**Principles of Concurrent Programming HT19**

**TDA384/DIT391**

Canvas room (CTH)

Canvas room (GU)

Home

Information

Lectures

Labs

Reading material

Computing resources

Exam

Exercises

# Lab 2: CCHAT

### Deadlines

- See labs page

- To pass this lab you have to give a short demo of your submission to the TAs on 10 October 2019; to do that, you have to sign up here for room ED3507 and here for room ED3582 by 9 October 2019, indicating your group number in Fire, to select a time slot. Please try to split evenly among the available rooms. If it is important to you to pick a specific time slot, make sure to sign up early.

### Lab description

In this lab, you will build a simple text-based messaging system called CCHAT. CCHAT is very much inspired by IRC, an old but still valid standard designed for group discussions. For simplicity, your implementation of CCHAT is not going to use IRC's protocol or low-level TCP/IP communication. Instead, it will leverage Erlang's processes and message passing features.
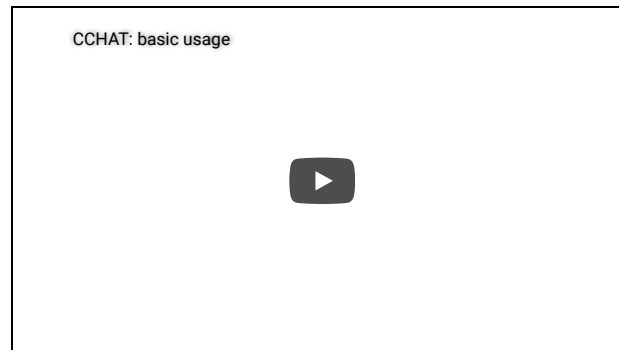
#### Programming language

This lab assignment must be developed in Erlang.

How to install Erlang/OTP in your computer.

#### Overview

The video below shows how your implementation of CCHAT should work.



CCHAT: basic usage

#### Erlang functions

- `cchat:server()` starts a chat server process and registers it to a pre-defined name. For the rest of the lab, users will chat through that server.

- `cchat:client()` opens a new client window where the chat commands (below) can be entered.

#### Chat commands

- `/join #channel` Join a channel with the given name. By convention, all the channels' names start with #.

- `/leave` Leave the current channel.

- `/leave #channel` Leave the specified channel.

- `/whoami` Return current nick (see the System tab).

- `/nick newnick` Change nick to `newnick`. The format for nicks is the same as for an Erlang atom, i.e. it starts with a lower case letter and is followed by letters, numbers, or underscore (_).

- `/quit` Quit the client.

#### Architecture

CCHAT uses a client/server architecture. It consists of a client part, which runs the GUI and a client

process, and a server, which hosts the chat channels.
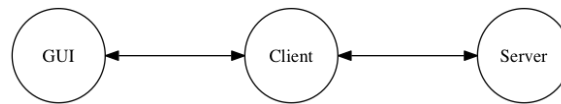
- Location of the client and server processes

  The picture suggests that the client and server processes might be located in different machines. For simplicity, *we will assume that all the processes are located on the same local machine*. If you get the implementation of CCHAT right, you will be able to easily adapt it to run in a distributed environment (see *Location transparency & distribution* in Lecture 7).

- The GUI and the client process

  The GUI will interact with the client process through a specific protocol (described below). You do not need to implement the GUI, as this will be given to you. Instead, you need to provide a implementation of the client process. Your implementation should know how to interact with the GUI, i.e., it should follow the GUI protocol (to be explained later on).

- The server process

  The server handles the requests coming from the clients. The protocol being used between a client and the server is up to you. Take into account that the chat server might be composed by several processes, not only the one that you see in the picture above. Think of the parts in your code where it makes sense to increase concurrency.

The GUI/client protocol

The protocol between the GUI and the client process is fixed. By following the protocol, you will be able to use the GUI without knowing its internal implementation details. In addition, we will test your code assuming that your client process follows the protocol. If you do not follow it, your code will not pass the tests (see the test section below) and your submission will be immediately rejected.

