UPPSALA
UNIVERSITET

Assignment 2

Language Abstractions for Concurrent and Parallel Programming

Group 12
Alexis Remmers,
Josef Karakoca,

January 11, 2019

Server Commands	
’user_count’	Shows the number of active users
’quit’	Exits the server and disconnects clients

Client Command	
’list_users’	List all current users connected to the server’
‘connect [user-name]’	Starts a chat with [user-name]
’enter [chat-name]’	Enter a conversation open to the client
’list’	List all open conversations for the user
’exit’	Exit a conversation and return to the main menu
’quit’	Disconnects the client from the server

Table 1: Available commands

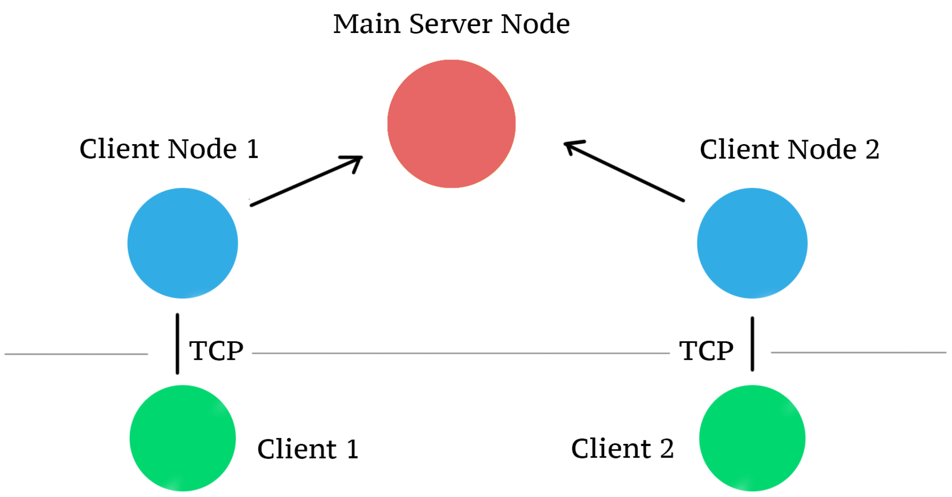
1 Introduction

For our project we chose to write a chat server in Erlang. We were fascinated by the actor model and wanted to learn more about it and a chat server seemed like a good use case that was relatively easy to implement. We have separated the server and the client code and we communicate between them with TCP.

2 Functionality

When a client is started they are asked to enter a user name. After that the client can enter commands (see table below). One command is ’connect [user name]’ which starts a chat with another user. When the command is executed the server notifies the other user that the current user want to start a chat. The other user can enter the chat with ’enter [chatname]’ and list all current chats with ’list’. While inside a chat everything typed is sent to the other client after pressing enter. If a client is inside a chat it can go back to the main menu with the ’exit’ command. When a user is not connected to a chat it can enter the command ’list_users’ to see all active users on the server. Finally it can use ’quit’ to exit from the application. In the server terminal everything is logged and a user also have the ability to enter a limited set of commands. Enter ’user_count’ to see the amount of current connected users or ’quit’ to close down the server.

3 Server Architecture



In the architecture there are one client node on the server side for each connected client. If client 1 (see picture above) wants to do anything on the server it sends a message to client node 1 through TCP, client-node 1 then handles that message and makes a request to the main server. A request could for example be to send a message to another client or to list all active users. We have one main server node that is in a loop waiting for messages from client nodes or from the server admin (the admin can enter commands in the server terminal). In this main server node the state of the server is also stored, all active user names are listed with process ids from the client nodes handling them. There are also a list of the current active chats and which users are connected to each of them.

4 Client

The client is constructed by having input from the keyboard, input from the server and output to the terminal as three separate actors. The point of this were so that all of them can interact independently from each other, if a message would be received while the user were typing a text the actors could make sure that the display would look as one would expect and not have a race condition on the display writer or positional inconsistencies of the terminal cursor.

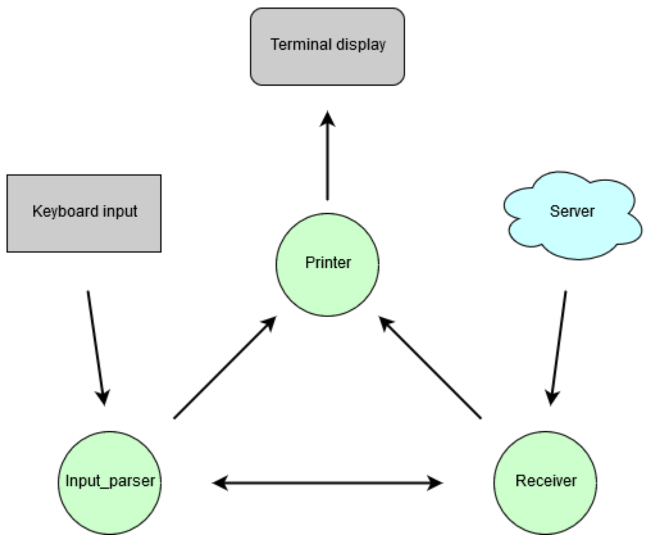


Figure 1: Client communication chart.

Actor: Input_parser

The input parser takes a text line from the terminal, determines what kind of command is stated and executes according to the list stated in Table 1 before looping around for the next input. Some of the commands are only available depending on what location you are in, which it saves in a state. Sometimes it can deduce what position it will have (if you exit a conversation to enter the main menu the parser already know the new position is `'home'`), otherwise it will send the command to the actor "Receiver" and wait for a reply.

Actor: Printer

The printer takes care of everything displayed on the terminal (Except text-input, see Section 7). By doing so we can guarantee atomicity for all print jobs, even if some work is extracted to separate functions for cleaner and more understandable code. The printer relies heavily on ANSI escape codes to move the cursor around and print in certain locations or erasing parts of the screen e.g. text-input, new messages or moving between screens.

Actor: Receiver

The heart of the client, it handles all messages sent from the server, determines what do and what to display on the terminal. To determine what should be displayed the Receiver holds a record with a buffer of all conversations, as well as a variable stating if the user is on the home-page or a specific conversation and if so, which. This is so that while a user is in one conversation, others can still send messages that will be stored locally so the user can read them later. The client only keeps the latest 50 messages in each conversation, we did not want our messages to be stored globally but still have a small buffer so that a user don't miss any information just because he wasn't in the correct conversation at the moment.

5 Use cases: Run the code

First compile both the server.erl and client.erl. Run erlang in one terminal window and type server:start(1055) (1055 is just an arbitrary port number). Then start up two clients in two new terminal windows with client:start(1055) to assume server and client is all local-host, or client:start(HOST-IP, PORT) (note: while it is possible to change host-address, this functionality has not been tested). Enter a user name in each client terminal and start a chat in one of the clients with: 'connect [username]', After this a chat between the clients should be active.

6 Server Code

```erlang
handle_client_msg(S, RawData, ServerPID) ->
    io:format("Raw:\n~s~n", [RawData]),
    try
        {ok, Toks, _Line} = erl_scan:string(RawData, 1),
        {ok, Expr} = erl_parse:parse_term(Toks),

        case Expr of
            {login, UserName} -> ServerPID ! {register, UserName, self()},
                                 send_client(S, ok);
            {msg, RoomID, Msg} -> io:format("Msg <~s>: ~s", [RoomID, Msg]), ServerPID ! {msg, RoomID, Msg, self()};
            {list_users} -> ServerPID ! {list_users, self()},
                            receive {Users} -> send_client(S, Users) end;
            {logout, UserName} -> ServerPID ! {logout, UserName};
            {connect, UserName, Friend} ->  ServerPID ! {connect, UserName, Friend, self()};
            {start_group, Users, RoomID, UserName} -> ServerPID ! {start_group, Users, RoomID, UserName, self()};
            {invite, Friend, RoomID, UserName} -> ServerPID ! {invite, Friend, RoomID, UserName};
            _ -> io:format("Error on:\n~s", [RawData]), 'error'
        end
    catch
        _Class:_ ->
          send_client(S, "Something went wrong")
    end.
```

