Erlang TinyChat Assignment*
Due: 19 November, 11:59PM

v0.02

Elliot Wasem, Eduardo Bonelli

The goal of this assignment is to create a simple chat application in Erlang which we dub TinyChat. This document is structured as follows:

- Section 1 presents the assignment policies.
- Section 2 describes the actors involved in the TinyChat application.
- Section 3 describes the message protocol associated to each operation that is to be supported by the actors of TinyChat.
- Section 4 presents submission instructions.

1 Assignment Policies

Collaboration Policy. This homework may be done individually or in pairs. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances can code be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be reused. Violations will be penalized appropriately.

Late Policy. Late submissions are allowed with a penalty of 2 points per hour past the deadline.

2 A High-Level Overview of TinyChat

TinyChat consists of 4 primary actors or process types and their corresponding Erlang modules:

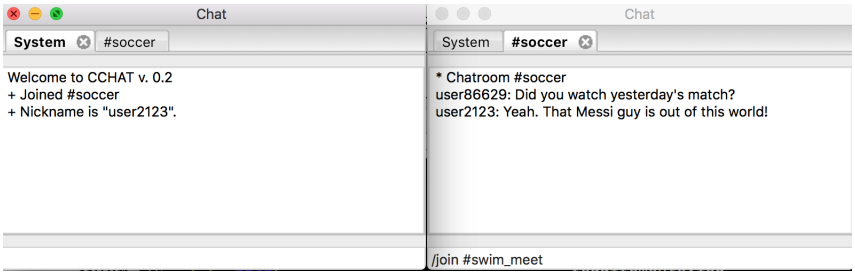
*Based on `http://www.cse.chalmers.se/edu/year/2018/course/TDA384_LP1/cchat/`. Note, however, that this assignment is different both in its actors and messaging protocols.

- GUIs (`gui.erl`): code provided for you.
- Clients (`client.erl`): must complete stub provided.
- A server (`server.erl`): must complete stub provided.
- Chatrooms (`chatroom.erl`): must complete stub provided.

There is also a `main.erl` module to start the program, as well as some additional auxiliary files that will be described below (all of which are provided).
The good news is that you are given the entirety of `gui.erl`, and part of `client.erl`! The even better news is that you will still have plenty of opportunity to practice message passing throughout the rest of the modules!

But how does this all work?

In addition to the process types mentioned above, there is of course the human user "process". Its role is important because the functionality of this program will be described in terms of the user's commands. The user can enter commands into the graphical user interface or GUI. Here is an image of two GUI processes, pictured side-by-side, corresponding to two different clients:



Each GUI process has a *system tab* named "System". It also has zero or more *chatroom tabs*, corresponding to each of the chatrooms that the human user is participating in. Chatroom names always start with a hashtag. In the example above only one chatroom is in use, namely `#soccer`. The current active/open tab is indicated by writing its name in boldface. Each GUI process also has a *command line entry pane*, located at the bottom of the window. In order to interact with the TinyChat system, users may enter any of the following commands in the command line entry pane:

`/join #chatroom_name`	join chatroom with name "`#chatroom_name`".
`/leave`	leave chatroom who's tab is currently active/open.
`/leave #chatroom_name`	leave chatroom with name "`#chatroom_name`".
`/whoami`	what is current nickname?
`/nick new_nickname`	change nickname from current nickname to "`new_nickname`". Must start with lowercase
`some string of text`	send a message "`some string of text`" to chatroom who's tab is currently active/open.
`/quit`	quit. Just that user's GUI and client exit, while server and chatrooms stay running.

2.1 Processes and their Local State

Each process has a local state with its properties. For example, the local state of a client includes its nickname and the chatrooms that it is connected to. The local state of a process is modeled as a parameter, of some appropriate record type, of the main loop function of the process (typically called `loop`). This section describes the local state, and the corresponding record type that holds it, for each process in TinyChat that you will have to implement, namely client, server and chatroom. All record types introduced below are declared in the header file `defs.hrl`. Figure 1 provides a high-level view of which actors interact with each other.

