

# Seminar Report: Chatty

Peter Grman, Leonardo Tonetto

September 30, 2012

## 1 Introduction

In this double seminar session we had a quick introduction to Erlang, with the basic concepts of the language. Then we were presented with a task where we should implement a distributed system that would allow client processes to communicate with each other using the messaging infrastructure provided by Erlang. We did two different implementations:

- One with a single centralized server, that would be responsible to accept clients to join and leave the chat, as well as broadcast the messages sent from any client to all the others connected at that time;
- Another one decentralized, with a possibility of existing multiple servers, that would hold a list of the available servers, and each new client could still have the possibility to connect to any available server. Any message sent from a client would be first broadcast to the clients of the same server, and then to the other servers that would send to their own clients.

Most parts of the code were already done, with the purpose of introducing the language, but still with gaps that should be completed to make us think about what should be used so that the whole code could work.

## 2 System overview

We'll define two different implementations of the server code (**server.erl** and **server2.erl**), with the first being the centralized one, and the second the distributed version.

### 2.1 **server.erl**

This source code had only three functions: **start()**, **process\_request()** and **broadcast()**.

The `start()` function is responsible for creating a child thread that executes `process_request()` and call `register()` to register the thread's PID to a name, in this case `myserver`.

The `process_request()` function is responsible for treating the different messages that our server can receive, and perform the appropriate action for each message. This implementation was capable of working with the following messages defined in [1]:

- `{client_join_req, Name, From}`
- `{client_leave_req, Name, From}`
- `{send, Name, Text}`

Finally `process_request()` is implemented like the following:

```
process_request(Clients) ->
    receive
        {client_join_req, Name, From} ->
            NewClients = [From | Clients],
            broadcast(NewClients, {join, Name}),
            process_request(NewClients);
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(From, Clients),
            broadcast(Clients, {leave, Name}),
            process_request(NewClients);
        {send, Name, Text} ->
            broadcast(Clients, {message, Name, Text}),
            process_request(Clients)
    end.
```

And `broadcast()` broadcasts any message to all the clients connected to this server. For this, it makes use of `lists:map()` and `fun()` functions, that perform a set of actions to all the members of a given list.

## 2.2 server2.erl

In this more robust version of `server.erl` there are two implementations of the `start()` function. One of them is to be used only by the first server to get on-line and has no arguments. The second implementation uses one argument which is a tuple that defines one of the existing servers to which the connection should go through in order to succeed. In this multi-step connection, the server that wants to connect will issue a message `{server_join_req, From}` that the existing server will use to update the list of all servers and broadcast it (in the format of `{update_servers, NewServers}` message) to all the servers on-line at that moment, including the new one. To process the messages, we use a different implementation of `process_requests()`:

```

process_requests(Clients, Servers) ->
    receive
        %% Messages between client and server
        {client_join_req, Name, From} ->
            NewClients = [From|Clients],
            broadcast(Servers, {join, Name}),
            process_requests(NewClients, Servers);
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(From, Clients),
            broadcast(Servers, {leave, Name}),
            process_requests(NewClients, Servers);
        {send, Name, Text} ->
            broadcast(Servers, {message, Name, Text}),
            process_requests(Clients, Servers);
        %% Messages between servers
        {server_join_req, From} ->
            NewServers = [From|Servers],
            broadcast(NewServers, {update_servers, NewServers}),
            process_requests(Clients, NewServers);
        {update_servers, NewServers} ->
            io:format("[SERVER_~w~n]", [NewServers]),
            process_requests(Clients, NewServers);
        {disconnect} ->
            NewServers = lists:delete(self(), Servers),
            broadcast(NewServers, {update_servers, NewServers}),
            unregister(myserver);
        %% Whatever other message is relayed to its clients
        RelayMessage ->
            broadcast(Clients, RelayMessage),
            process_requests(Clients, Servers)
    end.

```

So, as already explained, this version of the code send the messages to all the clients connected to that server, and broadcast it to the other servers that will replicate it to its own clients.

## 2.3 client.erl

As the client code, **client.erl** implements a `start()` and a `init_client()` functions that bind the client to a server (sending a message `{client_join_req, MyName, self()}`) and also calling two functions:

- `process_commands()` from the main process;
- `process_requests()` from a child thread.

The `process_requests()` function, similar to the server implementation, processes the messages received by the server, and `process_commands()` creates a terminal that accepts user input, the chat messages for instance,

and send them to the server to be broadcast to the other users connected. In case the user enters **exit** on the chat console, the client is disconnected from the server, and a message warning the other clients about the change is sent.

