Assignment 4: Char With Erlang

By Simon Gustav Cordes

Code explanation

This section will explain my solution to the assignment and will present the assumptions leading to this particular solution.

The solution consists of the assignment specified API, some communication methods, a loop for the server and some helper functions for the server loop.

The API

The API consists of the seven functions start/0, connect/2, chat/2, history/1, filter/3, plunk/2 and censor/2.

start/0

The function start/0 spawn a new process which runs the loop/3 function with empty lists of users, messages and user filters.

```
start() -> {ok, spawn(fun () -> loop([], [], []) end)}.
```

The function returns a tuple of type {ok, Ref} where Ref is the process id of the server.

connect/2

Connect sends a message to the server via the request_reply function with the requested nickname.

```
%% connect(Server, Nick)
connect(Server, Nick) ->
    request_reply(Server, {connect, Nick}).
```

All of the error handling and name checking is done by the server. The client outputs the answer when received from the server. To ensure that the client cannot continue unless and answer has been received, the request is synchronous.

chat/2

Chat sends an asynchronous request to the server in the argument, sending along a string message to be broadcast to the other users.

```
chat(Server, Cont) ->
    nonblocking(Server, {chat, Cont}).
```

Everything is handled by the server which doesn't respond to the request.

history/1

The history function takes the server reference as an argument and sends a synchronous request to the server and thus blocks until the server sends the list of the last 42 messages back.

```
history(Server) ->
    request_reply(Server, history).
```

All the logic is handled by the server. It was assumed when making this function that output from the history function should not be filtered.

filter/3

The filter function takes the server reference, a method of either *compose* or *replace* as argument as well as a predicate taking arguments {Nick, Msg}. The function sends a synchronous request to server and waits for the server to reply.

```
filter(Server, Method, P) ->
    request_reply(Server, {filter, Method, P}).
```

The server deals with the logic and error handling of the filters. The server respons with either {ok, Ref} or {error, P, *some message*}.

plunk/2

The plunk function takes as argument a server ref and a nickname to be ignored. The function then uses the filter function to compose a filter which blocks all messages from this user.

```
plunk(Server, Nick) ->
    filter(Server, compose,
        fun({Name, _}) ->
            case Name of
                Nick -> false;
                _ -> true
            end
        end).
```

The request is synchronous as filter is called.

censor/2

The censor function takes as argument a server reference and a list of words that the user does not want to receive messages containing.

```
censor(Server, Words) ->
    filter(Server, compose,
        fun({_, Msg}) ->
            % assume that messages only contain the special characters ,.!?
            Tokens = string:tokens(Msg, " .,!?"),
            case lists:any(fun(Token) -> lists:member(Token, Words) end, Tokens) of
                true ->
                    false;
                _ ->
                    true
            end
        end).
```

The function does this by calling the filter function with the Server, *compose* and a function argument. The function splits a message up into tokens which are then checking against the list of words. If a word from a message is found in the message the message is blocked.

Synchronous communication

All synchronous communication is handled via the request_reply function which sends a request to the server and waits for a reply, which is then returned.

The replying is done with the reply function which takes a pid of some process who've sent a message and returns the reply.

```
%%% Synchronous communication

request_reply(Server, Request) ->
    Server ! {self(), Request},
    receive
    {Server, Response} ->
        Response
    end.

reply(From, Msg) ->
    From ! {self(), Msg}.
```

Asynchronous communication

All asynchronous communication is performed by the nonblocking function. The request is sent to the server along with the process id of the sender. This is because we use the process id in the server loop for various functions

```
%%% Asynchronous communication

nonblocking(Server, Request) ->
    Server ! {self(), Request}.
```

Server implementation

The server is implemented as a loop which maintains a list of users (actually both users PID's and nicknames) , the last 42 messages as ChatHistory, and a list of the filters for all the users on the server. The server essentially just waits for a request and performs these:

```
loop(Users, ChatHistory, Filters) ->
    receive
```

The following sections will go into depth how the different requests are handled by the loop

Connect implementation

When the server loop receives the connect request the following code is executed:

```
{From, {connect, Nick}} ->
    % We reserve the nick admin for use by the server and an admin.
    case Nick == admin of
        false ->
            case lists:any(fun({N, _}) -> N == Nick end, Users) of
                true ->
                    reply(From, {error, Nick, is_taken}),
                    loop(Users, ChatHistory, Filters);
                false ->
                    % Assuming that the same process can try connecting multiple times
                    case lists:any(fun({_, Pid}) -> Pid == From end, Users) of
                        true ->
                            reply(From, {error, From, already_connected}),
                            loop(Users, ChatHistory, Filters);
                        false ->
                            reply(From, {ok, self()}),
                            % We maintain a list of both the user nicks and process ID's.
                            loop([{Nick, From}] ++ Users, ChatHistory, Filters)
                    end
            end;
        true ->
            reply(From, {error, Nick, is_protected}),
            loop(Users, ChatHistory, Filters)
    end;
```

