Module minichat

Overview 

Welcome to Single File MiniChat

Version: 0.1.0

Authors: Pascal Chapier (pascalchap@gmail.com).

Build

Clone the Git repositery, compile the unique file "minichat" in a valid erlang environment. This version was tested with Erlang/OTP 19, but it should work with earlier version.

Structure

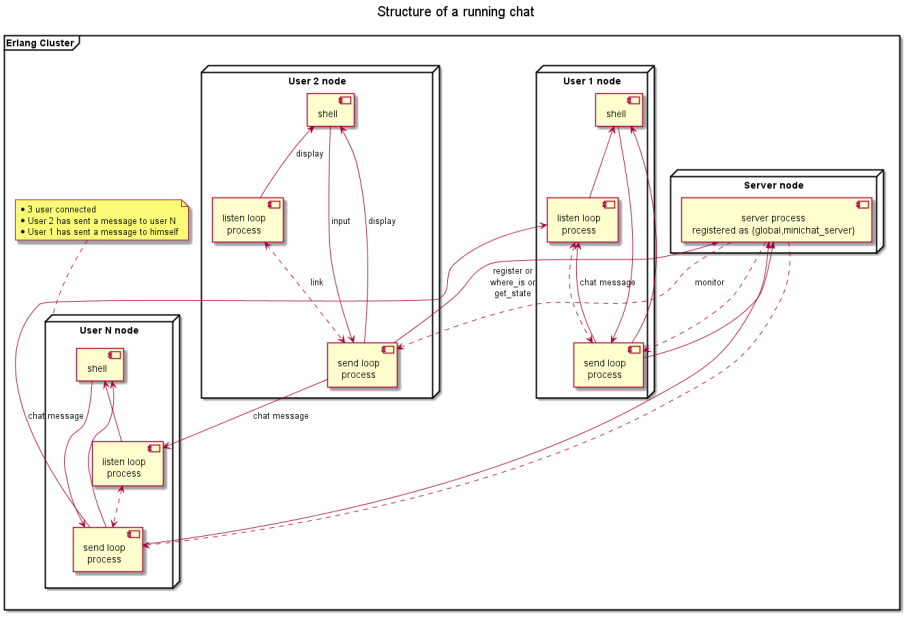
Although the code is in a single file, this chat uses a bunch of processes to work:

A server, registered as {global,minichat_server}, is used to solve the User addresses (get the user listen loop pid)

For each user, running on a dedicated node, a main loop is responsible to get the user input, react accordingly, and display some info in the shell panel(at least the prompt :o). The process is called **send loop** but it is not registered

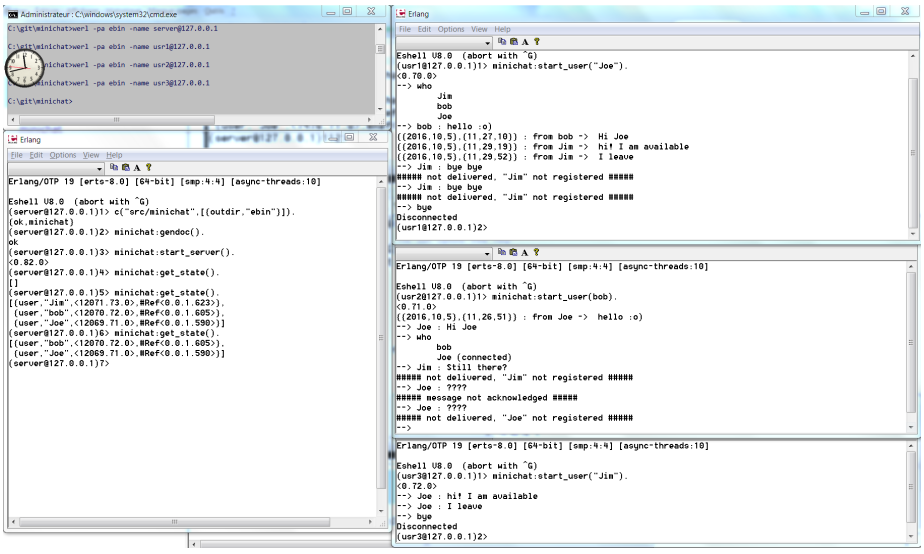
For each user as second loop is running, responsible to listen all the messages comming from other users and display them in the shell panel. The process is called **listen loop**, still not registered

The following diagram shows all the processes and the different relationship between them in a chat system involving 3 users.



Playing with the chat

Following a screen capture of the minichat in action.



Generated by EDoc, Oct 5 2016, 12:08:19.

Overview 

Module minichat

A simple, one file chat system working on an Erlang cluster.

