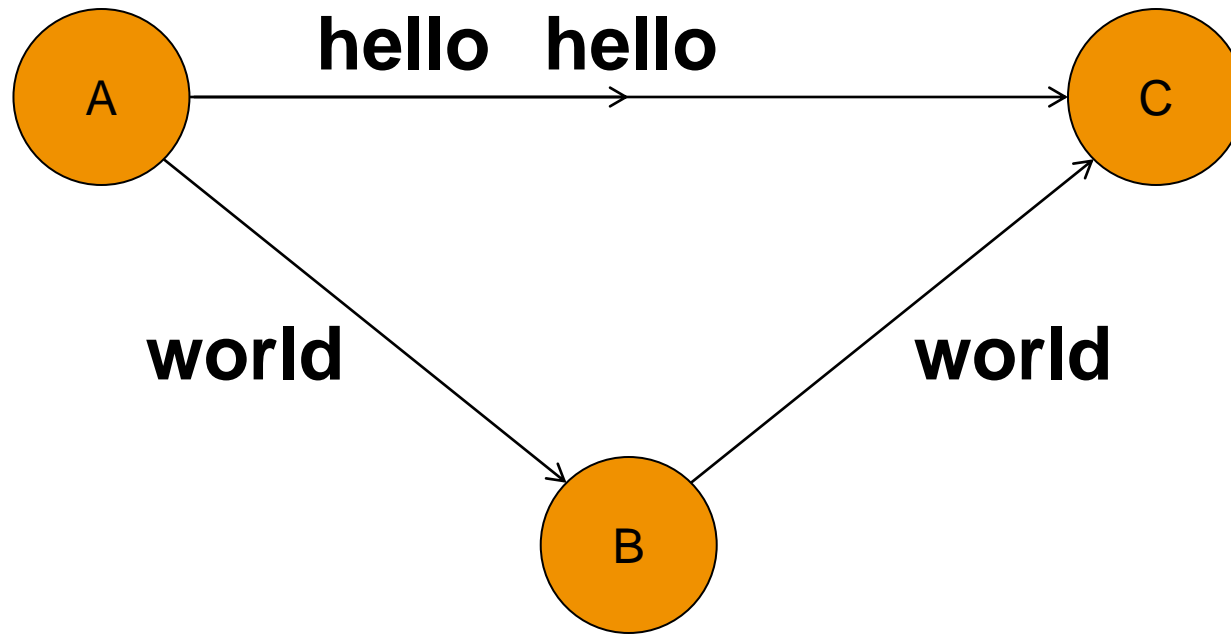


- Erlang also has **monitors**
 - a simplified version link (unidirectional)
 - `monitor(Pid)` starts monitoring of `Pid`
 - `demonitor(Pid)` removes monitoring
 - `{'DOWN', Ref, process, Pid2, Reason}` is received when the monitored process exits.
 - A common idiom is:
 1. `monitor(Pid)`
 2. Send message/task to `Pid`
 3. Wait for result/'DOWN'-message
 4. If result is received, `demonitor(Pid)`
-



- Asynchronous message passing can be tricky, consider the following scenario
 1. Process A sends message **hello** to process C
 2. Process A sends message **world** to process B
 3. Process B receives a message (**world**) and passes it along to process C
 4. Process C receives two messages, **hello** and **world**
-







...

4. Process C receives two messages, hello and world, **BUT not necessarily in that order!!**
- Order of messages is not fixed, except for between a pair of processes. Be aware!!



QUESTIONS?