We first check if the Nick is the term *admin*. This serves as a dual purpose – we can either at some point implement an admin function, and we have a reserved nickname that we can check filters against (more on this later). If the nick is admin we reply with a suitable error message and restart the loop. If the nick is not admin we check to see if the nick is found in the list of users. If it is, we return a suitable error message, if not, we reply with an {ok, self()} and add the nick and the process id of nick to the list of users.

We give an answer not matter what so as to not make the client wait indefinitely.

Chat implementation

When the server loop receives the chat request from a client, the following code is executed. It is assumed that the server should only send the message to the other users and not back to the client who sent the message.

```
{From, {chat, Cont}} ->
    case lists:keyfind(From, 2, Users) of
        {Nick, Pid} ->
            case lists:flatlength(ChatHistory) == 42 of
                true ->
                    NewChatHistory = lists:droplast(ChatHistory),
                    BroadcastList = lists:delete({Nick, Pid}, Users),
                    broadcastMsg({Nick, Cont}, BroadcastList, Filters),
                    loop(Users, [{Nick, Cont}] ++ NewChatHistory, Filters);
                false ->
                    BroadcastList = lists:delete({Nick, Pid}, Users),
                    broadcastMsg({Nick, Cont}, BroadcastList, Filters),
                    loop(Users, [{Nick, Cont}] ++ ChatHistory, Filters)
            end;
        false ->
            loop(Users, ChatHistory, Filters)
    end;
```

The PID of the client is sent along to the server so we first find the users nick in the user list via the PIA with lists:keyfind/3. If it is not found we just ignore the request and restart the loop. If it is found, we check if the length of ChatHistory is 42. If so, we remove the last element of the chat history, find a list of the other users and pass this, alongside the sender nick and the message, along to the broadcastMsg function below:

```
broadcastMsg(Msg, Users, Filters) ->
    lists:foreach(fun({Pid, _}) -> sendMsgToUser(Msg, Pid, Filters) end, Users).
```

The broadcastMsg function has the task of calling sendMsgToUser/3 to all the users in the Users list. The filters from the loop are passed along to it, too.

The sendMsgToUser function has the job of filtering the message and sending it if it passes the filter.

```
sendMsgToUser(Msg, UserPid, Filters) ->
    {_, Predicates} = lists:unzip(lists:filter(fun({Pid, _}) -> Pid == UserPid end, Filters)),
    case lists:all(fun(Predicate) -> Predicate(Msg) end, Predicates) of
        true ->
            UserPid ! {Msg, self()};
        false ->
            error
    end.
```

The function first finds the filters belonging to the receiving user. It then tests the message and sender against the all the filters. If it passes all filters, it is sent to the user along with the server process id, if not, nothing happens.

This way of stringing together the program also ensures that all clients receive messages in the same order, short of a user logging out (not currently possible) or a new user joining, since the new user will be first in the user list. None of the calls to the server changes the order of the Users list so it is passed to the sendMsgToUser function in the same order every time it is called.

History implementation

When the server receives the history request, it just replies with sending the ChatHistory list along to the requesting client and restarts the loop:

```
{From, history} ->
    reply(From, {ChatHistory, self()}),
    loop(Users, ChatHistory, Filters);
```

Filter implementation

When the server received a filter request if can be either a *replace* request or a *compose* request. The code for the former is the following

Replace

```
{From, {filter, replace, P}} ->
    % We assume that noone sends empty messages, and we reserve the nickname admin.
    % This serves two purposes -
    %   1) We ensure that the predicate can actually be called and doesn't throw an ex
    %   2) We ensure that admin is not blocked. However, this would require and admin
    try P({admin, ""}) of
        true ->
            FiltersRemoved = lists:filter(fun({Pid, _}) -> Pid /= From end, Filters),
            reply(From, {ok, self()}),
            loop(Users, ChatHistory, [{From, P}] ++ FiltersRemoved);
        false ->
            reply(From, {error, P, cannot_block_admin}),
            loop(Users, ChatHistory, Filters)
    catch
        error ->
            reply(From, {error, P, wrong_predicate_format}),
            loop(Users, ChatHistory, Filters)
    end;
```

We first check with some placeholder values (i.e. the term admin and the empty string “”) to see if the users attempts to block either the admin or the empty message. If this passes, the filters for the user is removed with the lists:filter function, we reply with an {ok, self()} and call the loop again while adding the new filter to the list of filters.

