An Introduction to Erlang

Part 2 - Concurrency

Richard Carlsson
Processes

Whenever an Erlang program is running, the code is executed by a process.

The process keeps track of the current program point, the values of variables, the call stack, etc.

Each process has a unique Process Identifier (“Pid”), that can be used to find the process.

Processes are concurrent (they run in parallel).

```erlang
fib(0) -> 1;
fib(1) -> 1;
fib(N) when N > 0 ->
fib(N-1) + fib(N-2).
```
Implementation

- Erlang processes are implemented by the virtual machine, not by operating system threads
- Multitasking is preemptive (the virtual machine does its own process switching and scheduling)
- Processes use very little memory, and switching between processes is very fast
- Erlang can handle large numbers of processes
  - Some applications use more than 100,000 processes
- On a multiprocessor machine, Erlang processes can run in parallel on separate CPUs
Concurrency

- Different processes may be reading the same program code at the same time
  - They have their own data, program point, and stack – only the text of the program is being shared
  - The programmer does not have to think about other processes updating the variables

```erlang
fac(0) -> 1;
fac(N) when N > 0 -> N * fac(N-1).
```
Message passing

- “!” is the send operator (often called “bang!”)
  - The Pid of the receiver is used as the address
- Messages are sent asynchronously
  - The sender continues immediately
- Any value can be sent as a message
• Each process has a *message queue* (mailbox)
  - Arriving messages are placed in the queue
  - No size limit – messages are kept until extracted
• A process *receives* a message when it extracts it from the mailbox
  - Does not have to take the first message in the queue
Receiving a message

- receive-expressions are similar to `case` switches
  - Patterns are used to match messages in the mailbox
  - Messages in the queue are tested in order
    - The first message that matches will be extracted
    - A variable-pattern will match the first message in the queue
  - Only one message can be extracted each time

```
receive
    Msg -> io:format("~w\n", [Msg])
end
```
Selective receive

```
receive
  {foo, X} -> ...;
  {bar, X} when ... -> ...;
  ... 
end
```

- Patterns and guards let you select which messages you currently want to handle
  - Any other messages will remain in the mailbox
- The `receive`-clauses are tried in order
  - If no clause matches, the next message is tried
- If *no* message in the mailbox matches, the process *suspends*, waiting for a new message
Receive with time-out

A receive-expression can have an after-part

- The time-out value is either an integer (milliseconds), or the atom 'infinity' (wait forever)
- 0 (zero) means “just check the mailbox, then continue”

The process will wait until a matching message arrives, or the time-out limit is exceeded

- Soft real-time: approximate, no strict timing guarantees

```erlang
receive
  {foo, X} -> ...;
  {bar, X} -> ...;
after 1000 ->
  ...       % handle timeout
end
```
Pids are often included in messages (`self()`), so the receiver can reply to the sender

- If the reply includes the Pid of the second process, it is easier for the first process to recognize the reply

```
P1 ! {hello, self()},
receive
  {reply, Pid, String} ->
    io:put_chars(String)
end

{hello,P1}  
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P2</td>
</tr>
</tbody>
</table>

{reply,P2,"Hi!"}

receive
  {hello, Sender} ->
    Sender ! {reply, self(), "Hi!”}
end
```
• The only guaranteed message order is when both the sender and receiver are the same for both messages (First-In, First-Out)
  – In the left figure, m1 will always arrive before m2 in the message queue of P2 (if m1 sent before m2)
  – In the right figure, the arrival order can vary
Selecting unordered messages

- Using selective receive, we can choose which messages to accept, even if they arrive in a different order.

- In this example, P2 will always print “Got m1!” before “Got m2!”, even if m2 arrives before m1.
  - m2 will be ignored until m1 has been received.

```
P1 P2

P3

receive m1 -> io:format("Got m1!")
end,
receive m2 -> io:format("Got m2!")
end
```
Starting processes

• The 'spawn' function creates a new process

• There are several versions of 'spawn':
  – spawn( fun() -> ... end )
    • can also do spawn( fun f/0 )
  – spawn( Module, Function, [Arg1, ..., ArgN] )
    • Module:Function/N must be an exported function

• The new process will run the specified function

• The spawn operation always returns immediately
  – The return value is the Pid of the new process
  – The “parent” always knows the Pid of the “child”
  – The child will not know its parent unless you tell it
A process *terminates* when:
- It finishes the function call that it started with
- There is an exception that is not caught
  - The purpose of 'exit' exceptions is to terminate a process
  - "exit(normal)" is equivalent to finishing the initial call
- All messages sent to a terminated process will be thrown away, without any warning
  - No difference between throwing away and putting in mailbox just before process terminates
- The same process identifier will not be used again for a long time
A stateless server process

```erlang
run() ->
    Pid = spawn(fun echo/0),
    Pid ! {hello, self(), 42},
    receive
        {reply, Pid, 42} ->
            Pid ! stop
    end.
```

```erlang
echo() ->
    receive
        {hello, Sender, Value} ->
            Sender ! {reply, self(), Value},
            echo(); % loop!
        stop ->
            ok
    end.
```