The client implementation of `process_requests()` is like the following:

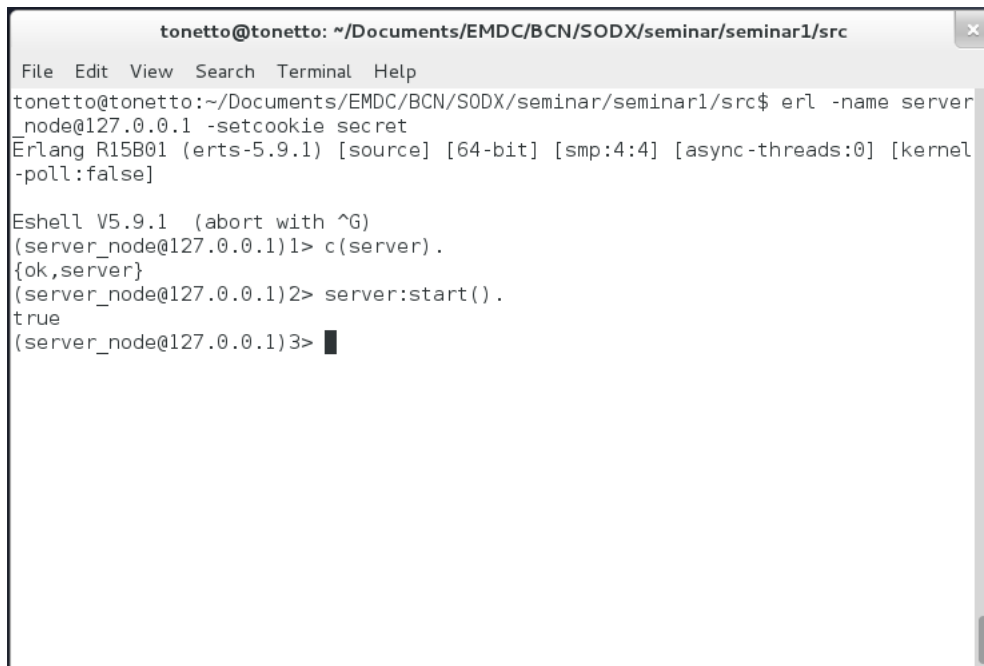
```
process_request() ->
  receive
    {join, Name} ->
      io:format("[JOIN] _~s_ joined _the_ chat~n", [Name]),
      process_request();
    {leave, Name} ->
      io:format("[LEAVE] _~s_ leaved _the_ chat~n", [Name]),
      process_request();
    {message, Name, Text} ->
      io:format("[~s] _~s_", [Name, Text]),
      process_request()
  end.
```

## 3 Evaluation

We'll make this evaluation in two parts, considering the simpler version and the more robust.

### 3.1 Simple version

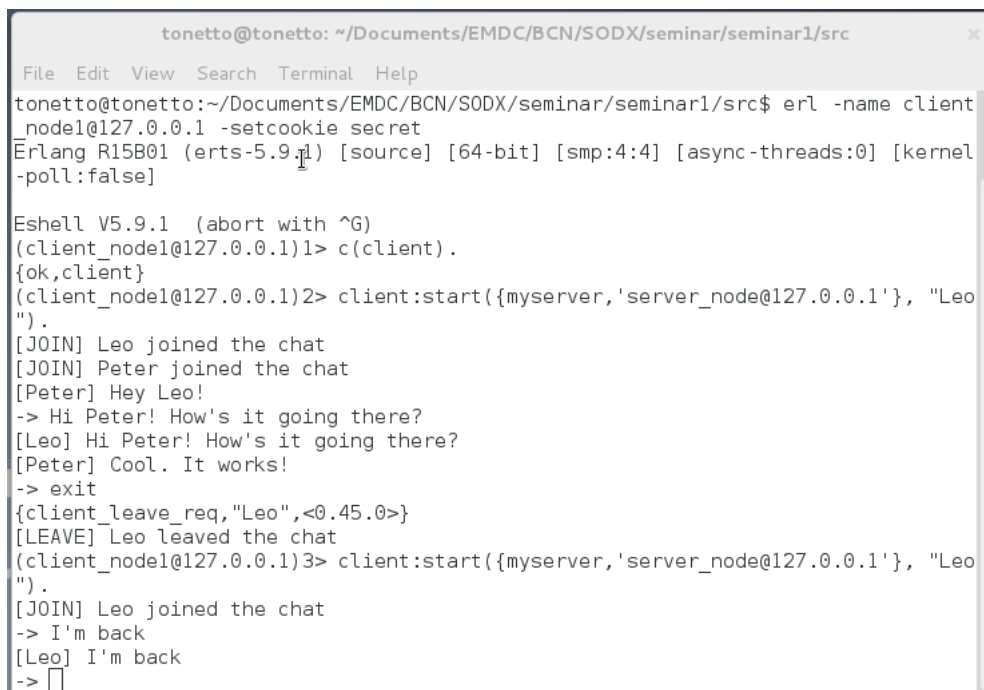
Following the design of the simple version, it does quite well what it's supposed to do. The server runs up on-line, allowing clients to connect to it, and broadcasts messages between them as expected.