If, on the other hand, the predicate provided calls results in some sort of exception, we return with an appropriate error message. We do this to ensure that the server does not crash because some filter throws an exception because it takes a wrong argument format or because the predicate does not return a Boolean value. This makes the filter functionality robust since we make sure that the user cannot kill the server.

Compose

```
{From, {filter, compose, P}} ->
    % We assume that noone sends empty messages, and we reserv
    % This serves two purposes -
    %   1) We ensure that the predicate can actually be called
    %   2) We ensure that admin is not blocked. However, this
    try P({admin, ""}) of
        true ->
            reply(From, {ok, self()}),
            loop(Users, ChatHistory, [{From, P}] ++ Filters);
        false ->
            reply(From, {error, P, cannot_block_admin}),
            loop(Users, ChatHistory, Filters)
    catch
        error ->
            reply(From, {error, P, not_a_predicate}),
            loop(Users, ChatHistory, Filters)
    end;
```

Similarly to the *replace* method we first check to see if the users blocks the admin or empty message strings. We throw an error message if the predicate blocks the admin or the empty string, and if the predicate does not return a Boolean value or if some other error occurred

Other functionality

If the server receives any other request we simple reply with an error message that the request is unknown and we loop again.

```
{From, Other} ->
    reply(From, {error, Other, unknown_request}),
    loop(Users, ChatHistory, Filters)
```

Testing

This section presents the tests and the results of the tests. The source code for the tests can be found in the erctests.erl file found in the src folder of the submitted .zip file.

The tests test all the functionality of the API. All of the tests can be reproduced by running the code as via the Erlang shell as presented in following sections.

Start tests

The first start test tests whether a server can be started. The other tests whether several can be started.

```
% tests if we can start a ERC server
startTest1() ->
    {ok, Ref} = erc:start(),
    Ref.

% tests if we can start multiple ERC servers
startTest2() ->
    {ok, Ref1} = erc:start(),
    {ok, Ref2} = erc:start(),
    {ok, Ref3} = erc:start(),
    {Ref1, Ref2, Ref3}.
```

Result

The tests give the following results:

```
6> erctests:startTest1().
<0.505.0>
7> erctests:startTest2().
{<0.507.0>,<0.508.0>,<0.509.0>}
8>
```

From these we can conclude that the start function works as intended

Connect tests

Test 1

The first test tests whether it is possible to connect a user to a server.

```
% tests if we can connect to an ERC server
connectTest1() ->
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user),
    Ref.
```

Result

The test gives the following result:

```
8> erctests:connectTest1().
<0.511.0>
```

This shows that it the function works for this small example.

Test 2

The second test tests whether it is possible to add several clients to the server.

```
% tests if we can connect to a server with multiple diff
connectTest2() ->
    Me = self(),
    {ok, Ref} = erc:start(),
    Client1 = spawn(fun() ->
                        Ret = erc:connect(Ref, user1),
                        Me ! {self(), Ret}
                    end),
    receive
        {Client1, Ret1} ->
            Ret1
    end,
    Client2 = spawn(fun() ->
                        Ret = erc:connect(Ref, user2),
                        Me ! {self(), Ret}
                    end),
    receive
        {Client2, Ret2} ->
            Ret2
    end,
    Client3 = spawn(fun() ->
                        Ret = erc:connect(Ref, user3),
                        Me ! {self(), Ret}
                    end),
    receive
        {Client3, Ret3} ->
            Ret3
    end,
    {Ret1, Ret2, Ret3}.
```

The result from this test is the following:

```
9> erctests:connectTest2().
{{ok,<0.513.0>},{ok,<0.513.0>},{ok,<0.513.0>}}
```

This shows that all three users could connect to the server without problem.

Test 3

The third test tests whether the server throws an error of the format {error, admin, is_protected} if someone tries to connect with the nickname admin.

```
connectTest3() ->
    {ok, Ref} = erc:start(),
    {error, admin, is_protected} = erc:connect(Ref, admin).
```

Result

Running the test gives the following output:

```
10> erctests:connectTest3().
{error,admin,is_protected}
```

This shows that the server blocks the request as it should.

