An Erlang primer

Johan Montelius

February 12, 2013

Introduction

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

1 Getting started

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

The first thing you need to do is download the Erlang SDK. This is found at www.erlang.org and is available both as a Windows binary and as source that you can make with the usual tools available on a Unix or even MacOs platform. If your using for example Ubuntu then the Erlang system is avialable in the repository.

The SDK includes a compiler, all the programming libraries and 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 either by starting Erlang found in the regular Windows program listing or by hand from a regular shell. Since you later will need to set runtime parameters 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 someting 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

As an development environment you need a text editor and what is better than Emacs. Download it from www.gnu.org/software/emacs/, available both for Unix, Mac 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 make sure that the Erlang mode is loaded as soon as you open a `.erl` file and that you can start an Erlang shell under Emacs etc. Change the `<Ver>` and `<ToolsVer>` to what is right in your system. On Linux it will look similar 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 do cut and paste then write in by hand "'erlang-start". This is true for all cut and paste from pfd 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 will have to find your emacs home directory where the file should be placed. If you're on the KTH student computers then the home area is "h:" but you then 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 quickly compile and run programs but when we experiment with distributed applications you need to run these in separate shells so make sure that you also know how start an Erlang shell manually.

1.3 Eclipse

If you prefer to use Eclipse you can install a Erlang plugin and do your development inside Eclipse. Fell 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 a Erlang shell, compile and load the file with the command `c(hello).` and, call the function `hello:world().`. Remember to end commands in th 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 and thereby give it 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` simply 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 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 register the process identifiers under names that can be access by all processes. 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 the only thing we sent was 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 where they received by the process. 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 work using domain names rather than explicit IP addresses. This could be a problem since we're working with a set of laptops that are not regularly given names in the DNS. There is a work around for this that we will use.

Connect to the WLAN, login and make sure that you have a 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

have knowledge of the cookie will be trusted to do just about anything. This is of course 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 to 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 that is 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 would also have to change this every time you get a new IP address.

Start a second Erlang shells now using the name `bar` on the same or another machine. Load and start the suspending hello process that we defined before on the *foo-node* and register it under the name `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 has to be changes is how the Erlang shell is started and how to access external registered processes.

If you run on a computer where you can not open ports for communication 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

If we could only send messages to registered processes it would not be a transparent system. We can send messages to any process but the problem is of course to get hold of the process identifier. There is no way to write

this down and you can not 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 is the creator of the process and the process itself. The process it self can find out by the built-in procedure `self()`. Now we are of course allowed to pass the process identifiers around and even send them in a message so we can let other processes know. Once a process knows it can use the identifier and does not have to know if it is a local or remote process.

Try to define a process on one machine `ping`, that sends a message to a registered process on another machine with its own 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 sometime cause a problem. If you implement a concurrent system that is not distributed and have a registered processes 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.