A screenshot of a terminal window titled "tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar1/src". The window contains the following text:

```
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar1/src$ erl -name server_node@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(server_node@127.0.0.1)1> c(server).
{ok,server}
(server_node@127.0.0.1)2> server:start().
true
(server_node@127.0.0.1)3> █
```

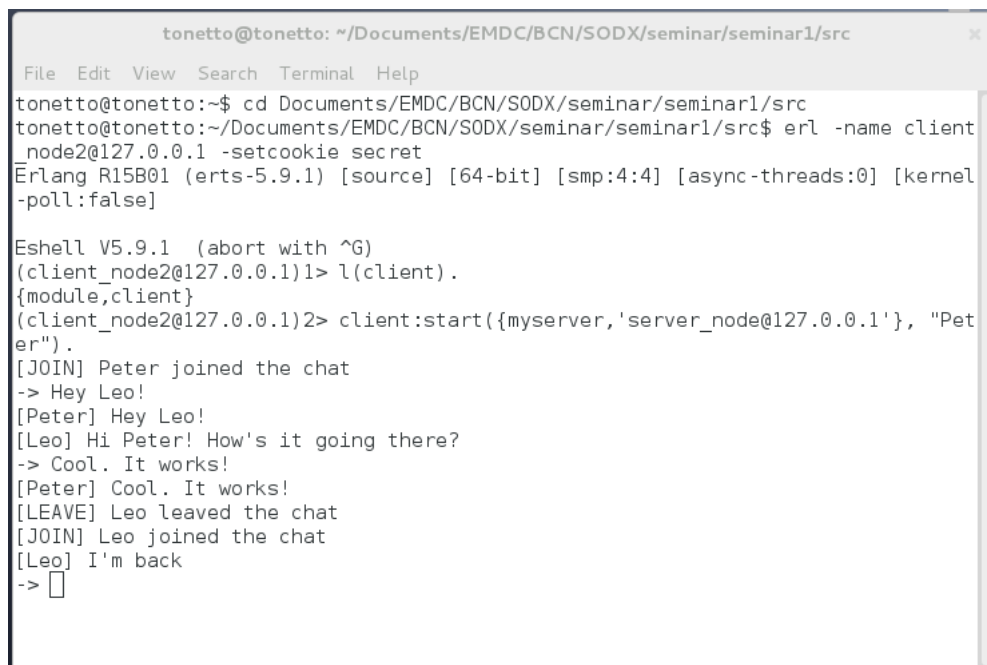
Figure 1: serve\_node window

A screenshot of a terminal window titled "tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar1/src". The window contains the following text:

```
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar1/src$ erl -name client_node@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(client_node@127.0.0.1)1> c(client).
{ok,client}
(client_node@127.0.0.1)2> client:start({myserver,'server_node@127.0.0.1'}, "Leo").
[JOIN] Leo joined the chat
[JOIN] Peter joined the chat
[Peter] Hey Leo!
-> Hi Peter! How's it going there?
[Leo] Hi Peter! How's it going there?
[Peter] Cool. It works!
-> exit
{client_leave_req,"Leo",<0.45.0>}
[LEAVE] Leo leaved the chat
(client_node@127.0.0.1)3> client:start({myserver,'server_node@127.0.0.1'}, "Leo").
[JOIN] Leo joined the chat
-> I'm back
[Leo] I'm back
-> █
```

Figure 2: client\_node1 window



```
tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar1/src
File Edit View Search Terminal Help
tonetto@tonetto:~$ cd Documents/EMDC/BCN/SODX/seminar/seminar1/src
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar1/src$ erl -name client_node2@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(client_node2@127.0.0.1)1> l(client).
{module,client}
(client_node2@127.0.0.1)2> client:start({myserver,'server_node@127.0.0.1'}, "Peter").
[JOIN] Peter joined the chat
-> Hey Leo!
[Peter] Hey Leo!
[Leo] Hi Peter! How's it going there?
-> Cool. It works!
[Peter] Cool. It works!
[LEAVE] Leo leaved the chat
[JOIN] Leo joined the chat
[Leo] I'm back
-> □
```

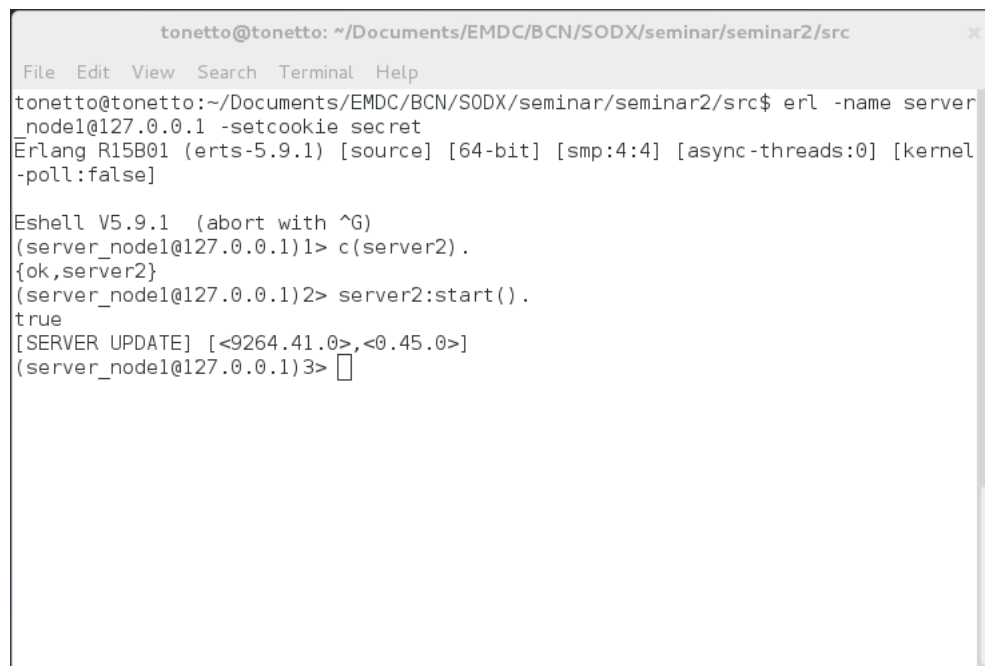
Figure 3: client\_node2 window

If for some reason the server disconnects (e.g. you kill the erlang instance that is running the server) the clients stay in the connected state, but of course without any connection. Trying to run the server up again, the clients won't be able to reach back the sever.

In a test running 10 clients connected to the server, all the clients were able to send and receive the messages without problems.

### 3.2 Robust version

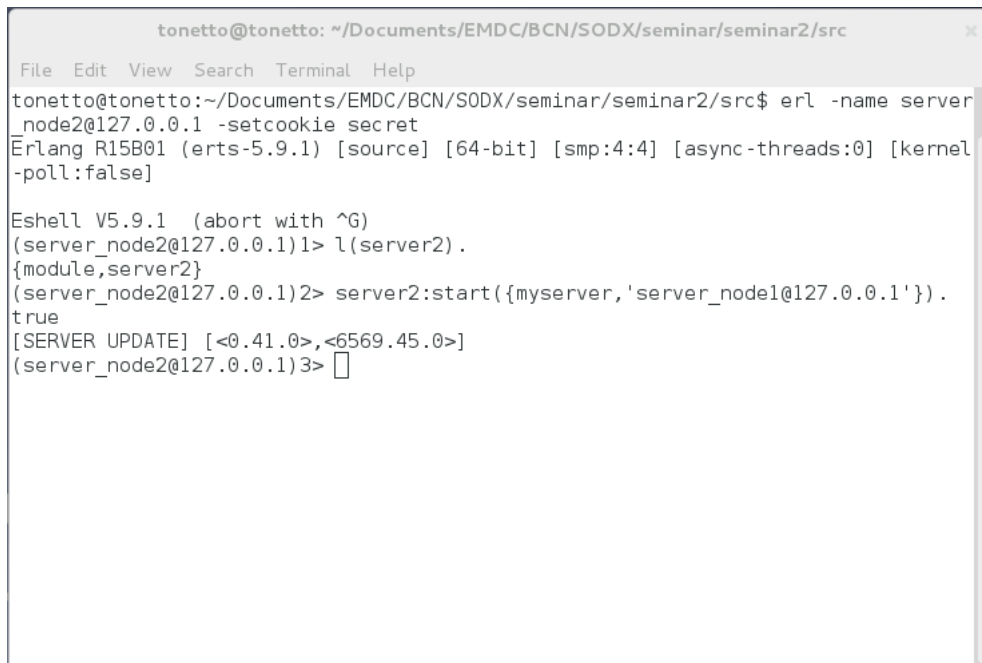
As the name suggest, this is a more robust version of the chat system implementation. Using a distributed set of servers, clients can connect to whatever server they desire, the messages are replicated to all clients, no matter to which server they are connected to and there are also interactions between the servers to handle the connections and disconnections.



```
tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name server_node1@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(server_node1@127.0.0.1)1> c(server2).
{ok,server2}
(server_node1@127.0.0.1)2> server2:start().
true
[SERVER UPDATE] [<9264.41.0>,<0.45.0>]
(server_node1@127.0.0.1)3> █
```

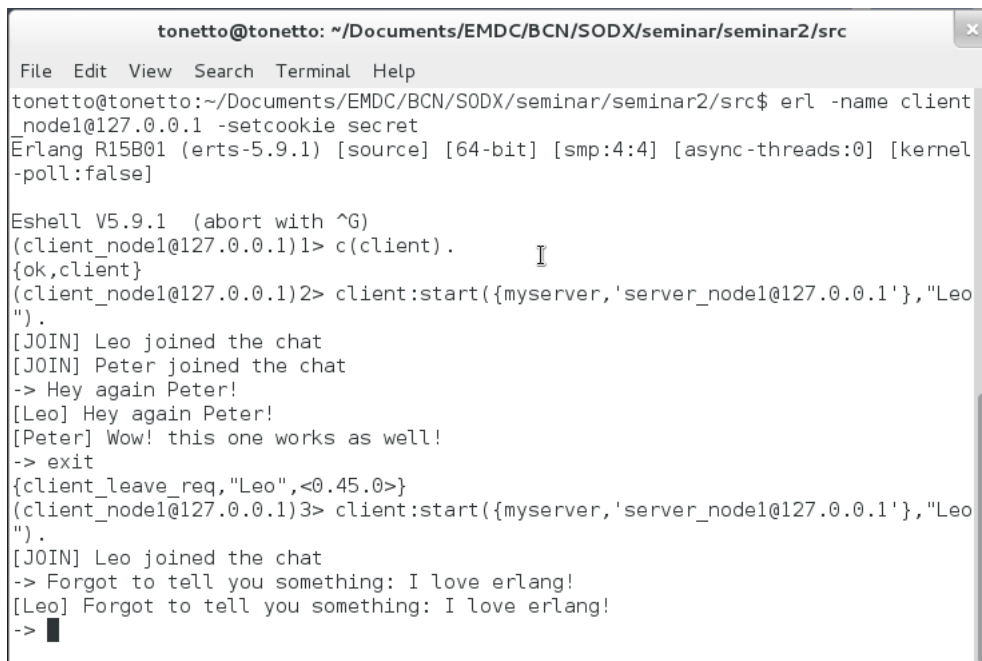
Figure 4: serve\_node1 window



```
tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name server_node2@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(server_node2@127.0.0.1)1> l(server2).
{module,server2}
(server_node2@127.0.0.1)2> server2:start({myserver,'server_node1@127.0.0.1'}).
true
[SERVER UPDATE] [<0.41.0>,<6569.45.0>]
(server_node2@127.0.0.1)3> 
```

Figure 5: serve\_node2 window

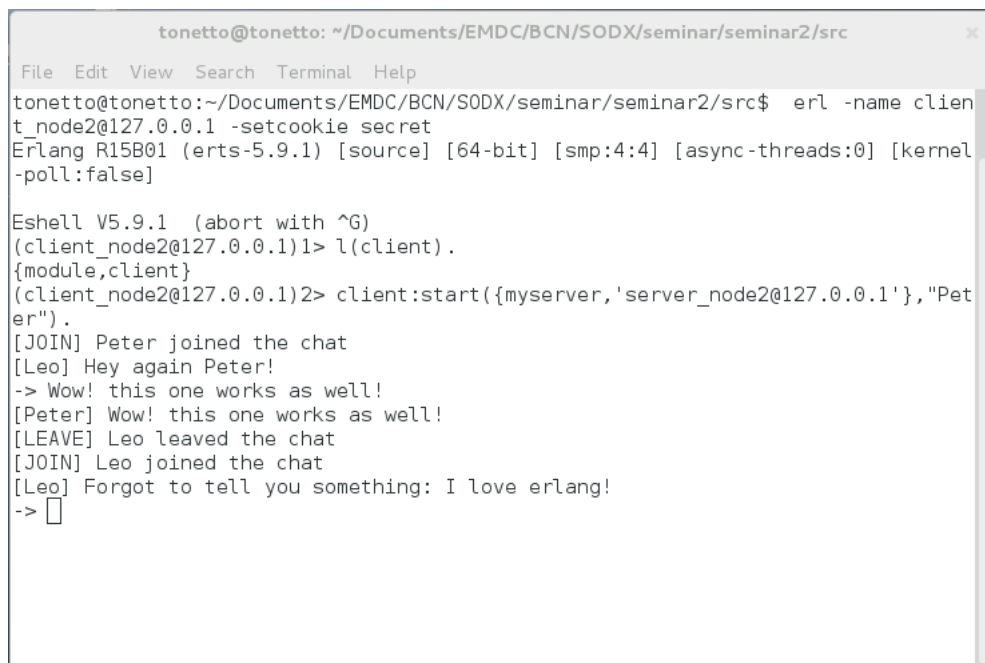


```
tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name client_node1@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(client_node1@127.0.0.1)1> c(client).
{ok,client}
(client_node1@127.0.0.1)2> client:start({myserver,'server_node1@127.0.0.1'},"Leo").
[JOIN] Leo joined the chat
[JOIN] Peter joined the chat
-> Hey again Peter!
[Leo] Hey again Peter!
[Peter] Wow! this one works as well!
-> exit
{client_leave_req,"Leo",<0.45.0>}
(client_node1@127.0.0.1)3> client:start({myserver,'server_node1@127.0.0.1'},"Leo").
[JOIN] Leo joined the chat
-> Forgot to tell you something: I love erlang!
[Leo] Forgot to tell you something: I love erlang!
-> 
```

Figure 6: client\_node1 window





```
tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name client_node2@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(client_node2@127.0.0.1)1> l(client).
{module,client}
(client_node2@127.0.0.1)2> client:start({myserver,'server_node2@127.0.0.1'},"Peter").
[JOIN] Peter joined the chat
[Leo] Hey again Peter!
-> Wow! this one works as well!
[Peter] Wow! this one works as well!
[LEAVE] Leo leaved the chat
[JOIN] Leo joined the chat
[Leo] Forgot to tell you something: I love erlang!
-> □
```

Figure 7: client\_node2 window

As we can see from the previous figures, the chat systems works well for a normal use of the system. We can also send a `{disconnect}` message from one server to the other, so that it leaves the network. But now, some problems rise:

- When the server is disconnected, its clients stay in a **connected** state, and even after it's back on-line, the former clients that belonged to it can reach the chat network, but they can't be reached by the other's messages. And it's because the clients still have the server's alias (i.e. `{myserver, 'server_node2@127.0.0.1'}`) but the server is not aware of them anymore (i.e. `Clients` list is empty). A weird inconsistency that could be solved with a message being sent from the server, to all it's clients when it's disconnected. The Figure 8 on page 11 shows this behavior.
- If an existing server instance is killed (e.g. `Ctrl+G` and `Q`, to quit, or the terminal window is closed) the server will this time not only leave it's clients unsupported, but also all the other servers won't be aware that it left the chat network. If this server connects back, it will broadcast it's PID to the other servers, creating another inconsistency. This can be fixed by trapping the operating system's exit signal and performing a proper disconnection procedure (i.e. tell the others servers that it's leaving, and again, tell it's clients that it's leaving as well). The Figure 9 on page 12 shows this behavior.

