Concurrency in Erlang

1st Semester, A.Y. 2020-2021

1 Concurrency

Concurrency refers to the idea of having many actors (processes) running independently, but not necesarily all at the same time. In Erlang, there are three primitives required for concurrency: *spawning new processes, sending messages, and receiving messages.*

Processes can be broadly defined as functions and once it's done executing, it disappears.

A. Creating Processes

Example of creating a process:

```
1> F = fun() -> 4 + 3 end.    %create a function assigned to F
#Fun<erl_eval.20.80484245>
2> spawn(F).                  %executes the function
<0.48.0>
```

`(1)` creates a function using the `fun()` function. The `fun()` function is an anonymous function that can be declared on the fly. You can declare them inline without naming them.

`(2)` uses the `spawn/1` function of Erlang which takes a single function as parameter and runs it.

The result of the `spawn/1` function (`<0.48.0>`) is called a **Process Identifier** (PID, Pid, pid) which is an arbitrary value representing any process.

We did not see the result of the function `F` and was only able to see the `pid`. This is because processes do not return anything.

B. Sending Messages

To send messages from one process to another, we use the the `!` operator (known as `bang`). Any Erlang term on the right-hand side is sent to the process represented by the `pid` in the left-hand side.

```
3> self() ! {hello,world}.    %sends to the current shell
{hello,world}
4> self() ! "Hello World!".   %sends a string
"Hello World!"
```

`self/1` is a function that returns the `pid` of the current process, which in this case, is the Erlang shell.

The message that was sent using the `!` operator has been put in the process' **mailbox**, but it hasn't been read yet. The output shown is the return value of the send operation.

Every message sent to a process is kept in the process' mailbox in the order they are received. Every time a message is read, it is taken out of the mailbox.

To see the contents of the current mailbox, you can use the `flush/0` function while in the shell.

```
5> flush().                  %prints the contents of the mailbox
Shell got {hello,world}
Shell got "Hello World"
ok
```

The `flush` function is just a shortcut that outputs received messages. We still can't bind the result of a process to a variable.

C. Receiving Messages

To get a message that was sent to the process, we need to use the `receive` statement.

```
% Filename should be: cats.erl
-module(cats).              %filename
-compile(export_all).       %tells which functions to compile

cat() ->
   receive                  %waits for the mailbox to have a message
      come_here ->
         io:format("Shut up, human.~n");
      catfood ->
         io:format("Thank you!~n");
      _ ->                  %if message is not come_here or not catfood
         io:format("I'm your master.~n")
   end.
```

Save the code above in a file named `cats.erl` then compile it as follows:

```
6> c(cats).
{ok,cats}
7> Cat = spawn(cats,cat,[]). %start cat function from cats.erl with no params
<0.63.0>
8> Cat ! catfood.            %send catfood to Cat
Thank you!
catfood
9> Cat ! come_here.
come_here
```

The `spawn/3` function `(7)` takes three parameters: the module name, the function name of the function to be spawned, and the arguments of the function to be spawned. It then returns the `pid` of the new process.

Once the function is running, the following events take place:

1. The function hits the `receive` statement. Since the process' mailbox is empty, the cat will now wait until it gets a message `(7)`.
2. The message `catfood` is received. The function tries to pattern match against `come_here`. It fails and now tries at `catfood`. This pattern now matches `(8)`.
3. The process outputs the message `Thank you!`.

Notice that in the first message, the cat replied. However, the second message was ignored. This is because the function terminated after the output. Thus, we need to restart the cat.

```
10> f(Cat).
ok
11> Cat = spawn(cats,cat,[]).
<0.90.0>
12> Cat ! come_here.
Shut up, human.
come_here
```

D. Sending Back Messages

On our previous code, the spawned process does not send back a message to us, it just prints a message using the `io:format/2`. To allow the program to give us a reply, we need to give our own address.