Client P1 ➔ P2 ➔ Server

{hello,P1,42} ➔ {reply,P2,42} ➔

A server process with state

server(State) ->
    receive
        {get, Sender} ->
            Sender ! {reply, self(), State},
            server(State);
        {set, Sender, Value} ->
            Sender ! {reply, self(), ok},
            server(Value);  %% loop with new state!
        stop ->
            ok
    end.

- The parameter variables of a server loop can be used to remember the current state.
- Note: the recursive calls to server() are tail calls (last calls) – the loop does not use stack space.
- A server like this can run forever.
Hot Code Swapping

-module(server).
-export([[start/0, loop/1]]).

start() -> spawn(fun() -> loop(0) end).

loop(State) ->
    receive
        {get, Sender} ->
            ...
        {set, Sender, Value} ->
            ...
        server:loop(State);
        server:loop(Value);
    ...

• When you use “module:function(...), Erlang will always call the latest version of module
  – If you recompile and reload the server module, the process will jump to the new code after handling the next message – you can fix bugs without restarting!
Hiding message details

- Using interface functions keeps the clients from knowing about the format of the messages
  - You may need to change the message format later
- It is the client who calls the self() function here

```erlang
get_request(ServerPid) ->
    ServerPid ! {get, self()}. 

set_request(Value, ServerPid) ->
    ServerPid ! {set, self(), Value}. 

wait_for_reply(ServerPid) ->
    receive
    {reply, ServerPid, Value} -> Value
    end.

stop_server(ServerPid) ->
    ServerPid ! stop.
```
Registered processes

- A process can be registered under a name
  - the name can be any atom
- Any process can send a message to a registered process, or look up the Pid
- The Pid might change (if the process is restarted and re-registered), but the name stays the same

```
Pid = spawn(...),
register(my_server, Pid),
my_server ! {set, self(), 42},
42 = get_request(my_server),
Pid = whereis(my_server)
```
Any two processes can be linked
  - Links are always bidirectional (two-way)

When a process dies, an exit signal is sent to all linked processes, which are also killed
  - Normal exit does not kill other processes
Trapping exit signals

- If a process sets its `trap_exit` flag, all signals will be caught and turned into normal messages
  - `process_flag(trap_exit, true)`
  - `{EXIT', Pid, ErrorTerm}`

- This way, a process can watch other processes
  - 2-way links guarantee that sub-processes are dead
Robust systems through layers

- Each layer supervises the next layer and restarts the processes if they crash.
- The top layers use well-tested, very reliable libraries (OTP) that practically never crash.
- The bottom layers may be complicated and less reliable programs that can crash or hang.
Running “erl” with the flag “-name xxx”
- starts the Erlang network distribution system
- makes the virtual machine emulator a “node”
  - the node name is the atom 'xxx@host.domain'

Erlang nodes can communicate over the network
- but first they must find each other
- simple security based on secret cookies
Connecting nodes

- Nodes are connected the first time they try to communicate – after that, they stay in touch
  - A node can also supervise another node

- The function “net_adm:ping(Node)” is the easiest way to set up a connection between nodes
  - returns either “pong” or “pang” :-)

- You can also send a message to a registered process using “{Name,Node} ! Message”
Distribution is transparent

- You can send a Pid from one node to another
  - Pids are unique, even over different nodes
- You can send a message to *any* process through its Pid – even if the process is on another node
  - There is no difference (except that it takes more time to send messages over networks)
  - You don't have to know where processes are
  - You can make programs work on multiple computers with no changes at all in the code (no shared data)
- You can run several Erlang nodes (with different names) on the same computer – good for testing
Running remote processes

P = spawn('barney@foo.bar.se', Module, Function, ArgList),
global:register_name(my_global_server, P),
global:send(my_global_server, Message)

- You can use variants of the `spawn` function to start new processes directly on another node
- The module `global` contains functions for
  - registering and using named processes over the whole network of connected nodes
    - not same namespace as the local “register(...)”
    - must use “global:send(...)”, not “!”
  - setting global locks
Ports – talking to the outside

- Talks to an external (or linked-in) C program
- A port is connected to the process that opened it
- The port sends data to the process in messages
  - binary object
  - packet (list of bytes)
  - one line at a time (list of bytes/characters)
- A process can send data to the port

```
PortId = open_port({spawn, “command”}, [binary]),
PortId ! {self(), {command, Data}}
PortId ! {self(), close}
```
End

Sorry, no more stuff this time