Listing 1: Client node handler

The code in Listening 1 shows how the client node on the server side handles a message from a client. The 'RawData' on line 4 is the content that was received from the client trough TCP. The RawData is parsed to Erlang tuples which later are matched with the correct case. For example on line 14 the atom 'connect' is matched with that case, the client here then sends a normal Erlang message to request a chat between the current user and its friend.

```erlang
start_client_node(LS,ServerPID) ->
    spawn(fun () -> client_node(LS,ServerPID) end).

client_node(LS,ServerPID) ->
    case gen_tcp:accept(LS) of
        {ok,S} ->
            start_client_node(LS, ServerPID),
            loop(S,ServerPID);
        _ -> ok
    end.
```

Listing 2: How new client nodes are created

The code in Listing 2 shows how new client nodes are created, it's easy but very useful pattern. There is always one client_node() waiting on new TCP connection, when a connection is found it spawns new one to take care of the next TCP connection and go into an receive loop to handle messages from the server and from it's own client.

7 Known shortcomings

There are some shortcoming in our project, due to the way of how the structured the project in the beginning, time constraints or technical difficulties.

Server

- All communication from one client is handled by its own actor but all information still goes through the main server node. This makes it hard to scale this for multiple users and load balance. A perhaps better solution would be that if the client nodes could communicate directly to the client nodes in which the message is supposed to be sent to, and only contact the main server node if a connection is requested to an unknown client, to retrieve the client nodes PID. If this were the case it would be easier to load balance. It would also make it easier to construct an intermediary node between the client nodes as well as the main node, that would start open the port users connect to. That way, you could have multiple addresses and ports open but still able to communicate between each other.

- The server is already constructed to have an arbitrary amount of users in each conversation and work started to have the ability to both create a group conversation with a set of users as well as invite a user into a conversation after the fact, but this functionality have not been fully implemented and does not return the correct messages back to all clients.

Client

- The server does have the ability to have multiple users in a conversation but due to time constraints we did not have time to implement this on the client side.
- Mac-OS does not support all regular ANSI escape codes, specially the command to save and restore the cursor position, this means that when conversing with a user the position in which the user writes text does not stay in the lower part of the terminal.
- It was meant so the printer would take care of everything printed on the output, this would safeguard that if a user would write something while one or multiple messages gets recieved, the cursor position would look like it did not change and all text is printed in the proper location. This was meant to work by sending each character to the printer and the printer would buffer the input until the message was sent. unfortunately there is no easy way to retrieve each character without waiting for the return. This results in visual inconsistencies if either a user writes while the cursor moves around the window when printing or if the position does not return to the correct location if some characters were already inserted. (e.g. the previous point). there are ways to go around this, either by changing how the file descriptor handles I/O before running the program or with an external program written in a lower level language (like C `getchar()`) which forwards all input to the erlang program, each of these solutions comes with their own problems and shortcomings and we had not time to overcome this problem in a correct and visually consistent manner.

8 Conclusions

After working with Erlang on a larger project we can certainly see some of the benefits you can get by using it. Using an architecture with actors in a system where we have many different nodes talking to each other, concurrency and parallelism is really something you get for free. We somewhat explored the supervisor pattern with client nodes but if you would extend the supervisor pattern for the whole architecture it could make it very fault tolerant. The first negative to mention is while the official function documentation is good, finding documentation or forums regarding how some modules and libraries is supposed to be used in Erlang was quite hard, compared to if you would choose to write something in a more popular language (e.g. how to use gen_server concurrently with multiple clients). We spent almost a whole day trying to get simple connections working with TCP using the popular pattern found in the Erlang OTP library, but later after some hassle we decided to go one step back and set up simple TCP connection by our selves instead.

9 Individual contributions

Josef have mostly worked on the back end and Alexis on the client side. We somewhat came up with the system architecture on our own for the server and client but discussed the solution with each other so both had understanding of the actual implementation and both have added and debugged functionality on the others code. We have spent similar amount of time and we most often sat together and programmed.