2.1.1 Client (`client.erl`)

The client has the following properties: a nickname that is unique across the running of this program, a number of chatrooms the client is connected to, and the name of the gui process with which it has an exclusive connection. These properties make up the client's local state. It is should always be kept track of in a record of type `cl_st`. Its declaration is given below:

```
-record(cl_st, {gui, nick, con_ch}).
```

`cl_st` consists of the following fields:

- **gui**: a string representing the name under which the client's corresponding GUI process is registered. To look up the GUI's PID, use `whereis(list_to_atom(State#cl_st.gui))`. The GUI name should be stored as a string.
- **nick**: the client's current nickname. Nickname should be stored as a string.
- **con_ch**: a map from chatroom names (string) to chatroom PIDs; this map models all chatrooms to which the client is registered.

This state should be passed through each call to whatever repeated function you use to run/loop your client.

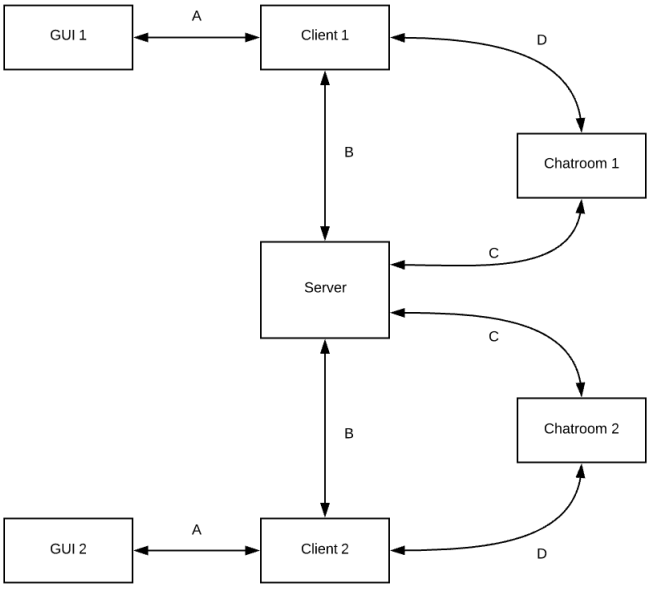


Figure 1: Layout of processes and connections.

2.1.2 Server (`server.erl`)

The server has the following properties: a map of client PIDs to their current nickname (system wide), a map of chatroom names (string) to a list of clients registered to the chatroom (list of PIDs), and a map of chatroom names (string) to the corresponding chatroom PIDs. These properties make up the server's local state. It is should always be kept track of in a record of type `serv_st`. Its declaration is given below:

```
-record(serv_st, {nicks, registrations, chatrooms}).
```

`serv_st` consists of the following fields:

- **nicks**: a map from a client's PID as the key to the nickname (string) under which the client is currently registered as the value. It should be ensured that no two clients have the same nickname, and a client is not allowed to change its nickname to a nickname currently in use by another client. Look at the maps api in the Erlang docs to complete this. Additionally, nicknames should be stored as strings.
- **registrations**: a map from a chatroom's name (string) as the key to a list of the client processes' PIDs of clients registered in that chatroom. At a given time, the map may look a bit like this:

```
{ "#soccer" => [<0.0.1>, <0.0.3>, ...],
  "#swim_meet" => [<0.0.2>, <0.0.3>, <0.0.4>, ...],
  ... }
```

This map must be kept up to date as clients join and leave chatrooms.

- **chatrooms**: a map from a chatroom's name (string) as the key to the chatroom's corresponding PID as the value.

The server is responsible for spawning any chatrooms that a client requests to join if they don't exist, or telling a chatroom the PID of a client that wants to join it if that chatroom already exists. Whether the chatroom is newly spawned or already existent, the chatroom will be responsible, upon receipt of the new client info, of informing the client about itself.

For the purposes of this assignment, it can be expected that a chatroom will not crash (unless there are bugs in your code) and so there is no need for monitoring. It can also be assumed that once a chatroom exists, it is never destroyed for the running duration of the program.

