ID2201 Distributed Systems, basic course

An Erlang primer

Johan Montelius, Vladimir Vlassov and Klas Segeljakt
Email: {johanmon,vladv,klasseg}@kth.se
July 7, 2023

Introduction

This is not a crash course in Erlang since plenty of tutorials are available on the web. I will, however, describe the tools you need to get a programming environment up and running. We will take for granted that you know some programming languages, have heard of functional programming, and that recursion is not a mystery to you.

1 Getting Started

If you run on your computer, you need to install the Erlang development environment. This is probably already done if you're running on one of the KTH computers.

The first thing you need to do is download the Erlang SDK. This is found at www.erlang.org and is available as a Windows binary and a source you can use with the usual tools available on a Unix or macOS platform. If you use Ubuntu, the Erlang system is available in the repository.

The SDK includes a compiler, all the programming libraries, and a virtual machine. It does not include a development environment. You can also download the Erlang manual in HTML.

1.1 Erlang

When you have installed Erlang, you should be able to start an Erlang shell. You do this by starting Erlang, found in the regular Windows program listing, or by hand from a regular shell. Since you will need to set runtime parameters later, you need to learn how to start Erlang by hand. Open a regular shell and type:

```
erl
```

This should start the Erlang shell, and you should see something like this:

```
Erlang (BEAM) emulator version 5.6.1 [source] [smp:4]
[async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.6.1  (abort with ^G)
1>
```

Type `help().` (note the dot) followed by return to see all the shell commands and `halt().` to exit the shell.

1.2 Emacs

You need a text editor, e.g., Emacs, as a development environment. If you want to use Emacs, download it from www.gnu.org/software/emacs/, available for Unix, MacOS, and Windows. Ubuntu users will find it in the repository.

You need to add the code below to your `.emacs` file (provided that you have Erlang installed under `C:/Program Files/`). This will ensure that the Erlang mode is loaded when you open a `.erl` file and can start an Erlang shell under Emacs etc. Change the `<Ver>` and `<ToolsVer>` to what is right in your system. It will look similar on Linux, but the install directory is something like `/usr/local/lib/erlang/`.

Note! If you cut and past this text, the '-character will not be a '-character if you see what I mean. When you load the file in emacs, you will have errors. If you cut and paste, write in by hand "'erlang-start". This is true for all cut and paste from pdf documents; things might not be what they appear to be, so be careful.

```
(setq load-path
  (cons
    "C:/Program Files/erl<Ver>/lib/tools-<ToolsVer>/emacs"
    load-path))
(setq erlang-root-dir
    "C:/Program Files/erl<Ver>")
(setq exec-path
  (cons
     "C:/Program Files/erl<Ver>/bin"
     exec-path))
(require 'erlang-start)
```

You must find your emacs home directory where the file should be placed. If you're on the KTH student computers, the home area is "h:", but you need to set the "HOME" environment variable so that emacs finds its way.

If everything works, you should be able to start an Erlang shell inside Emacs by `M-x run-erlang` (`M-x` is < *escape* > followed by x). A shell inside Emacs will allow you to compile and run programs quickly. Still, when we experiment with distributed applications, you need to run these in separate shells, so make sure that you also know how to start an Erlang shell manually.

1.3 Eclipse

If you prefer to use Eclipse, you can install an Erlang plugin and do your development inside Eclipse. Feel free to use whatever environment you want.

2 Hello World

Open a file `hello.erl` and write the following:

```
-module(hello).

-export([world/0]).

world()->
  "Hello world!".
```

Now open an Erlang shell, compile, and load the file with the command `c(hello).` and, call the function `hello:world()`.. Remember to end commands in the shell with a dot. If things work out, you have successfully written, compiled, and executed your first Erlang program.

Find the Erlang documentation and read the "Getting Started" section.

3 Concurrent Programming

Erlang was designed for concurrent programming. You will quickly learn how to divide your program into communicating processes, giving it a far better structure. Try the following:

```
-module(wait).
-export([hello/0]).

hello() ->
    receive
      X -> io:format("aaa! surprise, a message: ~s~n", [X])
    end.
```

The `io:format` procedure will output the string to the stdout and replace the control characters (characters preceded by a tilde) with the elements in the list. The `s` means that the next element in the list should be a string, `n` outputs a newline. Load the above module and execute the command:

```
P = spawn(wait, hello, []).
```

The variable `P` is now bound to the *process identifier* of the spawned process. The process was created and called the procedure `hello/0` (this is how we name a function with zero arguments). It is now suspended, waiting for incoming messages. In the same Erlang shell, execute the command:

```
P ! "hello".
```

This will send a message, in this case, a string "hello", to the process that now wakes up and continues the execution.

