Comparative Programming Languages
Erlang Homework

October 24, 2014

This homework will guide you through some aspects of the Erlang programming language. This homework is split in two steps: the first one will recap the exercises presented in class, the second one contains what future exercise sessions in the lab will build upon. To complete the exercises, download the following file from toledo: `CPL-2014-Erlang.zip`.

The directory `Es1` contains these files, which are used to complete the first exercise:

- `esOne.erl` — contains the functions you need to expand.
- `srvEsOne.erl` — contains a server that interacts with processes implementing functions of `esOne.erl`.

To complete the second exercise, inspect directory `Es2`, it contains the following files:

- `srv.erl` — the implementation of all the game servers.
- `dummy.erl` — a dummy client which will form the basis of the second exercise.

Note, these will be used also in future exercise sessions in the lab.

1 General remarks

Useful links. Before starting, here are some useful links related to Erlang.

- Firstly, the link to the Erlang API and their usage: http://www.erlang.org/doc/man_index.html
- Secondly, the link to Part I of *Concurrent programming in Erlang*, a very useful book by J. Armstrong, R. Virding, C. Wikström and M. Williams: http://www.erlang.org/download/erlang-book-part1.pdf
- Then a series of links to getting-started pages and examples: http://www.erlang.org/static/getting_started_quickly.html
- http://www.erlang.org/doc/getting_started/users_guide.html
- http://learnyousomeerlang.com/

Compilation and execution of programs. In order to compile an Erlang file called `foo.erl`, type the command `erlc foo.erl` from the command line. It will generate a binary file called `foo.beam` in the same folder as `foo.erl`. In order to execute Erlang binaries, load the Erlang console by executing command `erl`. From within the console for can execute functions. If file `foo.erl` defines module `foo` and function `bar`, you can execute `foo:bar().` in order to call function `bar` from within the Erlang console. Notice that you need to complete the function call with a dot at the end. From within the console you can also compile files, type `c(foo.erl).` in order to recompile file `foo.erl`, in this way you will be able to use auto-completion for calling functions defined in `foo.erl`.

Common mistakes. One of the greatest source of errors in Erlang code is not understanding the variable convention. Variables are defined using **uppercase**; thus `Var` is a variable and `var` is not. Erlang uses single assignment, meaning that each variable can only be bound once; thus the following code is invalid: `Var=3, Var=Var+1`. **Lowercase** words are called *atoms*. They are global, non-numerical, constant values (like symbols in Ruby).

Terminology. The bodies of the exercises may contain some of the following expressions, here is the intended meaning. "send an `atom` message" means that the process must sent a message whose structure is {`atom, ...`}. Analogously, "receive an `atom` message" means that the process must implement a receive (or a receive-loop) that includes a pattern of the form {`atom, ...`}. "attack" means that the process must send a message {`atk, ...`}.

2 Recap from the lecture

Following are some of the exercises that were solved in class. Re-solve them here to warm up before the next exercise.

For this batch of exercises you will use files from the `Es1` directory; you will need to modify file `exOne.erl`. In order to test whether your implementation is correct, compile your sources and execute `srvEsOne:start().` from the Erlang shell. Right now both file compile and if you execute `srvEsOne:start().`, you see a message from the `Server`, one from the `Client` and then the `Server` disconnects since it does not receive messages.

2.1 Spawning processes

Processes are created with the built-in function (BIF) `spawn`. A call to `spawn` returns the process identifier (PID) of the spawned process. The PID can be used to send messages to the corresponding process.

Modify function `start` in `esOne.erl` so that it spawns a new process executing function `func` with one argument: the PID of the process. A process can obtain its own PID via the BIF `self()`.

Execute `srvEsOne:start().`, in addition to the previous messages you should see a message from the `Child`. The `Server` will disconnect anyway.

2.2 Sending messages

Message sending is performed using syntax `Target ! Message`, where `Target` is either a PID or a registered process name. Any kind of data can be sent as a message, including integers `14`, atoms `foobar`, and tuples `{12, "String"}`.

Modify function `start` in `esOne.erl` so that it sends a message of the form {`onPid, P`} to the PID of the process that spawned it, which is stored in variable `Par`. The argument `P` of the message is the PID of the process sending it.

Execute `srvEsOne:start().`, in addition to the previous messages you should see a message form the `Server` saying that it received a message from the `Client`. The `Server` will disconnect anyway.

2.3 Sending messages to registered processes

Erlang provides a mechanism for registering processes to make it possible to send a message to a process whose PID is unknown. The BIF `register(name,Pid)` associates the process with pid `Pid` with the atom `name`, so that sending a message to `name` has the same effect as sending a message to `Pid`. It is good programming practice to check whether a name is valid before registering it. The list of all registered names can be obtained with the `registered` BIF.

