Chatty: a simple chat service

Jordi Guitart

Adapted with permission from Johan Montelius (KTH) & Xavier Leon (UPC)

February 6, 2025

Introduction

Your task will be to implement a distributed system that will allow you to chat among buddies. The purpose of this seminar is that you think about the main problems in distributed systems as well as learn a little bit of Erlang. We are going to implement two different versions of our chat system: i) a system composed by a single server to chat with your buddies; ii) a decentralized system with several servers which allows to clients connected to different servers chat with each other.

This document provides almost every piece of code. However, you need to fill in the gaps (...) to ensure you understand what the expected behavior of the system is.

This assignment includes a number of coding and experimental milestones. On accomplishing each of them, the team must show their outcome to the professor during the practical sessions, so that he can track the progress and evaluate the team accordingly. The team can validate their coding milestones beforehand through the review questionnaires at ATENEA. This assignment also includes a number of open questions. The team must answer each of them through the questionnaire at ATENEA before the deadline (one week after the last session of the seminar).

1 Chatting with buddies

The initial version of the chat will consist of two different types of processes: *clients* (friends to chat with) and a *server*. Clients will connect –and, of course, disconnect– to the server and the server is going to be responsible to maintain the list of clients attached and relay messages sent by a client to the rest of clients. As this design is quite simple from the distributed systems point of view, we are going make things more robust later on.

1.1 The server

We need to implement a server that keeps track of connected users and relays messages sent by one user to the rest of users. Thus, the message interface the server is going to handle is as follows. Note that we also included a catch-all clause in case some strange message is received.

- {`client_join_req, Name, From`}: a join request from a client containing its username (`Name`) and its process identifier (`From`) that the server should

use to contact him/her. The server needs to update the list of connected clients and send this new event (`join`) to the connected users.

- {`client_leave_req, Name, From`}: a leave request from a client containing its username (`Name`) and its process identifier (`From`). The server needs to remove the process from the list of connected clients and send this new event (`leave`) to the connected users.
- {`send, Name, Text`}: a request to send a message (`Text`) to the connected users from the client with username `Name`.
- `disconnect`: the server receiving the message disconnects.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open up a new file `server.erl` and declare the module `server`:

```
-module(server).
%% Exported Functions
-export([start/0]).

start() ->
    ServerPid = spawn(fun() -> process_requests([]) end),
    register(myserver, ...).  %% TODO: COMPLETE

process_requests(Clients) ->
    receive
        {client_join_req, Name, From} ->
            NewClients = [...|Clients],  %% TODO: COMPLETE
            broadcast(NewClients, {join, Name}),
            process_requests(...);  %% TODO: COMPLETE
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(..., ...),  %% TODO: COMPLETE
            broadcast(Clients, ...),  %% TODO: COMPLETE
            From ! exit,
            process_requests(...);  %% TODO: COMPLETE
        {send, Name, Text} ->
            broadcast(..., ...),  %% TODO: COMPLETE
            process_requests(Clients);
        disconnect ->
            unregister(myserver);
        Error ->
            io:format("Received unsupported message: ~w~n", [Error])
    end.

broadcast(PeerList, Message) ->
    Fun = fun(Peer) -> Peer ! Message end,
    lists:foreach(Fun, PeerList).
```

1.2 The client

So far, we have seen how the server is implemented. Now, we will specify how the client works. The client process will have two different tasks to perform. We will have a background task responsible for handling replies from the server and the main task will be responsible to read a message from the standard input to be sent to the rest of your buddies. The background task (the one handling server replies) should handle the following message interface. Again, we also included a catch-all clause in case some strange message is received.

- {`join, Name`}: the server informs that a new client is connected. We should write this information through the standard output.
- {`leave, Name`}: the server informs that a connected client is about to disconnect. We should write this information through the standard output.
- {`message, Name, Text`}: this is a message relayed by the server from one of the connected users (`Name`). We should print this message (`Text`) to the standard output so the end user can actually read it.
- `exit`: the client background task is terminated.

Notice that those message do not need to carry the process identifier of the referred client, as buddies do not contact directly with other buddies but with the server.

The main task should block waiting for the user to write down some text to send to the chatting room. If a user wants to leave, we need to write a *command* (`exit`) to actually leave the chat. Once the user writes this keyword, the main task will send a request to the server to leave the room.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open a new file `client.erl` and declare the module `client`.

```
-module(client).
%% Exported Functions
-export([start/2]).

start(ServerPid, MyName) ->
    ClientPid = spawn(fun() -> init_client(ServerPid, MyName) end),
    process_commands(ServerPid, MyName, ClientPid).

init_client(ServerPid, MyName) ->
    ServerPid ! {client_join_req, ..., ...},  %% TODO: COMPLETE
    process_requests().

%% This is the background task logic
process_requests() ->
    receive
        {join, Name} ->
            io:format("[JOIN] ~s joined the chat~n", [Name]),
            %% TODO: ADD SOME CODE
        {leave, Name} ->
```

```
            %% TODO: ADD SOME CODE
            process_requests();
        {message, Name, Text} ->
            io:format("[~s] ~s", [Name, Text]),
            %% TODO: ADD SOME CODE
        exit ->
            ok;
        Error ->
            io:format("Received unsupported message: ~w~n", [Error])
    end.

%% This is the main task logic
process_commands(ServerPid, MyName, ClientPid) ->
    %% Read from standard input and send to server
    Text = io:get_line("-> "),
    if
        Text  == "exit\n" ->
             ... ! {client_leave_req, ..., ...},  %% TODO: COMPLETE
            ok;
        true ->
            ServerPid ! {send, ..., ...},  %% TODO: COMPLETE
            process_commands(ServerPid, MyName, ClientPid)
    end.
```

1.3 Testing

Debugging Erlang code may be quite confusing sometimes because of the error messages. Once you get used to them, you will quickly understand what is wrong with your code. In the meanwhile, it is useful to put debugging information to know where your code is failing and the state of your variables.

If you are not using an Erlang IDE which automatically compiles your code, you will need to do it by hand –every time you modify the file– before calling your implemented procedures. A simple test can be done in a single computer. You will execute a server and a client in different terminals. To start a server write the following:

```
user@host:~/Chatty$ erl -name server_node@127.0.0.1 -setcookie secret

(server_node@127.0.0.1)1> c(server). %% Compile server module
(server_node@127.0.0.1)2> server:start().
true
```

To start a client open up a new terminal and write the following (note that clients reach the server by means of its registered local name (`myserver`)):

```
user@host:~/Chatty$ erl -name client_node@127.0.0.1 -setcookie secret

(client_node@127.0.0.1)1> c(client). %% Compile client module
(client_node@127.0.0.1)2> client:start({myserver, 'server_node@127.0.0.1'}, "John").
[JOIN] John joined the chat
```

And if you type a message, you should see something like...

```
-> hi!
[John] hi!
```

CODE REVIEW (2 points)
Client's functionality: Erlang file `client.erl`.
Server's functionality: Erlang file `server.erl`.

EXPERIMENTAL MILESTONES (1.75 points)	
MS1 (1 points)	Connect 3 clients to the chat and test that all of them receive all the messages.
MS2 (0.25 points)	Disconnect the server (by sending a `disconnect` message from the Erlang prompt) and see what happens.
MS3 (0.5 points)	Communicate 3 clients running on **different physical machines**[1].

OPEN QUESTIONS (1.5 points)

a) Does this implementation scale when the number of clients increases?

b) What happens to the chat system if the server disconnects?

c) Are the messages from a single client guaranteed to be delivered to any other client in the order they were issued? (hint: search for the 'order of message reception' in Erlang FAQ[2])

d) Are the messages sent concurrently by several clients guaranteed to be delivered to any other client in the order they were issued?

e) Given a client B who responds to a message from client A, is it possible that a third client C receives the response from B before receiving the original message from A?

f) If a client joins or leaves the chat while the server is broadcasting a message, will he/she receive that message?

2 Making it robust

The previous design has its disadvantages. Mainly, it is not robust to failures –i.e. if the server fails the whole system will become useless. A solution to this problem is to have more servers (replication). Thus, if a server fails, we will have other servers still running and sending messages. As usually, solving a problem is an open door for other interesting challenges.

[1] Replace the localhost IP by the domain name (or the IP address) of the corresponding machines. If you work in the lab, remember to specify the allowed port range when starting the Erlang runtime: `-kernel inet_dist_listen_min 1025 -kernel inet_dist_listen_max 2000`.

[2] http://erlang.org/faq/faq.html

We will have a set of replicated servers to which clients may connect indistinctly. This way, if a server fails, only those clients connected to the failing server will lose connectivity but the rest will remain connected. Of course, we could implement a solution in which clients automatically reconnect to another server when they detect a failure but we are going to keep things simple.