Test 4

The fourth test attempts to add two users with the same user name. This, of course, should lead to an error message.

```
connectTest4() ->
    Me = self(),
    {ok, Ref} = erc:start(),
    _ = spawn(fun() ->
                  erc:connect(Ref, user),
                  Me ! done
              end),
    receive
        done -> ok
    end,
    {error, user, is_taken} = erc:connect(Ref, user).
```

Result

The result of running the test is the following:

```
11> erctests:connectTest4().
{error,user,is_taken}
```

This shows that this functionality works as expected.

Test 5

The fifth test attempt for the client to add itself several times under different aliases. This should throw an error message of the type {error, self(), already_connected}.

```
connectTest5() ->
    Me = self(),
    {ok, Ref} = erc:start(),
    _ = erc:connect(Ref, user),
    {error, Me, already_connected} = erc:connect(Ref, user2).
```

Result

This gives the following result:

```
12> erctests:connectTest5().
{error,<0.485.0>,already_connected}
```

This shows that you cannot connect to the same server more than once.

Conclusion

The tests in this section all pass showing that the connect functionality works as expected and that the connect function adheres to the problem specification.

Chat tests

This section shows the code and the results for the tests done to the chat function. There are two tests done testing the chat function.

Test 1

The first chat tests whether user2 receives the message “hello” sent by user1. The message is then sent back and printed.

```
chatTest1() ->
    Me = self(),
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    Spawn = spawn(fun() ->
                      erc:connect(Ref, user2),
                      Me ! done,
                      receive
                          Msg ->
                              Me ! {self(), Msg}
                      end
                  end),
    receive
        done ->
            ok
    end,
    erc:chat(Ref, "hello"),
    receive
        {Spawn, Msg} ->
            Msg
    end.
```

Result

The result of the test is the following

```
13> erctests:chatTest1().
{{user1,"hello"},<0.525.0>}
```

This shows that the message from user1 is actually received by user2 and sent back to user1 again.

Test 2

The second test tests the same functionality, although with several users. user1 sends a message which should be sent to two other users. If so, we should receive two answers from user2 and user3, one from each. This test might be a bit cryptic, however, it tests the implementation in a satisfactory way.

```
chatTest2() ->
    Me = self(),
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    spawn(fun() ->
                    erc:connect(Ref, user2),
                    Me ! done,
                    receive
                        Msg ->
                            Me ! Msg
                    end
                end),
    receive
        done ->
            ok
    end,
    spawn(fun() ->
                    erc:connect(Ref, user3),
                    Me ! done,
                    receive
                        Msg ->
                            Me ! Msg
                    end
                end),
    receive
        done ->
            ok
    end,
    erc:chat(Ref, "hello"),
    receive
        Msg1 ->
            Msg1
    end,
    receive
        Msg2 ->
            Msg2
    end,
    {Msg1, Msg2}.
```

Result

The test gives the following result:

```
14> erctests:chatTest2().
{{{user1,"hello"},<0.528.0>},{{user1,"hello"},<0.528.0>}}
```

Since we have a tuple of two messages we can conclude that it works as intended since both user2 and user3 have received the message.

Conclusion

Since both tests gave the expected result we can conclude that chat works as intended and adheres to the problem specification.

History test

This section presents the only test done to history. It is a lightweight function with not much to test,

The restriction of 42 messages in the list has, however, not been tested.

```
historyTest1() ->
    Me = self(),
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    erc:chat(Ref, "msg1"),
    erc:chat(Ref, "msg2"),
    spawn(fun () ->
            erc:connect(Ref, user2),
            erc:chat(Ref, "msg3"),
            erc:chat(Ref, "msg4"),
            erc:chat(Ref, "msg5"),
            Me ! done
          end),
    receive
        done ->
            ok
    end,
    erc:chat(Ref, "msg6"),
    erc:chat(Ref, "msg7"),
    {Ret, Ref} = erc:history(Ref),
    Ret.
```

Result

The result of the test is the following:

```
18> erctests:historyTest1().
[{user1,"msg7"},
 {user1,"msg6"},
 {user2,"msg5"},
 {user2,"msg4"},
 {user2,"msg3"},
 {user1,"msg2"},
 {user1,"msg1"}]
```

This shows that history prints the last messages as needed.

Filter tests

The following tests test the filter function.

Test 1

The first test tests whether the filter of “hello” messages work. It so, the only output should be {user2, “hi”}.

```
filterTest1() ->
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    {ok, Ref} = erc:filter(Ref, compose, fun({_, Msg}) ->
                                            case Msg of
                                                "hello" ->
                                                    false;
                                                _ ->
                                                    true
                                            end
                                        end),
    spawn(fun () ->
              erc:connect(Ref, user2),
              erc:chat(Ref, "hello"),
              erc:chat(Ref, "hi")
          end),
    receive
        {Msg, Ref} ->
            Msg
    end.
```

Result

The result of the test is the following:

```
19> erctests:filterTest1().
{user2,"hi"}
```

This shows that the filter function can filter messages.