A proposed implementation for these fixes is included in the tarball file sent along with this report, in the *seminar2/src/bis* directory.

```

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name server_node1@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(server_node1@127.0.0.1)1> c(server2).
{ok,server2}
(server_node1@127.0.0.1)2> server2:start().
true
[SERVER UPDATE] [<9264.41.0>,<0.45.0>]
(server_node1@127.0.0.1)3> {myserver,'server_node2@127.0.0.1'} ! {disconnect}.
{disconnect}
[SERVER UPDATE] [<0.45.0>]
[SERVER UPDATE] [<9264.56.0>,<0.45.0>]
(server_node1@127.0.0.1)4> []

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name server_node2@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(server_node2@127.0.0.1)1> l(server2).
{module,server2}
(server_node2@127.0.0.1)2> server2:start({myserver,'server_node1@127.0.0.1'}).
true
[SERVER UPDATE] [<0.41.0>,<6569.45.0>]
(server_node2@127.0.0.1)3> server2:start({myserver,'server_node1@127.0.0.1'}).
true
[SERVER UPDATE] [<0.56.0>,<6569.45.0>]
(server_node2@127.0.0.1)4> []

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
Eshell V5.9.1 (abort with ^G)
(client_node1@127.0.0.1)1> c(client).
{ok,client}
(client_node1@127.0.0.1)2> client:start({myserver,'server_node1@127.0.0.1'},"Leo").
[JOIN] Leo joined the chat
[JOIN] Peter joined the chat
-> Hey again Peter!
[Leo] Hey again Peter!
[Peter] Wow! this one works as well!
-> exit
(client_leave_req,"Leo",<0.45.0>)
(client_node1@127.0.0.1)3> client:start({myserver,'server_node1@127.0.0.1'},"Leo").
[JOIN] Leo joined the chat
-> Forgot to tell you something: I love erlang!
[Leo] Forgot to tell you something: I love erlang!
-> Hola!
[Leo] Hola!
-> Hola(2)!
[Leo] Hola(2)!
[Peter] Wow, am I alone here?
-> []

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name client_node2@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel-poll:false]

Eshell V5.9.1 (abort with ^G)
(client_node2@127.0.0.1)1> l(client).
{module,client}
(client_node2@127.0.0.1)2> client:start({myserver,'server_node2@127.0.0.1'},"Peter").
[JOIN] Peter joined the chat
[Leo] Hey again Peter!
-> Wow! this one works as well!
[Peter] Wow! this one works as well!
[LEAVE] Leo leaved the chat
[JOIN] Leo joined the chat
[Leo] Forgot to tell you something: I love erlang!
-> Anybody there?
-> Wow, am I alone here?
-> []

```

Figure 8: server2.erl disconnect issue

The figure displays four terminal windows arranged in a 2x2 grid, showing the execution of Erlang code and the resulting chat messages. The top-left window shows the execution of `server2:start()` and the top-right window shows the execution of `server2:start()` with a message being discarded. The bottom-left window shows the execution of `client:start()` and the bottom-right window shows the execution of `client:start()` with a message being discarded.

```

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name server
node1@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel
-poll:false]

Eshell V5.9.1 (abort with ^G)
(server_node1@127.0.0.1)1> l(server2).
{module,server2}
(server_node1@127.0.0.1)2> server2:start().
true
[SERVER UPDATE] [<6722.41.0>,<0.41.0>]
[SERVER UPDATE] [<6722.41.0>,<6722.41.0>,<0.41.0>]
(server_node1@127.0.0.1)3> █

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
{module,server2}
(server_node2@127.0.0.1)2> server2:start({myserver,'server_node1@127.0.0.1'}).
true
[SERVER UPDATE] [<0.41.0>,<6569.41.0>]
(server_node2@127.0.0.1)3>
User switch command
--> q
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name server
node2@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel
-poll:false]

Eshell V5.9.1 (abort with ^G)
(server_node2@127.0.0.1)1> l(server2).
{module,server2}
(server_node2@127.0.0.1)2> server2:start({myserver,'server_node1@127.0.0.1'}).
true
[SERVER UPDATE] [<0.41.0>,<0.41.0>,<6568.41.0>]
(server_node2@127.0.0.1)3>
=ERROR REPORT==== 30-Sep-2012::16:47:47 ===
Discarding message {message,'Peter','we have it back...\n'} from <0.41.0> to <0.
41.0> in an old incarnation (2) of this node (3)

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name client
node1@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel
-poll:false]

Eshell V5.9.1 (abort with ^G)
(client_node1@127.0.0.1)1> l(client).
{module,client}
(client_node1@127.0.0.1)2> client:start({myserver,'server_node1@127.0.0.1'},"Leo
").
[JOIN] Leo joined the chat
[JOIN] Peter joined the chat
[Peter] hey!
-> ...
[Leo] ...
-> :-)
[Leo] :-)
[Peter] we have it back...
-> wow, finally!
[Leo] wow, finally!
-> █

tonetto@tonetto: ~/Documents/EMDC/BCN/SODX/seminar/seminar2/src
File Edit View Search Terminal Help
tonetto@tonetto:~/Documents/EMDC/BCN/SODX/seminar/seminar2/src$ erl -name client
node2@127.0.0.1 -setcookie secret
Erlang R15B01 (erts-5.9.1) [source] [64-bit] [smp:4:4] [async-threads:0] [kernel
-poll:false]

Eshell V5.9.1 (abort with ^G)
(client_node2@127.0.0.1)1> client:start({myserver,'server_node2@127.0.0.1'},"Pet
er").
[JOIN] Peter joined the chat
-> hey!
[Peter] hey!
[Leo] ...
-> still there?
-> we have it back...
-> █