2.1.3 Chatroom (`chatroom.erl`)

The chatroom has the following properties: a name (string), a map from a client's PID to a client's nickname (string) which represents clients registered to that chatroom, and a history of the chat since the beginning of that chatroom's life. These properties make up the chatroom's local state. It is should always be kept track of in a record of type `chat_st`. Its declaration is given below:

```
-record(chat_st, {name, registrations, history}).
```

`chat_st` consists of the following fields:

- **name**: the name (string) of the chatroom.
- **registrations**: a map from a client's PID to a client's nickname (string). The map represents all clients registered in that chatroom.
- **history**: chat history since the beginning of that chatroom. It should be represented as a list of tuples, where each tuple is {`Client_nickname, Message`}, both of which are strings.

The chatrooms will be responsible for propagating messages sent to it to all clients in the chatroom. The chatroom depends on the server to update it when a client joins or leaves a chatroom, or changes its nickname. Please note that messages (as in messages a user types to a chatroom) are sent directly to the chatroom, and are **not** routed through the server first. See Figure 1, connection D. This will be discussed in more detail below.

3 Message Passing

The following sections will discuss the sequence of messages passed from process to process given certain user input. We will use Figure 1 as a guide. The boxes in Figure 1 represent running processes. For example, in Figure 1 there are two GUI processes running, namely `GUI1` and `GUI2`. Letters A to D represent

communcation connections between processes.

When a process sends a message to another process along a certain connection, the letter denoting the communication connection will refer to the messaging connection letters in Figure 1, and the messaging connection letter will be in [square brackets].

As we will have to test all your code, it is important that you **stick to the described message passing protocols**, or your code **will** fail our test cases. Additionally, we will be listening for all the proper processes to be created, and for the correct processes to speak with each other in the specified portions of the programs. In short, don't get too creative and try to stick to the specs.

3.1 New client is initialized

When a new **client** is created, it automatically sends a message consisting of the tuple {`self(), connect, InitialState#cl_st.nick`} to the **server**. The **server** then stores that **client**'s PID and nickname in its state (see `server.erl:30`). This is already implemented.

3.2 `/join #chat`

1. The **GUI** will send the message {`request, self(), Ref, {join, ChatName}`} to the **client** [A].
2. The **client** checks in its `cl_st` record to see if it is already in the **chatroom**. If the **client** is already in the chatroom identified by `ChatName`, then the message {`result, self(), Ref, err`} [A] should be sent back to the **GUI**, and the following steps should be skipped. Otherwise, if the **client** *is not* in the chatroom, continue on to step 3.
3. Since the **client** is not currently in the **chatroom** identified by `ChatName`, the client should ask the server to join said **chatroom**. The **client** will send the message {`self(), Ref, join, ChatName`} [B] to the server.
4. Once the **server** receives the message from the **client**, the server needs to check if the chatroom exists yet. This can be done using the `chatrooms` element of the `serv_st` record. If the **chatroom** does not yet exist, the server must spawn the chatroom.
5. Next, the **server** should look up the **client**'s nickname from the **server**'s `serv_st` record.
6. Once either the existing **chatroom** PID is found or the new chatroom is spawned, the server must tell the **chatroom** that the **client** is joining the **chatroom**. To achieve this, the **server** will send the message {`self(), Ref, register, ClientPID, ClientNick`} [C] to the **chatroom**.
7. The **server** will then update its record of chatroom registrations to include the **client** in the list of clients registered to that chatroom.

8. Once the **chatroom** has received the message from the **server**, it will update its local record of registered **clients**. Then it will tell the **client** about itself by sending the following message to the **client**: {`self(), Ref, connect, State#chat_st.history`} [D] where `State` is the **chatroom**'s `chat_st`.

9. The **client** will receive the message, update its record of connected chatrooms, and send the message {`result, self(), Ref, History`} [A] back to the **GUI**, where `History` is the chatroom history received from the **chatroom** process.

10. The **GUI** code with which you have been provided will then handle that request and create the appropriate tab, populating it with the chat history.

3.3 `/leave` or `/leave #chat_name`

As far as the **client** is concerned, both of these commands are executed the same. In the case that the user enters the command `/leave`, the **GUI** simply fills in `#chat_name` by analyzing the active tab. Either way, the same message is sent off to the **client**, and you can treat them as equivalent.

1. **GUI** sends the message {`request, self(), Ref, {leave, ChatName}`} [A] to the **client**.