To make life easier, one often registers the process identifiers under names that all processes can access. If two processes should communicate, they must know the process identifier. Either a process is given the identifier to the other process when it is created, in a message, or through the registered name of the process. Try this in a shell (first type `f()` to make the shell forget the previous binding to P):

```
P = spawn(wait, hello, []).
```

Now register the process identifier under the name "foo".

```
register(foo, P).
```

And then send the process a message.

```
foo ! "hello".
```

In this example, we only sent a string, but we can send arbitrary complex data structures. The `receive` statement can have several clauses that try to match incoming messages. Only if a match is found will a clause be used. Try this:

```
-module(tic).
-export([first/0]).

first() ->
    receive
      {tic, X} ->
          io:format("tic: ~w~n", [X]),
          second()
```

```
    end.

second() ->
    receive
      {tac, X} ->
          io:format("tac: ~w~n", [X]),
          last();
      {toe, X} ->
          io:format("toe: ~w~n", [X]),
          last()
    end.

last() ->
    receive
      X ->
          io:format("end: ~w~n", [X])
    end.
```

Then in a shell, execute the following commands:

```
P = spawn(tic, first, []).

P ! {toe, bar}.

P ! {tac, gurka}.

P ! {tic, foo}.
```

In what order did the process receive them? Note how messages are queued and how the process selects in what order to process them.

4 Distributed Programming

Distributed programming is extremely easy in Erlang; the only problem we will have is finding the name of our node. The Erlang distributed systems normally use domain names rather than explicit IP addresses. This could be a problem since we're working with laptops that are not regularly given names in the DNS. There is a workaround for this that we will use.

Connect to the WLAN, login, and ensure you have access to the Internet. Run `ipconfig` or `ifconfig` to find out what IP address you have been allocated. You will use this IP address explicitly when starting an Erlang node.

4.1 Node name

When you start Erlang, you can make it network-aware by providing a name. You also would like to give it a secret cookie. Any node that can prove to know the cookie will be trusted to do just about anything. This is quite dangerous, and it's very easy for a malicious node to close a whole network down.
Start a new Erlang shell with the following command, replacing the IP address with whatever you have.

```
erl -name foo@130.237.250.69 -setcookie secret
```

In the Erlang shell, you can now find the name of your node with the bif `node()`. It should look something like `'foo@130.237.250.69'`.

Doing the same if you're running Erlang under Emacs is slightly more tricky. You have to set a lisp variable used when Emacs starts Erlang. Type `M-x eval-expression` and evaluate the function.

```
(set 'inferior-erlang-machine-options
    '("-name" "foo@130.237.250.69" "-setcookie" "secret"))
```

You can also set the environment variable `ERL_FLAGS` to the same string or include it in your `.emacs` file. This will, however, prevent you from running multiple Erlang shells on the same node. You must also change this every time you get a new IP address.

Start a second Erlang shell using the name `bar` on the same or another machine. Load and start the suspending hello process we defined on the *foo-node* and register it under `wait`.

Now on the *bar-node* try the following:

```
{wait, 'foo@130.237.250.69'} ! "a message from bar".
```

Note how interprocess communication in Erlang is handled in the same way regardless if the process is executing in the same shell, another shell, or on another host. The only thing that must be changed is how the Erlang shell is started and how to access external registered processes.

If you run on a computer that cannot open communication ports, you will need to run Erlang in local distribution mode. You then start Erlang with a short name as follows:

```
erl -sname foo -setcookie secret
```

The rest is the same, but use foo as the name without the IP address.

4.2 ping-pong

It would not be a transparent system if we could only send messages to registered processes. We can send messages to any process, but the problem is to get hold of the process identifier. There is no way to write this down, and you cannot use the cryptic "<0.70.0>" that the Erlang shell uses when it prints the value of a process identifier.

The only processes that know the process identifier of a process are the creator of the process and the process itself. The process can find its identifier by the built-in procedure `self()`. Now we can pass the process identifiers around and even send them in a message to let other processes know. Once a process knows an identifier, it can use it without knowing if it is a local or remote process identifier.

Try to define a process on one machine, `ping`, that sends a message to a registered process on another machine with its process identifier in the message. The process should the wait for a reply. The receiver on the other machine should look at the message and use the process identifier to send a reply.

A world of warning, the send primitive `!` will accept both registered names, remote names, and process identifiers. This can sometimes cause a problem. If you implement a concurrent system that is not distributed and has a registered process under the name `foo`, you could pass the atom `foo` around, and anyone could treat it as a process identifier. Now if we distribute this application, a remote process will not be able to use it as a process identifier since the registration process is local to a node. So keep track of what you pass around: is it a process identifier (that can be used remotely), or is it a name under which a process is registered?