Test 2

The second filter test tests whether the replace filter function works. If so, the only thing outputted should be {user2, “hello”} since we replace the filter blocking “hi” with another.

```
filterTest2() ->
    Me = self(),
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    spawn(fun () ->
            erc:connect(Ref, user2),
            erc:chat(Ref, "hello"),
            erc:chat(Ref, "hi"),
            Me ! done
        end),
    receive
        done ->
            ok
    end,
    receive
        _ ->
            receive
            _ ->
                ok
            end
    end,
    {ok, Ref} = erc:filter(Ref, compose, fun({_, Msg}) ->
                                            case Msg of
                                                "hello" ->
                                                    false;
                                                _ ->
                                                    true
                                            end
                                        end),
    {ok, Ref} = erc:filter(Ref, replace, fun({_, Msg}) ->
                                            case Msg of
                                                "hi" ->
                                                    false;
                                                _ ->
                                                    true
                                            end
                                        end),
    spawn(fun () ->
            erc:connect(Ref, user3),
            erc:chat(Ref, "hello"),
            erc:chat(Ref, "hi")
        end),
    receive
        {Msg, Ref} ->
            Msg
    end.
```

Result

The result of the test is the following:

```
20> erctests:filterTest2().
{user2,"hello"}
```

This shows that the replace function works as intended.

Test 3

The third tests whether the server recognizes functions that don't output Boolean functions when using the replace option. If so, an error should be thrown:

```
filterTest3() ->
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user),
    {error, _, wrong_predicate_format} = erc:filter(Ref, replace,
                                                    fun({_, _}) -> kage end).
```

Result

The result of the test is the following:

```
21> erctests:filterTest3().
{error,#Fun<erctests.14.93042727>,wrong_predicate_format}
```

This shows that it works as intended in this case.

Test 4

The fourth tests whether the server recognizes functions that don't output Boolean functions when using the compose option. If so, an error should be thrown:

```
filterTest4() ->
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    {error, _, wrong_predicate_format} = erc:filter(Ref, compose,
                                                    fun({_, _}) -> kage end).
```

Result

The result of the test is the following:

```
22> erctests:filterTest4().
{error,#Fun<erctests.15.93042727>,wrong_predicate_format}
```

This shows that it works as intended

Plunk tests

This section presents the plunk tests

Test 1

The first tests whether the plunk function works by blocking user2 and sending a message from user2. If the message doesn't arrive we should see that plunktest timed out:

```
plunkTest1() ->
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    {ok, Ref} = erc:plunk(Ref, user2),
    Spawn = spawn(fun() ->
                erc:connect(Ref, user2),
                receive
                    go ->
                        erc:chat(Ref, "hello")
                end
              end),
    Spawn ! go,
    receive
        Msg ->
            Msg
    after 5000 ->
        io:format("plunkTest timed out", [])
    end.
```

Result

The result of the test is the following:

```
24> erctests:plunkTest1().
plunkTest timed outok
```

Works as intended.

Test 2

The second test tests whether you can block the admin. An error should be thrown

```
plunkTest2() ->
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    {error, _, cannot_block_admin} = erc:plunk(Ref, admin).
```

Result

The result of the test is the following:

```
25> erctests:plunkTest2().
{error,#Fun<erc.1.9328301>,cannot_block_admin}
```

The error is thrown and the function works as intended.

Censor tests

This section presents the only test for censor.

The test censors the word “kage” and then tries to send a message containing “kage”. The function should time out.

```
censorTest() ->
    {ok, Ref} = erc:start(),
    {ok, Ref} = erc:connect(Ref, user1),
    {ok, Ref} = erc:censor(Ref, ["kage"]),
    Spawn = spawn(fun() ->
                      erc:connect(Ref, user2),
                      receive
                          go ->
                              erc:chat(Ref, "kage")
                      end
                  end),
    Spawn ! go,
    receive
        Msg ->
            Msg
    after 5000 ->
        io:format("censorTest timed out", [])
    end.
```

Result

The result of the test is the following:

```
27> erctests:censorTest().
censorTest timed outok
```

Works as intended.

Final Conclusion

The tests in this section show that the implementation works as intended and conforms to the API description in the assignment, AND that it take into account many different special cases.