2. The **client** needs to check that it is in the **chatroom** with the name `ChatName`. If the chatroom is *not found* in the **client**'s list of connected chatrooms, then the **client** should send the message {`result, self(), Ref, err`} [A] to the **GUI**, and should skip steps 3 and on.

3. If the **chatroom** *is found*, then the **client** should send the message {`self(), Ref, leave, ChatName`} [B] to the **server**.

4. The **server** will lookup the **chatroom**'s PID from the **server**'s state `serv_st`.

5. The **server** will remove the **client** from its local record of chatroom registrations.

6. The **server** will send the message {`self(), Ref, unregister, ClientPID`} [C] to the **chatroom**.

   (a) the **chatroom** will remove the **client** from its record of registered clients.

7. The **server** will then send the message {`self(), Ref, ack_leave`} [B] to the **client**.

8. The **client** will then remove the **chatroom** from its list of chatrooms.

9. The **client** will then send the message {`result, self(), Ref, ok`} [A] back to the **GUI**.

10. The **GUI** will handle deleting the tab and such.

3.4 `/whoami`

1. The **GUI** sends the message {`request, self (), Ref, whoami`} [A] to the **client**.
2. The **client** will then send the message {`result, self(), Ref, Nickname`} [A] back to the **GUI**, where `Nickname` is the nickname found in its state `cl_st`.
3. The **GUI** will then take care of displaying the nickname in the *System* tab.

3.5 `/nick new_nickname`

1. The **GUI** will send a message {`request, self (), Ref, {nick, Nick}`} [A] to the **client**.
2. The **client** should check `Nick` against its current nickname. If the **client** finds that `Nick` is the same as current nickname, then the **client** should send a message to the **GUI** {`result, self(), Ref, err_same`} [A], and skip steps 3 and on.
3. If **client** finds that `Nick` *is not* the same as current nickname, then the **client** will send a message {`self(), Ref, nick, Nick`} [B] to the server.
4. The **server** first needs to check if the new nickname `Nick` is already used. If the nickname *is* already in use by another client, the **server** will send the message {`self(), Ref, err_nick_used`} [B] back to the **client**, and the **client** will send the message {`result, self(), Ref, err_nick_used`} [A] back to the **GUI**. If this is the case then skip steps 5 and on.
5. Since we now know that the new nickname `Nick` is free to be used, the **server** must update it's record of nicknames by pointing that **client**'s PID to the new nickname instead.
6. The **server** must now update all **chatroom**s to which the **client** belongs that the nickname of the user has changed, by sending each relevant **chatroom** the message {`self(), Ref, update_nick, ClientPID, NewNick`} [C] to the **chatroom**s.
7. The **server** will then send the message {`self(), Ref, ok_nick`} [B] to the **client**.
8. The **client** will then send the message {`result, self(), Ref, ok_nick`} [A] back to the **GUI**.
9. The **GUI** will take care of alerting the user to the change in the *System* tab.

3.6 `Send a message`

The send-a-message protocol needs to be broken into two contexts: The **client** who initializes the chat message, and the other **client**s in the chatroom to which the message is sent. The former of these will be quite similar to what you have seen before. However, pay special attention to the latter half of the protocol as it needs to use an API call that is otherwise not part of this assignment. This is necessary just to simplify the message passing for you, and to simplify the GUI application for us. The first client

(the one who sends the message) will be denoted **sending client**, and the clients who receive the message initiated by **sending client** will be denoted as **receiving client** or **receiving clients**.

3.6.1 Sending client

Any text entry that the user enters into the graphical interface such that the first word is not prefaced with a forward-slash (`/`) is recognized as an outgoing message. This message will be sent to the chatroom identified by the active tab. Obviously a message being sent to the *System* tab is not allowed as this is not a chatroom, but merely a place for all non-chat messages to be put. Fortunately, you do not need to handle this as the message will be denied by the GUI code.

The protocol will be as follows:

1. The **GUI** will send the message {`request, self(), Ref, {outgoing_msg, ChatName, Message}`} [A] to the **sending client**.
2. The **sending client** must look up the PID of the **chatroom** in its list of connected chats.
3. The **sending client** will then send the message {`self(), Ref, message, Message`} [D] to the **chatroom**.
4. The **chatroom** will then send back to the **sending client** the message {`self(), Ref, ack_msg`} [D] The **chatroom**'s behavior will be continued below, as it only relates to the receiving client(s).
5. The **sending client** will then send back the message {`result, self(), Ref, {msg_sent, St#cl_st.nick}`} [A] to the **GUI**.
6. The **GUI** will then take care about posting to the chat history for the appropriate tab.