The GUI sends a message requesting some operations. The client either replies with the atom `ok`, meaning the operation succeeded:
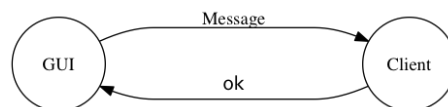
Alternatively, the client replies with the tuple `{error, Atom, Text}`, denoting that something went wrong while processing the request (see the Error Handling section below):
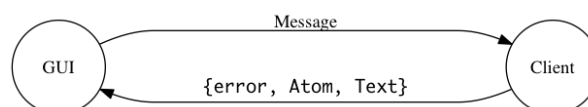
Supported operations

- Joining a channel

  To join a channel, users write the command `/join Channel`, where `Channel` is a string starting with "#", e.g., "#hobbits". The GUI sends the message `{join, Channel}`.

  Internally, if the channel does not exists in the server side, the server process will create it. We assume that, once created, channels are never destroyed, i.e., they will always exist as long as the server runs. Bear in mind that only users who have joined a channel can write messages on it.

  Atom `user_already_joined` is returned when a user tries to join to the same channel again.

- Writing messages in a channel

  When the user is in a channel and writes an string (not starting with /), the GUI sends the message {message_send, Channel, Msg} where Channel contains the name of the channel as an string (e.g. "#hobbits") and Msg stores the message (e.g. "hello fellow hobbits").

  Atom user_not_joined is returned when a user tries to write a message in a channel that the user has not joined.

- Leaving a channel

  When the user types /leave in a channel, the GUI sends the message {leave, Channel}, where Channel contains the name of the channel.

  Atom user_not_joined is returned when a user tries to leave a channel that the user has not joined.

- Asking for the nickname

  When the user writes the command /whoami, the GUI sends the message whoami to the client process. The client should respond with the nick as a string (instead of the atom ok). There are no errors to report in this case.

- Changing the nickname

  When the user writes the command /nick Name, the GUI sends the message {nick, Name} to the client process. Variable Name contains the new chosen nickname for the user.

  Distinction assignment for 2 points: If you want to check for duplicate nicks, you should return atom nick_taken when trying to change to a nick that is already taken. (See below for more details).

- Receiving messages

  Until this point, the protocol describes communications initiated by the GUI. There is only one occasion when the client process starts communication with the GUI: when something is written to a channel, the client needs to tell the GUI to display the new text. The client process sends the message {message_receive, Channel, String} when it wishes to print out the line String in the channel Channel.
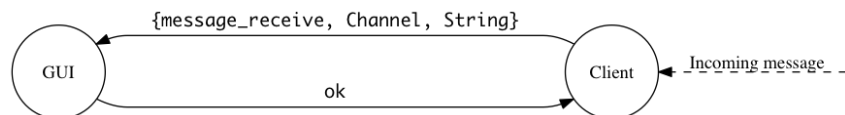


### Error handling

- Recoverable errors

  Sending the tuple {error, Atom, Text} to the GUI denotes that something went wrong while processing the request. These errors are not fatal and the GUI can recover from them. Variable Text contains the text to be reported in the System tab of the GUI. You are free to choose the value of Text, but you should strictly use the values for Atom as described by the protocol.

  In the case when there is no response from the server (e.g. it is non-existent, or non-responsive), return the error atom server_not_reached.

- Fatal errors

  The client process has the chance to respond to the GUI with the tuple {'EXIT', Reason}. This tuple indicates to the GUI that something went very wrong on the client side, e.g. when the server process crashes in the middle of processing a request. The GUI will display the content of variable Reason and exit. You might see this behavior during development but it should not appear in your submission when we test it.

### Assignment

Your assignment is to implement the chat server described here, based on the skeleton available below in the code skeleton section.

Successful solutions require that you do the following:

- You should implement all the TODO sections in client.erl and

`server.erl`. You may to choose to create extra modules to help keep your code organised, if needed.

- Your code should pass the test cases.

- You must use message passing. Directly invoking functions between client/server modules is strictly forbidden.

## Submission:

- You should submit the files `client.erl`, `server.erl`, and any other files which are needed for your solution to work (do not submit the `lib` folder).
- Do not submit compressed archives. Just upload the individual source files which you have worked in.
- No separate documentation is required, but you should comment your code so that graders can easily review your submission.