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

cat() ->
   %edit this part
   receive
```

```
        {From, come_here} ->  %receives a tuple
            From ! "Shut up, human.~n";
        {From, catfood} ->
            From ! "Thank you!~n";
        _ ->
            io:format("I'm your master.~n")
    end.
```

If we try to execute this program:

```
13> c(cats).
{ok,cats}
14> Cat2 = spawn(cats,cat,[]).
<0.108.0>
15> Cat2 ! {self(),catfood}. %sends the PID of sender and the message
{<0.106.0>,catfood}.
16> flush().
Shell got "Thank you!~n"
ok
```

Though we can get a reply now, we still need to restart the cat every time we try to send a message. To solve this problem, we will use recursion.

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

cat() ->
    %edit this part
    receive
        {From, come_here} ->  %receives a tuple
            From ! "Shut up, human.~n",
            cat();
        {From, catfood} ->
            From ! "Thank you!~n";
        _ ->
            io:format("I'm your master.~n"),
            cat()
    end.
```

Now, our cat will always give us a reply except for when it receive catfood where it leaves us.

```
17> c(cats).
{ok,cats}
18> Cat3 = spawn(cats,cat,[]).
<0.108.0>
19> Cat3 ! come_here.
I'm your master.
come_here
20> Cat3 ! {self(), come_here}.
{<0.106.0>,come_here}
21> Cat3 ! {self(), come_here}.
{<0.106.0>,come_here}
22> Cat3 ! {self(), kitty_kitty}.
I'm your master.
{<0.106.0>,kitty_kitty}
23> Cat3 ! {self(), catfood}.
{<0.106.0>,catfood}
24> flush().
Shell got "Shut up, human.~n"
Shell got "Shut up, human.~n"
Shell got "Thank you!~n"
ok
25> Cat3 ! {self(), come_here}.
```

```
{<0.106.0>,come_here}
26> flush().
ok
```

2 Distributed Computing

Each Erlang virtual machine that is up and running is called a *node*. A node can connect to other nodes running on a single computer/host or to other nodes running on other computers.

Whenever you start a node, you give it a name and it will connect to an application called **EPMD (Erlang Port Mapper Daemon)**, which will run on each of the computers which are part of your Erlang cluster. EPMD will act as a name server that lets nodes register themselves, contact other nodes, and warn you about name clashes if there are any.

The node name follows the format `name@hostname`, where `name` is the name given by the user, and `hostname` is the name of the host PC.

There are two types of name you can give to a node:

1. **long names**
   Can be given using the `-name` option and follows the full `name@hostname` format.
2. **short names**
   Can be given using the `-sname` option and follows the name of the node without a host.

```
% exit erlang and then re-enter the shell
% USING THE -name OPTION
$ erl -name 'dolphin@ocean'
(dolphin@ocean)1> node().
dolphin@ocean
```

```
% exit erlang and then re-enter the shell
% USING THE -sname OPTION
$ erl -sname 'dolphin'
(dolphin@ics-user)1> node().
dolphin@ics-user
```

Remember!
A node with a long name **cannot** communicate with a node with a short name.

A. Single Host

There are a lot of function to connect nodes. One example is the `net_adm:ping/1` function.

```
% ON TERMINAL 1
$ erl -sname kei
(kei@ics-user)1>
```

```
% ON TERMINAL 2
$ erl -sname kat
(kat@ics-user)1> net_adm:ping('kei@ics-user').
pong
(kat@ics-user)2> nodes().
[kei@ics-user]
```

Using the `nodes/0` function above, we can see which nodes are connected to the current node we are using.

Now that the nodes are connected, we can now make processes communicate with one another. We can pass messages to {`Name, Node`} where `Name` is the registered name of a process using the `register/2` function.

```
% Type and save this file with the filename: pingpong.erl
-module(pingpong).
-compile(export_all).

start_pong() ->
  register(pong,spawn(pingpong,pong,[])).

pong() ->
  receive
    finished ->
      io:format("Pong finished~n");
    {ping,Ping_Pid} ->
      io:format("Pong got ping~n"),
      Ping_Pid ! pong,
      pong()
  end.

start_ping(Pong_Node) ->
  spawn(pingpong,ping,[3,Pong_Node]).

ping(0,Pong_Node) ->
  {pong, Pong_Node} ! finished,
  io:format("Ping finished~n");
ping(N,Pong_Node) ->
  {pong, Pong_Node} ! {ping,self()},
  receive
    pong ->
      io:format("Ping got pong~n")
  end,
  ping(N-1,Pong_Node).
```