Modify function `start` in `esOne.erl` so that it sends a message of the form {`onName, P`} to the PID of the process that spawned it, which is stored in variable `Par`. The argument `P` of the message is the PID of the process sending it.

Execute `srvEsOne:start().`, in addition to the previous messages you should see a message form the `Server` saying that it received another message from the `Client`. The `Server` will disconnect anyway.

2.4 Receiving messages

Receiving messages is done using the construct `receive [patt -> body]* end` (see the code for concrete examples). The receive operation is a pattern-matching operation performed on the structure of messages that are collected from a process' mailbox. Each pattern is applied in turn to all received messages. When the first message matches, the computation in the `body` is executed. For example if one expects a temperature message with two integers, one can write the pattern {`temperature,C,K`}.

Typically, each receive clause is contained within a loop (implemented using recursion) so that the actor can continually process messages. It is important to ensure that this loop is tail recursive to ensure that memory consumption is constant.

Modify function `start` in `esOne.erl` so that it receives for a message of form {`reply, N`}, once that message arrives it prints out feedback.

Execute `srvEsOne:start().`, in addition to the previous messages you should see a message your feedback and another message from the `Server`, that will terminate instead of disconnecting.

3 New exercise

This exercise will see you writing a process that interacts with a local server (another process). The local server has been written for you, you can find it in directory `Es2` in file `srv.erl`. Execute `srv:startD().` to run the server, it will be registered locally via name `dummy`: this is the name your program can use to send messages to the server.

Your process will be a modification of the file `dummy.erl`. Based on that file, you will create a process (from now on called *the killer*) that will interact locally with the aforementioned server. To run this exercise, you will have to execute its main function: `dummy:test().`.

Below is an explanation of the exercise; you are advised to implement each functionality one at a time and check its correctness before moving to the next point. The explanation is followed by a table that recaps the structure of all messages that can be exchanged between the dummy and the server and by a UML sequence diagram that depicts the interaction between server and client over time.

Initialisation. The killer performs the following steps before starting the main receive loop:

- *Register* itself locally with via a name of your choice.
- *Send* its PID and its registered name to the server as a `register` message.
- *Receive* two numbers from the server, an attack and a defence value, as a `vals` message.

Then the killer starts its receive loop, which identifies its lifetime.

Debug. The server may communicate any existing problem to the killer by sending it a `comm` message. Any such message should be printed to screen and added to a local log. For printing and adding to a local log, use function `logSer(What,Arg,Name)` in `srv.erl`. `Name` is the file to write, which will be found in the folder `logs` (if this function fails, create that folder). `What` is a string with possible ∼`p` or ∼`w`, `Arg` is a list of arguments that replace all ∼`p` or ∼`w` of the `What` string. Calling to this function will also print to screen `What` and `Arg`.

Process life. The killer has 20 HP. This is part of the state that needs to be remembered inside the killer's *receive loop*. During its lifetime the killer can perform different actions:

- attack: send an `atk` message to the server containing its PID, its registered name and its attack value.
- defend: receive `atk` messages from the server. In this case it will calculate (without cheating) the new amount of HP (HP $-$ attack value communicated $+$ the process' defence value). When the killer's HP reach 0 or less, it dies, sending itself a `die` message.
- die: when a killer dies it sends its PID and its name to the server on the atom `killed` and then it stops executing. Notice that the server may also tell the killer to die by sending a message `die`.

To implement the killer's lifecycle in a truly concurrent fashion, it must be allowed to attack and defend at the same time. This is done by using an external process that tells the attacker when to attack the server. This process has been written for you, it is called `striker`, and it is available at line 108 in the `srv.erl` module. Inspect the code to understand how it works, intuitively the `striker` will send a message `strike` to the killer *every 200 msec*. When the `strike` message is received, the killer must attack the server.

The messages to be sent and received have the following structures:

Message structure	**Example**
`die`	`die`
`{register, Pid, Name}`	`{register, 3012, Gothmog}`
`{vals, ValAttack, ValDefense}`	`{vals, 10, 5}`
`{atk, Pid, Name, WeaponPower}`	`{atk, 3012, Gothmog, 10}`
`{killed, Pid, Name}`	`{killed, 6798, Fëanor}`
`{comm, String, Arg}`	`{comm, "Some error your process ~p does.", 3012}`
`strike`	`strike`

The UML sequence diagram that models the interaction client-server is the following: