Assignment 4: Chat with Erlang

Advanced Programming, fall 2016

Nikolaj Høyer
ctl533@ku.dk

Andrew Tristan Parli
qcm239@ku.dk

1. INTRODUCTION

In this assignment, we describe an implementation of a relay chat server written in Erlang, named ERC.

In Section 2 we will go through the base for the chat client, describing functions for starting a new chat server, connecting to a chat server, sending messages to clients connected to the chat server, and showing a history of recent messages. In Section 3 we'll briefly discuss ordering of messages, while in Section 4 we will go through a generic function for adding client message filters along with two particular filters: plunk and censor. In Section 5 we'll discuss compliance with the API given in the assignment text and also discuss blocking vs non-blocking functions. Finally we'll describe testing methodologies in Section 6.

2. BASIC IMPLEMENTATION

For the skeleton of the server, we've used the one provided during the lecture: we have certain client API functions, blocking and async communication primitives and a loop containing our state variables. Furthermore, we've split the logic inside the loop into several distinct functions for readability.

2.1 Start

This function initializes the state variables and starts the server side loop. For the server reference, we generate an id that can be passed to clients, identifying the server. For clients, filters and the message log we initialize empty lists.

Next, we attempt to spawn a new process, which will run our loop - this should not fail, but in case of an anomaly in the system, we catch the error and display a message to the user.

2.2 Connect

This function takes a server id and a nickname in the form of an atom, validates these as pid and atom, respectively, and pass the request to our loop.

In the server logic - `connect_logic` more specifically, we first check if the nickname is taken by another user by looking through the Client state list, and return an error-message to the client if this is the case. Otherwise we proceed to add the nick of the client, along with the id of the client, to the Client state list.

2.3 Chat

For this function we have error handling ensuring that both server and message input are `pid` and `list`, respectively. For the message input we also check if each element is an integer within the ASCII alphabet. Then we pass the message to our loop, invoking our server side logic.

On the server side we first make sure that the calling client exist in the clients state - that the client is actually connected before sending messages.

Next, we add the message to the log. Note that we only store the latest 42 messages, which makes the implementation of `history` very simple. For the purpose of this assignment we argue that it is sufficient, but for production purposes a larger log would probably be needed and could be written to disk and cleared in memory once in a while.

Finally we send messages using the `foreach` function on the clients state list with a custom `SendMsg` function. This function takes a `pid` and looks for filters in the filter state list (see Section 4), sending a message to the input `pid` if no filters block this.

2.4 History

Since we have chosen to only store the latest 42 messages in our message log state, the implementation of this function is very straightforward.

We simply check the validity of the server input using `is_pid`, then pass the call to the server logic, which sends the message log state back to the client.

3. ORDERING OF MESSAGES

We cannot guarantee that clients will receive messages in the same order.

Every call to `chat/2` is asynchronous, meaning that multiple instances of the `lists:foreach(SendMsg, Clients)` (`src/erc.erl`, line 250) function may run in parallel.

4. FILTERING

4.1 Filter

We have chosen to store filters as a list consisting of tuples in the following form: `{ClientPid, [Pred1, Pred2, ..., PredN]}` and include this in our loop along with the server reference, connected clients and our message log.

On the client API level (the `filter(Server, Method, Pred)` call line 89 in `src/erc.erl`), we check for validity of both the server id, the method used and the predicate - if an error is found, we throw an error to the user indicating which part did not match the expected value or function. This is necessary, as otherwise it would be possible to pass all kinds of values instead of for example the predicate function, since functions in Erlang are treated the same way as values.

For the server id, we simply check that it's an id with the `is_pid` function. For the method, we check if it's an atom with the `is_atom` function and check that it's either the `compose` or the `replace` atom. For predicates, we first check if it's a function with the `is_function`. Then we try out the `Pred` function with some generic parameters and check if the result is either true or false - that is, we both check the parameters of the function and that it outputs a boolean atom. Because of this, we would argue that our filter function is quite robust.

In the backend `filter_logic` function, we simply look through the filter list for any client matching the requester - if no one is found, we add a new filter entry. Otherwise we either append the new filter to the clients filters or replace them, depending on whether `compose` or `replace` was used.

4.2 Plunk

This function creates a predicate that checks whether the supplied `Nick` equals that received by the predicate. This is then passed to the `filter` function along with the client id and the `compose` atom for storage in the filter state.

