Introductory Laboration: Erlang

Jari Stenman

January 27, 2012

Contents

1 Information

This laboration assumes that you have read the required material mentioned on the website. You are required to work in pairs. The deadline is *2012-02-06 Mon 23:59*.

2 Introduction

The fictitious bookstore Bookus, selling cheap science fiction books, is setting up operations in Sweden. They need an Erlang programmer to implement the webstore, and management has decided to hire you. You will need to write a simple server that handles book purchases.

3 Task

Create a file called **bookstore.erl** that contains the following:

```
%% Distributed Systems 2012: Erlang Lab
%% Student 1:
%% Student 2:

-module(bookstore).

-export([server/1, client/3, start/0]).

server(State) -> server(State).

client(Server, X, Wait) ->
    timer:sleep(Wait),
    case X of
        1 -> Server ! {buy, "Dune", self()};
        2 -> Server ! {buy, "Excession", self()};
        3 -> Server ! {buy, "The Forever War", self()}
    end.

start() ->
    Server = spawn(bookstore, server, [[]]),
 %% Server ! {add, Name, Author, Price, Quantity}
    Server ! {add, "Dune", "Frank Herbert", 150, 10},
    Server ! {add, "Excession", "Iain M. Banks", 300, 2},
    Server ! {add, "The Forever War", "Joe Haldeman", 170, 3},
    register(bookus, Server).

spawnclients(Server, N) ->
    Vals = lists:map(fun(_) -> random:uniform(3) end,
                     lists:seq(1,N)),
    lists:map(fun(X) -> spawn(bookstore, client, [Server, X,
        random:uniform(5000)]) end,
              Vals).
```

Here, the start function spawns a server process, sets up the initial inventory and registers the server PID. Your job is to implement the server function. When testing, run `Bookus = whereis(bookus).` so that you have a name for the server. You can then send messages by writing `Bookus !Message.` For your convenience, there is a function called spawnclients which spawns a given number of clients that buy books randomly. You can run it by writing e.g. `bookstore:spawnclients(Bookus, 10).`

The following functionality is required. You are of course free to add behaviour that does not conflict with the requirements.

3.1 Adding a book

The people at Bookus regularly add new books to the bookstore. This is done by sending the message `{add, Book, Author, Price, Quantity}` to the server. Quantity denotes how many copies are available.

For example, `{add, "Dune", "Frank Herbert", 150, 10}` would add 10 copies of the science fiction classic *Dune* by *Frank Herbert* to the bookstore, at a price of 150 kr per book.

Let's try to implement this. We remember that Erlang has a receive expression for receiving messages. To receive an add message, we pattern match against `{add, Book, Author, Price, Quantity}` . We modify the server function to receive these messages:

```
server(State) ->
    receive
        {add, Book, Author, Price, Qty} -> %% deal with add
            request
        _ -> %% anything else
    end.
```

Now the question is what we should do when we receive an add message. It is clear that we should somehow modify the state. But what does the state look like? The information we need to store in the server is basically a list of books together with their authors, prices and quantities. So let us assume that the state is a list containing tuples of the form `{Book, Author, Price, Quantity}` . Now, how should we modify this list? If the book in question already exists, we are not going to add it to the list. To perform this check, we can use the function `lists:keysearch` . According to the manual, `keysearch(Key, N, TupleList)` "searches the list of tuples `TupleList` for a tuple whose `N` th element compares equal to `Key` . Returns `{value, Tuple}` if such a tuple is found, otherwise `false` ". We are interested in the book name, which is the first element of each tuple. This yields the following check.

```
server(State) ->
    receive
        {add, Book, Author, Price, Qty} ->
            case lists:keysearch(Book, 1, State) of
                false -> %% book not in list: do something
                _ -> %% book already in list
            end
    end.
```

If the book is not in the list, `lists:keyfind` will return `false` . In this case, the new state is `[{Name, Author, Price, Quantity}| State]` . Otherwise, the new

state is the same as the old one. We modify the function accordingly:

```
server(State) ->
    receive
        {add, Book, Author, Price, Qty} ->
            case lists:keysearch(Book, 1, State) of
                false ->
                    NewState = [{Book, Author, Price, Qty}|State],
                    io:format("Bookus ~w: Added ~s! New inventory:
                         ~p~n", [self(), Book, NewState]),
                    server([{Book, Author, Price, Qty}|State]);
                _ -> server(State)
            end
    end.
```

We are done with this part. Start the server and add some books to see that it works. That was easy, wasn't it? You should now be able to implement the remaining parts.

3.2 Buying a book

Clients should be able to buy books. To buy a book, one sends the message `{buy, Book, Sender}` , where Sender is the PID of the client.

When the server receives `{buy, Book, Sender}` , it should print "Buy request received from " and the process ID of the sender. Additionally, one of three things should happen:

- If the book does not exist, the server should state that.

```
4> Bookus = whereis(bookus).
<0.37.0>
5> Bookus ! {buy, "Hyperion", self()}.
Bookus <0.37.0>: Buy request received from <0.49.0>
Bookus <0.37.0>: Sorry <0.49.0>, we do not have Hyperion
{buy,"Hyperion",<0.49.0>}
6>
```

- If the book exists, but is sold out, the server should state that.

```
6> Bookus ! {buy, "The Forever War", self()}.
Bookus <0.37.0>: Buy request received from <0.49.0>
Bookus <0.37.0>: Sorry <0.49.0>, The Forever War is sold out
```

```
{buy,"The Forever War",<0.49.0>}
7>
```

- If the book exists, and there are copies left, the server should print the PID of the buyer and decrement the number of available copies of that particular book. The server should then print its inventory.

```
7> Bookus ! {buy, "Dune", self()}.
Bookus <0.37.0>: Buy request received from <0.49.0>
{buy,"Dune",<0.49.0>}
Bookus <0.37.0>: Client <0.49.0> bought a copy of Dune
Bookus <0.37.0>: New inventory: [{"The Forever War","Joe Haldeman"
    ,170,0},
                                 {"Excession","Iain M. Banks"
                                     ,300,0},
                                 {"Dune","Frank Herbert",150,7}]
8>
```

Hint: `lists:keyreplace`

3.3 Closing the bookstore

It should be possible to close the bookshop (i.e. end the server process) by sending the message `close` .

When the server receives this message, it should print that it is closed and terminate.

```
8> Bookus ! close.
Bookus <0.37.0>: Closed!
close
9>
```

3.4 Other messages

Other messages are obviously junk and should be thrown away.

```
15> Bookus ! "Hello Bookus. This is Skatteverket. Where is the tax
     money?".
Bookus <0.57.0>: Junk received
"Hello Bookus. This is Skatteverket. Where is the tax money?"
16>
```

4 Handing in

There is a file area called Erlang Lab in Studentportalen, under this course. In this file area, you fill find a folder with your name. Upload your modified version of **bookstore.erl** there. Since you work in pairs, only one of you needs to submit. **Before you hand in**, please fill in your and your lab partner's name in the topmost comment section. The deadline is *2012-02-06 Mon 23:59*.