```

Figure 9: buggy server2.erl

## 4 Open questions

### 4.1 Simple server

- *Does this solution scale when the number of users increase?*

Trying it with 10 users, the service was still working. But imagining it spread to different locations, by having only one centralized server this solution won't scale. Clients closer or with faster connections with the server will experience the service in a different way compared to others that would be far from it.

- *What happens if the server fails?*

The clients stay in a *connected* state, although it's not possible to reach the other clients in the chat network. If the server comes back, they'll still be isolated because the clients list on the server won't be populated.

- *Are the messages guaranteed to be delivered in the order they were issued (hint: think on several clients sending messages concurrently)?*

Since the messages are exchanged using TCP, they are guaranteed to be delivered. But since clients can connect from different sources, in different locations, messages may not arrive in the same order that they were really created. So in fact, each client may experience the connection in a different way, depending on how far they are from the server, and therefore from the other clients. The erlang interpreter will queue the messages that arrive to the server and it will be delivered on that order.

- *Does it matter the time in which users join and leave the chat? Does this influence the order of message delivery?*

Depending on the latency of the network where a group of clients are and the load of the server, it might happen that a client leaves the chat, but another client that was chatting with him may send a new message. The client that left will not receive the message, but the sender might see in his log that the message was sent and after that the other client left, creating an unreal experience for the sender.

### 4.2 Robust server

- *What are the advantages and disadvantages of this implementation regarding the previous one?*

Advantages could be that, considering a good connection between the servers, the latency between the messages sent and received by the clients can be enhanced/reduced, since each client now has the option to connect to different servers and chose the one that's best for him; considering a number

of clients connected to the system, now as they'll be spread across multiple servers, each server will be less loaded, and then more capable of handling the messages and connections.

Disadvantages could be that if the connection between the servers for some reason breaks, the clients will be islanded from the others connected to different servers; a server can be disconnected by receiving a `{disconnect}` message, creating another big possibility of inconsistency, since the clients will stay in the *connected* state.

- *What happens if a server fails?*

The other servers will not know it failed and the clients will stay in the *connected* state, resulting an inconsistency between the members (servers and clients) of the chat network.

- *Are the messages guaranteed to be delivered in order?*

Still the same problem of latency can exist, but with a good possible enhancement brought by a distributed servers sets.

- *Does it matter the order and time which users join and leave the chat?*

Same as 4.1, but this time a new server with a group of clients can connect to the network, as well as an existing server can disconnect or fail, so the user experience may vary depending on where it is placed and to which server it's connected to. Even though we have multiple servers, the list of existing clients is not shared between them.

- *And for the servers, what happens if there are concurrent requests to join or leave the system?*

Since the servers have a **Servers** populated list with the PID of all the other servers, but the connection is not centralized in one individual or they don't share memory, easily an inconsistency can be created when one server disconnects, issues a `{update_servers, NewServers}` message, and at the same time a new server joins, and a new `{update_servers, NewServers}` will be issued, from a different server. From this point, it's impossible to predict what will happen to each server connected to the network regarding which list will prevail.

## 5 Conclusions

We learned the basics of the Erlang language, implementing two versions of a chat server that make possible users (processes) to communicate, using the messaging system available in Erlang. The simple implementation was not scalable but was not open to problems that rose on the robust version. So we could also learn that the more features and complexity you add to a system, the more flaws and problems your code will be suitable.

## References

- [1] Jordi Guitart. Chatty: a simple chat service. September 2012. Adapted with permission from Johan Montelius (KTH) and Xavier Leon (UPC).