The client API function also includes error handling - we check with `is_pid` that the server provided is actually an id, and with `is_atom` that the nick provided is a valid nick.

4.3 Censor

This function creates a predicate that looks through each element in the supplied list of words, checking for each token (delimited by whitespace) in a message if there is a match. This is then passed to the `filter` function along with the client id and the `compose` atom for storage in the filter state.

The client API function also includes error handling - we check with `is_pid` that the server provided is actually an id, and with `is_list` that the words provided is actually a list. We have not, however, included error handling to ensure that the words provided actually are strings. This would probably be a good idea (and we already have the implementation, see section 2.3), but would also introduce additional overhead, since we would need to check that each element in the list is a valid numerical value corresponding to a character.

5. API COMPLIANCE

5.1 General remarks

Our implementation complies with the API, with the exception of an added function, `get_filters/1` for listing the filters currently installed for all clients, which was necessary for testing. We refer to Section 6 for tests verifying this claim.

5.2 Blocking vs non-blocking

All functions except `chat` are blocking.

- `connect/2` is blocking, since the connecting user needs some kind of response confirming the connection and providing the reference id of the server.
- `chat/2` is non-blocking (asynchronous), as the need for a server response to the user is irrelevant compared to the performance penalty this would incur on the server. Especially with many clients sending messages simultaneously blocking other server calls could severely impact performance.
- `history/2` is blocking, since the server needs to send a message back to the user containing the chat history.
- `filter/2` could be both blocking and non-blocking, but we have chosen to implement it as blocking. Filtering *should* be a fairly infrequent function call (compared with `chat/2` at least), and confirmation that the desired filter has actually been applied outweighs, in our opinion, the performance penalty associated with blocking.
- `plunk/2` and `censor/2` both make use of the `filter/2` function for their server calls, so these are blocking as well, and for the same reason as `filter/2`.

6. TESTING

We have tested in three different ways: with the Online TA, with our own test suite and manually in the Erlang shell. Each are described in the following sections.

6.1 Online TA

Uploading the code to the Online TA with `curl-Fsrc=@src.zip-Freport=@report.pdf-Fgroup=@group.txthttps://ap16.onlineta.org/grade/4` yields the following result:

```
======================== EUnit ========================
erc_tests: test_connect...[0.001 s] ok
erc_tests: test_nick_taken...ok
erc_tests: test_chat_is_nonblocking...ok
erc_tests: test_chat_with_self...ok
erc_tests: test_chat_with_many...[0.001 s] ok
=======================================================
  All 5 tests passed.
---
Looks good enough to me..
Farewell young padowan.
---
```

6.2 Our own test suite

We have written some tests of our own as well - these are not exhaustive, however. Start, connect, chat, history and adding filters are tested, but using actual filters are not. For testing filters, an additional method, `get_filters/1`, has been added. This should probably not be included as a generally available function in a final version of the chat server, but has been included in this version for testing purposes.

Moreover, there seems to be an issue with validating that the chat function actually works - we have noted that removing the actual chat functionality: `lists:foreach(SendMsg, Clients)` (`src/erc.erl`, line 250) does not result in any errors.

Our tests give the following results:

```
Eshell V8.1  (abort with ^G)
1> c(erc).
{ok,erc}
2> c(erc_tests).
{ok,erc_tests}
3> erc_tests:test().
  All 30 tests passed.
ok
```

Tests consists of the following methods, which can also be run individually for a verbose mode:

- `erc_tests:export_test_().`
- `start_test_().`
- `connect_test_().`
- `chat_test_().`
- `chat2_test_().`
- `filter_test_().`

Especially `chat_test_().` and `chat2_test_().` are interesting, as they have a lot of output.

6.3 Manual testing

During development, we have tested manually with the Erlang shell in the following way:

```
Eshell V8.1  (abort with ^G)
1> c(erc).
{ok,erc}
2> {_, S} = erc:start().
{ok,<0.64.0>}
3> erc:connect(S, n).
{ok,#Ref<0.0.3.183>}
4> erc:chat(S, "hello").
{<0.57.0>,{chat,"hello"}}
5> erc:chat(S, "how are you?").
{<0.57.0>,{chat,"how are you?"}}
7> erc:history(S).
[{n,"how are you?"},{n,"hello"}]
```

Filters, plunk and censor are tested this way as well, but only with debugging `io:fwrite`'s, as we did not manage to get tests working for this part - everything seemed to work as desired though.