We will only need to change the server implementation to introduce this new functionality. The server needs to know the list of replicated servers. Besides, we need to handle the membership of the set of servers. Clients will connect to one of the servers and send messages to it. This server will forward the message to the set of servers which in turn will relay the message to its clients.

The new messages that the server needs to handle are as follows:

- {`server_join_req, From`}: a new server (`From`) is added to the set of replicated servers. We need to update the list of servers and notify the servers about this new node.
- {`update_servers, NewServers`}: a server gets informed when a server has joined/left the set.
- `RelayMessage`: if the message received does not match any of the previous clauses, the server relays the message to all of its clients (it may be either a message of type `join`, `leave`, or `message`).

Be aware that now, the messages previously implemented (`client_join_req`, `client_leave_req`, and `send`) **are not forwarded directly to the clients but to every member of the set of servers**. They in turn will relay those messages to their connected clients. In addition, when a server disconnects, the rest of servers are informed.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open a new file `server2.erl` and declare the module `server2`.

```
-module(server2).
%% Exported Functions
-export([start/0, start/1]).

start() ->
    ServerPid = spawn(fun() -> init_server() end),
    register(myserver, ...).  %% TODO: COMPLETE

start(BootServer) ->
    ServerPid = spawn(fun() -> init_server(BootServer) end),
    register(myserver, ServerPid).

init_server() ->
    process_requests([], [self()]).

init_server(BootServer) ->
    BootServer ! {server_join_req, ...},  %% TODO: COMPLETE
    process_requests([], []).
```

```
process_requests(Clients, Servers) ->
    receive
        %% Messages between client and server
        {client_join_req, Name, From} ->
            NewClients = [...|Clients],  %% TODO: COMPLETE
            broadcast(..., {join, Name}),  %% TODO: COMPLETE
            process_requests(..., ...);  %% TODO: COMPLETE
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(..., ...),  %% TODO: COMPLETE
            broadcast(..., {leave, Name}),  %% TODO: COMPLETE
            From ! exit,
            process_requests(..., ...);  %% TODO: COMPLETE
        {send, Name, Text} ->
            broadcast(Servers, ...),  %% TODO: COMPLETE
            process_requests(Clients, Servers);

        %% Messages between servers
        disconnect ->
            NewServers = lists:delete(self(), ...),  %% TODO: COMPLETE
            broadcast(..., {update_servers, ...}),  %% TODO: COMPLETE
            unregister(myserver);
        {server_join_req, From} ->
            NewServers = [...|Servers],  %% TODO: COMPLETE
            broadcast(..., {update_servers, NewServers}),  %% TODO: COMPLETE
            process_requests(Clients, ...);  %% TODO: COMPLETE
        {update_servers, NewServers} ->
            io:format("[SERVER UPDATE] ~w~n", [NewServers]),
            process_requests(Clients, ...);  %% TODO: COMPLETE

        RelayMessage -> %% Whatever other message is relayed to its clients
            broadcast(Clients, RelayMessage),
            process_requests(Clients, Servers)
    end.

broadcast(PeerList, Message) ->
    Fun = fun(Peer) -> Peer ! Message end,
    lists:foreach(Fun, PeerList).
```

To test the system, first, you need to start a server via `server2:start()` –which creates the initial server including only itself in the list of servers– and then, start several servers via `server2:start({myserver,'server_node@IP'})` –which will connect each server to the rest of servers. Recall to change `IP` with the domain name (or the IP address) of the server you are using to connect to the chat, which can be any server already connected to the system.

CODE REVIEW (2 points)
Clustered server's functionality: Erlang file `server2.erl`.

EXPERIMENTAL MILESTONES (1.25 points)	
MS4 (1 points)	Deploy 3 servers, connect some clients to each of the server instances, and test the chat.
MS5 (0.25 points)	Disconnect some of the servers and see what happens.

OPEN QUESTIONS (1.5 points)

g) What happens to the chat system if a server disconnects?

h) Do your answers to previous questions c), d), and e) still hold in this implementation?

i) What might happen with the list of servers if there are concurrent requests from servers to join or leave the system?

j) What are the advantages and disadvantages of this implementation regarding the previous one? Compare their scalability, fault tolerance, and message latency (measured as the number of hops).