Compile the file in the terminals we've initialized. Let's say you are using the `kat@ics-user` shell.

```
(kat@ics-user)3> cd("/path/to/file"). %if the file is somewhere else
/path/to/file
ok
(kat@ics-user)4> c(pingpong).
{ok,pingpong}
(kat@ics-user)5> pingpong:start_pong().
true
```

And then on the other terminal (`kei@ics-user`), do this:

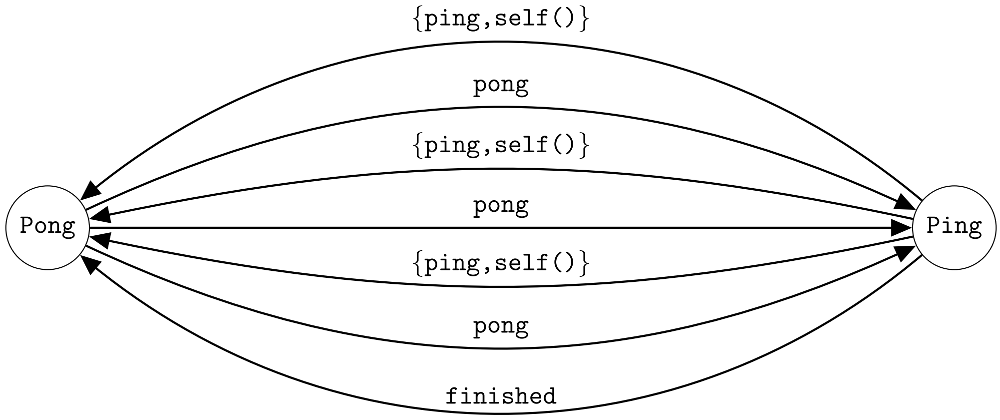
```
(kei@ics-user)3> cd("/path/to/file"). %if the file is somewhere else
/path/to/file
ok
(kei@ics-user)4> c(pingpong).
{ok,pingpong}
(kei@ics-user)5> pingpong:start_ping('kat@ics-user').
<0.56.0>
```

The `pingpong:start_pong/0` function creates a new process `pingpong:pong/0` and registered it as `pong`.

The `pingpong:pong/0` will wait for a message. If it receives the atom `finished`, it will output `"Pong finished"` then terminate. If it is a tuple `ping, Ping_Pid`, it will output `"Pong got ping"` and replies to the sender with a `pong` message. Then it will call the `pong/0` function again to wait for another message.

For the `pingpong:start_ping/1` function, we gave it the node name of the other node. It creates a new process `pingpong:ping/2` and gave the number `3` and `Pong_Node` as arguments.

If the first argument of `pingpong:ping/2` is `0`, it will send `finished` to the `Pong_Node` and output `"Ping finished"`. If it is not yet `0`, it will send the message `ping, self()` to the `Pong_Node` and waits for a reply. If the `Pong_Node` replies, it will output `"Ping got pong"` and makes a recursive call `ping(N - 1, Pong_Node)`.



B. Multiple Hosts

If we try to connect nodes from different hosts, we get an error message. This is because we can only connect nodes that have the same cookie. We can check the cookie using `erlang:get_cookie/0`.

```
(kei@ics-user)6> erlang:get_cookie().
'WFABWDTLNKYJTRVKFCRR'
```

To specify the cookie of a node, we can start the node while adding the `-setcookie` option in the terminal.

```
% ON TERMINAL 1
$ erl -name 'kat@10.0.3.120' -setcookie dota
(kat@10.0.3.120)1>
```

```
% ON TERMINAL 2
$ erl -name 'kei@10.0.3.189' -setcookie dota
(kei@10.0.3.189)1>
```

Now, we can connect the two nodes.

```
(kei@10.0.3.189)1> net_adm:ping('kat@10.0.3.120').
pong
```

3 User Input

To ask for input from the user, we can use the `io:get_line/1` function.

```
1> String = io:get_line('Name please: ').
Name please: natalie
"natalie\n"
2> String.
"natalie\n"
```

References

[1] Ericsson, A. *Erlang Reference Manual User's Guide*, 2018.

[2] Hébert, F. Learn You Some Erlang (for great good!).

[3] Tandoc, M. *CMSC 124 Erlang Handout 2*, 2013.