## Code skeleton

Download the skeleton code package here

The package conists of the following files. Do not edit (or submit) the ones highlighted in red!

- Client process: `client.erl`

  The exported function `client:handle/2` handles each different kind of request from the GUI, returning a tuple of the response and the updated state.

- Server process: `server.erl`

  For the server you need to implement the funtions `start/1` and `stop/1`. How you go about this is up to you!

- Internal libraries: `lib/`

  - CCHAT: `cchat.erl` This is the top level module. It is used to launch the server and clients with their respective GUIs.

  - GUI: `gui.erl`, `lexgrm.erl`, `grm.yrl`, `lex.xrl`, `grm.erl`, `lex.erl` These files contain the implementation of the GUI.

  - Generic Server: `genserver.erl` This file contains functions for spawning and running Erlang processes as servers and implementing synchronous message passing. It is used internally, but you might also want to use its functions yourselves.

  - Testing: `test_client.erl`, `dummy_gui.erl` Files used for testing.

## Compilation

After you have downloaded the skeleton code, compile everything with the command:

```
make all
```

You can then open the Erlang shell (`erl`) and start a client with `cchat:client()`. You should be able to open up chat windows, but users will not able to communicate because most functions are not implemented.

## Test cases

We provide some unit tests which check the basic functionality of your solution. All unit tests are contained in the file `lib/test_client.erl` and are run using EUnit. We have created entries in the `Makefile` to make life easier for you. To run these tests, simply execute the following:

```
make -s run_tests
```

If your submission fails any of the tests, it is a good indication

that your solution does not work correctly. However even if you pass all of the tests, that is not a guarantee that your submission will be accepted.

Turning off color codes in test suite

If you don't have a color-enabled terminal, you will see a lot of ugly color codes in your test output. You can disable these by commenting out the `colour` function in `lib/test_client.erl` and replacing it with:

```
colour(Num,S) -> S.
```

## Tips

- Start by implementing `server:start/1`, spawning a server process which replies to all requests with some default message. Then establish communication between client and server, e.g. implementing the `/join` command.

- The server process must be registered to the atom provided as the argument to `server:start/1`. So when the server name is *shire*, you can send messages directly to the atom `shire` without knowing the server's process ID.

- Think carefully about how you want to implement your server and how channels should be managed. Remember this is a course on concurrent programming — you should introduce concurrency wherever it is suitable. Some test cases will fail if you do not use concurrency where necessary.

- While optional, we suggest that you use the `genserver` module to handle the send/receive/loop cycle required in the server process.

- If you are not using genserver in your solution, then when sending chat messages to clients you will need to do something like the following:

```
Ref = make_ref(),
Pid ! {request, self(), Ref, {message_receive, Channel, Nick, Msg}},
receive
  {result, Ref, Data} -> ...
end
```

- We suggest you use Erlang's record syntax to make updating your state variables easier (see examples of this in `client.erl`).

- To dump the contents of a variable S to the console for debugging purposes, try:

```
io:fwrite("~p~n", [S])
```

- To clean all compiled files, use `make clean` from the system shell.

- To compile, load Erlang, start a server and two clients in one go, try the following command from the system shell:

```
make && erl -eval "cchat:server(), cchat:client(), cchat:client()."
```

- To run a single test, use the following (replacing `message_throughput` with whichever test you want to run):

```
make && erl -eval "eunit:test({test, test_client, message_throughput_test}), halt()."
```

- Erlang's `badarg` error message can be a bit confusing. If you get this when sending a message, it probably means there is no process registered to the name you are sending to.

## Distinction assignment

The distinction assignment will extend of the functionality of the chat and, if implemented correctly, is worth 2 points.

By default, every client gets a random nick. This can be changed with the command `/nick newnick`, for which a basic implementation has been provided. But what happens if two

## Principles of Concurrent Programming HT19

### TDA384 / DIT391

clients choose the same nick? As a distinction assignment, you should disallow this. In other words, when a user tries to change their nick, make sure it is not already taken before allowing the user to change it! You can return the error atom `nick_taken` when this is the case. (You do not need to handle the case when the initial random nick collides with an existing nick.)