A very simple client server in Erlang

February 17, 2009

1 Introduction

By now you should have read the first 28 pages of “Getting Started with Erlang” which is linked from the instruction homepage or can be found here:

http://www.erlang.org/download/getting_started-5.4.pdf

You shall now develop a very simple client server system in Erlang.

2 A simple server

Remember that Erlang is a functional language, which means that there are no variables but only values. A server generally has some state. Thus we have to use patterns similar to the following:

server(State) ->
receive
  some_request ->
  %%..... do some stuff
  %% ...... compute the new state NewState
  server(NewState)
end.

You will extend the following very simple server. Your server should have one piece of state, a value that records how many times it has been called.

server(State) ->
receive
  {request,Return_PID} ->
  io:format("SERVER ~w: Client request received from ~w\n", [self(), Return_PID]) ,
  NewState = State + 1,
  Return_PID ! {hit_count,NewState},
The client is simpler, it takes as a parameter the server PID, sends the request and prints out the value.

```erlang
client(Server_Address) ->
    Server_Address ! {request, self()},
    receive
        {hit_count, Number} ->
            io:format("CLIENT ~w: Hit count was ~w~n", [self(), Number])
    end.
```

Now put everything in one file named `simple.erl` and try it!!! (Actually type in the file yourself, you'll learn a lot just by typing the examples in).

```erlang
-module(simple).
-export([server/1, client/1, start/0]).

server(State) ->
    receive
        {request, Return_PID} ->
            io:format("SERVER ~w: Client request received from ~w~n", [self(), Return_PID]),
            NewState = State + 1,
            Return_PID ! {hit_count, NewState},
            server(NewState)
    end.

client(Server_Address) ->
    Server_Address ! {request, self()},
    receive
        {hit_count, Number} ->
            io:format("CLIENT ~w: Hit count was ~w~n", [self(), Number])
    end.

start() ->
    Server_PID = spawn(simple, server, [0]),
    spawn(simple, client, [Server_PID]).
```

To compile and run from the command line, open up the Erlang shell:

```
$> erl
Erlang (BEAM) emulator version 5.4.10 [source] [hipe]
```
3 Many clients

The previous example is a bit boring with only one client. Add the following function to spawn multiple clients:

```erlang
spawn_n(N, Server_PID) ->
    if
    N>0 ->
        spawn(simple,client,[Server_PID]),
        %% Use a random sleep in miliseconds to simulate the
        %% client traffic pattern.
        timer:sleep(random:uniform(100)),
        spawn_n(N-1,Server_PID);
    N == 0 ->
        io:format("Last client spawned."n")
    end.
```

4 Questions

You should hand in answers to the questions below in Ping Pong.

**Question 1:** In the simple server

```erlang
server(State) ->
    receive
    {request,Return_PID} ->
        io:format("SERVER ~w: Client request received from ~w~n", [
            self(), Return_PID]),
        NewState = State + 1,
        Return_PID ! {hit_count,NewState},
        server(NewState)
    end.
```

What does the line

```
Return_PID ! {hit_count,NewState},
```

do?
**Question 2:** In the simple client

```
client(Server_Address) ->
    Server_Address ! {request, self()},
    receive
        {hit_count, Number} ->
            io:format("CLIENT ~w: Hit count was ~w\n", [self(), Number])
    end.
```

What does `self()` do and why is it used in this example?

**Question 3:** In the simple `start()` function what does

```
spawn(simple,client,[Server_PID]).
```

do?

## 5 Add functionality to client and server

Start with the `simple.erl` file and add functionality by following the steps below. Hand in the finished `simple.erl` file in Ping Pong.

**Step 1:** Modify `start` to take a single parameter which is the number of clients that have to be spawned. Don’t forget to modify the export directives correctly.

**Step 2:** To make it easy to compile and test your program you should use a Makefile. Create a file named `Makefile` with the following content (or download the Makefile from the instruction page):

```
simple.beam: simple.erl
    erlc -W $<

test: simple.beam
    erl -noshell -s simple main 300
```

To use the Makefile you must also add the following function to `simple.erl`

```
main([Arg]) ->
    N = list_to_integer(atom_to_list(Arg)),
    start(N),
    init:stop().
```

Don’t forget to modify the export directives so that `main` is exported.

Now you could simple type `make test` at the command line to compile (if needed) and run your program.
Step 3: The owner of the server should have the right to query the count without incrementing it. Modify the server by adding code for the message:

\{server_owner,Server_PID\}

In the following place

server(State) ->
receive
  \{request,Return_PID\} ->
    io:format("SERVER \w: Client request received from \w\n", [self(), Return_PID]),
    NewState = State + 1,
    Return_PID ! \{hit_count,NewState\},
    server(NewState);
  \{server_owner,Owner_PID\} ->
    .......
end.

Step 4: Modify start to have a single owner process that constantly queries the server for the state. You should fill in the ....s and decide on what other messages are passed between the server and the owner. You should print out the value received from the server using io:format.

owner(Server_PID) ->
  %% Use a random sleep in ms to simulate the owner traffic pattern.
  timer:sleep(random:uniform(100)),
  Server_PID! .... ,
  receive
    \{ .... \} ->
    .... ,
    io:format(......) ,
    owner(Server_PID)
end.

Step 5: You are to give the owner the power to reset the server counter to 0. You should modify the server as follows:

server(State) ->
receive
  \{request,Return_PID\} ->
    io:format("SERVER \w: Client request received from \w\n", [self(), Return_PID]),
    NewState = State + 1,
    Return_PID ! \{hit_count,NewState\},
    server(NewState);
  reset ->
io:format( .... ) ,
server( .... )
end.

again print out some message to say that you’ve been reset on the screen, but
also you’ll need to work out what value pass when you do the recursive call to
server.

**Step 6**: Now modify the *owner* process to reset the server counter if it exceeds
5. Look at how the Erlang *if* statement works or how to use *guards* in Erlang
*receive* statements.

## 6 Test run

A test run of the finished program should look something similar to this:

```
make test
erlc -W simple.erl
erl -noshell -s simple main 20
SERVER <0.27.0>: Client request received from <0.29.0>
CLIENT <0.29.0>: Hit count was 1
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.30.0>
OWNER <0.28.0>: Hit count is 1
CLIENT <0.30.0>: Hit count was 2
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.31.0>
OWNER <0.28.0>: Hit count is 2
CLIENT <0.31.0>: Hit count was 3
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.32.0>
OWNER <0.28.0>: Hit count is 3
CLIENT <0.32.0>: Hit count was 4
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.33.0>
OWNER <0.28.0>: Hit count is 4
CLIENT <0.33.0>: Hit count was 5
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.34.0>
OWNER <0.28.0>: Hit count is 5
CLIENT <0.34.0>: Hit count was 6
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.35.0>
OWNER <0.28.0>: Hit count is 6, send reset message....
CLIENT <0.35.0>: Hit count was 7
SERVER <0.27.0>: Owner reset message received
SERVER <0.27.0>: Owner query received from <0.28.0>
```
SERVER <0.27.0>: Client request received from <0.36.0>
OWNER <0.28.0>: Hit count is 0
CLIENT <0.36.0>: Hit count was 1
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.37.0>
OWNER <0.28.0>: Hit count is 1
CLIENT <0.37.0>: Hit count was 2
SERVER <0.27.0>: Owner query received from <0.28.0>
SERVER <0.27.0>: Client request received from <0.38.0>

Note: The server hit count may not be immediately reset once the owner process discovers the count to exceed the limit (5). This is due to client requests being received by the server before the owner reset message is received by the server.