3.6.2 Receiving client(s)

This bit of the protocol picks up after the **chatroom** receives the new message from the **sending client**.

1. The **chatroom** will send a message {`request, self(), Ref, {incoming_msg, CliNick, State#chat_st.name, Message}`} [D] to each **receiving client** registered to the **chatroom** *except for the* ***sending client!*** In other words, if the clients Beezle, Bub, and Blitzen are registered to the **chatroom**, and Bub sent the initial message to the **chatroom**, the chatroom will send this message to Beezle and Blitzen.
2. The **chatroom** will then append the new `Message` to its own chat history. It does so by appending {`CliNick,Message`} to the `history` field of the record holding its state .

3. When the **receiving client** receives the message outlined in step 2, it will make the following function call:
`gen_server:call(list_to_atom(State#cl_st.gui), {msg_to_GUI, ChatName, CliNick, Msg})`. See explanation below for information on what this is.

4. The **GUI** will then receive the message {`msg_to_GUI, ChatName, CliNick, Msg`} through that function call in step 3, and post the message to the appropriate chatroom tab.

Explanation for `gen_server:call(...)` function (if interested)

The gen_server [link] is a server that is used in the GUI implementation. Since the GUI doesn't have any open receive statements listening for incoming messages, we needed to use the gen_server as a way to contact the GUI. For further reading, follow the link above, although you will have no need for greater understanding in this assignment.

3.7 `/quit`

This section perhaps warrants the most detailed attention, as everything must be cleaned up properly. When a user quits, the **server** and all **chatroom**s must be properly updated, and the **client** and **GUI** must be properly spun down.

NOTE: if the GUI window does not close, BUT when you enter commands into the prompt and hit enter, nothing happens, do not worry. There is some issue we were having with the visual element of the GUI properly exiting, and it was frankly not worth our time to debug this as it does not affect the students' end of the assignment. If someone comes up with a fix, that's great and please let us know, but it is not your responsibility to fix this.

1. The **GUI** will send the message {`request, self(), Ref, quit`} [A] to the **client**.
2. The **client** will send the message {`self(), Ref, quit`} [B] to the **server**.
3. The **server** must clean up a bit.
   (a) Remove **client** from nicknames.
   (b) Tell each **chatroom** to which the **client** is registered that the **client** is leaving. To do so, send the message {`self(), Ref, unregister, ClientPID`} [C] to each **chatroom** in which the client is registered. Note that this is the same message as when a **client** asks to leave a **chatroom**, so this should be handled already.
   (c) Remove **client** from the **server**'s copy of all chat registrations.
4. The **server** must then send the message {`self(), Ref, ack_quit`} [B] to the **client**.
5. The **client** must then send the message {`self(), Ref, ack_quit`} [A] to the **GUI**.
6. The **client** must then cleanly exit. The `GUI` will exit cleanly upon receiving the message from the client (apart from in the situation discussed above).

3.8 Specifics of implementation

1. There is a makefile attached. To build the application, type`make run` into your terminal in the directory containing your code. This will also launch the Erlang interpreter for you.
2. The program will be started with `main:start().` This will spawn a server and 2 GUIs.
3. The caller can specify the number N of GUIs to be started initially with `main:start(N).`
4. The caller can run additional GUIs with `gui:start_gui().`

The following items should be followed very closely. Failure to do so will cause the program to fail test cases.

1. **GUI code should not be altered to make your code work**. We are testing your code as if the GUI code has not been altered.
2. When a client receives the string of the name of a chatroom, it includes the pound symbol (#) at the beginning of the chatroom name. Leave this in, and store this as part of the chatroom name.
3. Client's nicknames should be stored as strings, not converted to atoms.
4. When the server registers a new client to a chatroom, **append the new client to the front of the chatroom's registration list** in the **server** state.
5. We will not test the order in which items appear in maps i.e.

   ```
   {foo => ``food'', bar => ``barber''}
   ```

   will be tested the same as

   ```
   {bar => ``barber'', foo => ``food''}
   ```

   but you should not go about recreating the `maps` library. Just use erlang's library calls.
6. Do not alter the declaration of the records in `defs.h`. This includes ordering of its elements, renaming its elements, etc. Leave `defs.h` alone.
7. Note that you should not create new references with `make_ref()` anywhere in your code. The **GUI** code will take care of creating a unique reference ID for each sequence, and you should use this reference for the entirety of the messaging sequence. Creating new reference IDs will result in failing test cases.

Above all, ask questions when you are confused! This is a complex assignment with a lot of intricate details.

3.9 Usage trace with expected states

This is an outline of the usage of the program (with some assumptions made about initial client nickname), showing the states of all processes at the end. You can use this as a way to compare that your program does what is expected.

1. `main:start()` is called, spawning 2 GUIs (client 1 with nickname "client1", and client 2 with nickname "client2") and 1 server. The GUI for client 1 has name "gui1", and the GUI for client 2 has name "gui2"

2. client 1 issues command `/nick newclient1`:
   client 1 changes nickname to "newclient1"

3. client 1 issues command `/nick newclient1` again:
   client 1 tries to change nickname to "newclient1" again (and fails to do so)

4. client 1 issues command `/nick client2`:
   client 1 tries to change nickname to "client2" (and fails to do so)

5. client 1 issues command `/join #chat1`:
   client 1 joins chatroom 1 "#chat1"

6. client 1 issues command `hello` from the `#chat1`: tab
   client 1 sends message "hello" to chatroom 1

7. client 1 issues command `/nick newclient1_2`:
   client 1 changes nickname to "newclient1_2"

8. client 1 issues command `/join #chat2`:
   client 1 joins chatroom 2 "#chat2"

9. client 1 issues command `/nick newclient1_3`:
   client 1 changes nickname to "newclient1_3"

10. client 1 issues command `/leave #chat1`:
    client 1 leaves chatroom #chat1

11. client 1 issues command `/leave #notachatroom`:
    client 1 tries to leave some chatroom that does not exist or to which they are not connected (and fails to do so)

12. client 2 issues command `/join #chat1`:
    client 2 joins chatroom 1

13. client 2 issues command `/join #chat2`:
    client 2 joins chatroom 2

14. client 2 issues command `world` from the `#chat2` tab:
    client 2 sends message "world" to chatroom 2

15. client 1 issues command `/join #chat1`:
    client 1 rejoins chatroom 1

16. client 1 issues command `/join #chat1`:
    client 1 tries to join chatroom 1 again (and fails to do so)

The final states of all 5 processes will be as follows:

- Client 1:

```
{cl_st, "gui1","newclient1_3",
  #{"#chat1" => <chat 1 pid>,
    "#chat2" => <chat 2 pid>}}
```

- Client 2:

```
{cl_st, "gui2", "client2",
  #{"#chat1" => <chat 1 pid>,
    "#chat2" => <chat 2 pid>}}
```

- Chatroom 1:

```
{chat_st,"#chat1",
   #{<client 1 pid> => "newclient1_3",
     <client 2 pid> => "client2"},
   [{"newclient1","hello"}]}
```

- Chatroom 2:

```
{chat_st,"#chat2",
   #{<client 1 pid> => "newclient1_3",
     <client 2 pid> => "client2"},
    [{"client2","world"}]}
```

- Server:

```
{serv_st,
   #{<client 1 pid> => "newclient1_3",
     <client 2 pid> => "client2"},
   #{"#chat1" => [<client 1 pid>,<client 2 pid>],
     "#chat2" => [<client 2 pid>,<client 1 pid>]},
   #{"#chat1" => <chat 1 pid>,
     "#chat2" => <chat 2 pid>}}
```

4 Submission Instructions

Submit a file `tchat.zip` containing all the files listed below. The list below also indicates which files from the stub you should have left unmodified and which ones you should have completed with your own code. One submission per group.

File Name	**Do I have to modify this file?**
chatroom.erl	Yes
client.erl	Yes
defs.hrl	No
grm.yrl	No
gui.erl	No
lex.xrl	No
lexgrm.erl	No
main.erl	No
makefile	No
server.erl	Yes