Description

A simple, one file chat system working on an Erlang cluster

Data Types

connected_user_list()

```
connected_user_list() = [{string(), pid()}]
```

user()

```
user() = #user{name = string(), pid = pid(), monitor = reference()}
```

Function Index

start_server/0	Start the “name server”.
register_me/2	Register a user and its listen loop pid.
where_is/1	Find the user listen loop pid of a given user name.
get_state/0	Get the user record list from the server.
stop_server/0	Stop the server.
server_init/0	Server initialization.
server_loop/1*	*Private* Server loop.
do_register/4*	*Private* Function to register a new user.
unregister/2*	*Private* Function to un-register a user whose process died.
search/3*	*Private* Get the user listen loop pid given his name.
start_user/1	Start the user process with a default server node.
start_user/2	Start the user process with given server node.
user_init/1	Initialize the user processes.
listen_loop/0	Listen loop.
send_loop/3*	*Private* Send loop.
send_message/3*	*Private* First step of the send message operation responsible to extract the Recipent nam
send_message/4*	*Private* Second step of the send message operation responsible to retreive the Recipient li
do_send/4*	*Private* Last step of the send message operation responsible to send the message.
split/1*	*Private* Extract the Recipient name from a string.
show_people/1*	*Private* Request the list of the registered users.
gendoc/0	Generate the documentation.

Function Details

start_server/0

```
start_server() -> pid()
```

Start the “name server”. This function must be executed prior to any other one on the server node. It is in charge of keeping a list of all the connected users. Each user info is stored in a record containing its name, the pid of the user listen loop and a monitor reference used to manage the death or deconnection of the users.

register_me/2

```
register_me(Name::string() | atom(), Pid::pid()) -> ok | {error,
already_exist}
```

Register a user and its listen loop pid. This function is an interface provided to the user in order to register its name and the pid of its listen loop. Doing

so, he allows other users to get its liten loop pid which will be used later for communication. The Name must be unique.

Note the this function, as all other server interfaces, uses global:send/2 to send messages to the server rthar than the operator !. It is because the server is registered "globally" in order to "find" it easily from any node of the cluster. On the other hand, in the rest of the code, we can use the operator ! since we know the pid of the processes we want to reach.

where_is/1

```
where_is(Name::string()) -> {ok, pid()} | {error, not_registered}
```

Find the user listen loop pid of a given user name. This function is an interface provided to the user in order to retreive the listen loop pid of a given user.

get_state/0

```
get_state() -> [user()]
```

Get the user record list from the server. This function is an interface provided to the user to get the list of all connected users.

stop_server/0

```
stop_server() -> pid()
```

Stop the server.

server_init/0

```
server_init() -> ok
```

Server initialization. This fuction is exported to allow the usage of spwan/3 to start the server process. It register the server with the name 'minichat_server' using the global module. Thus the registration is available for all the nodes of the cluster.

Then it calls the server_loop with an empty list of users as initial state.

server_loop/1 *

```
server_loop(Users::[user()]) -> ok
```

Private Server loop. This function is in charge to manage the incomming messages to the server. It maintains a list of user records who are connected to the chat system. It add the new users, monitor their processes and remove them from the list when the user process dies. It ignores any unexpected (badly formatted) message.

do_register/4 *

```
do_register(Users::[user()], Name::string(), Pid::pid(), From::pid())
-> ok | {error, already_exist}
```

Private Function to register a new user. Check if the Name of the registering user is not already in use,

if not, start to monitor the user process (send loop) and add the user record to the existing list and send back the message ok.

if yes send back an error message.

Note that the pid used for the communication is the is the user send loop, the one which is waiting for an answer, while the pid stored in the user record is the listen loop's one which will used by other users to send messages.

unregister/2 *

```
unregister(MonitorRef::reference(), Users::[user()]) -> [user()]
```

Private Function to un-register a user whose process died.

search/3 *

```
search(Users::[user()], Name::string(), From::pid()) -> {ok,
pid()} | {error, not_registered}
```

Private Get the user listen loop pid given his name.

start_user/1

```
start_user(Name::string()) -> pid()
```

Start the user process with a default server node. The node server is defined as 'server@127.0.0.1'. Convenient to make the test without getting annoyed by the firewall configuration.

start_user/2

```
start_user(Name::string(), ServerNode::atom()) -> pid()
```

Start the user process with given server node. the function connects the local node to the server node and waits for the node synchronization before spawning the user process. It is important since the first job that the user process will perform is to register itself to the server.

user_init/1

```
user_init(Name::string()) -> ok | {error, already_exist}
```

Initialize the user processes. First, it starts the listen loop, an then it registers itself to the server.

if it works, call the send loop. At this point the user is made of 2 processes linked together : the send and the listen loops. If any of them dies, the other will die also. When the user closes its process (normal termination) the code must take care to ends both processes.

if it fails, stop the listen loop and return an error message.

listen_loop/0

```
listen_loop() -> ok
```

Listen loop. A simple receive bloc listening any message.

If it is a well formed message from another user, it prints to the shell the date an time, the emitter name and then the message text. Then it aknowledges the message

If it is the stop message, it ends the loop.

It ignores all other messages, simply removing them from the mailbox.

The listening activity run in a separate process, so it is always available to display incoming messages, independently of the writting activity of the user. Nevertheless both processes share the shell panel to display their results, The basic test I made worked correctly, even if an incomming message is displayed while the user is writting.

send_loop/3 *

```
send_loop(Name::string(), Me::pid(), Connected::connected_user_list())
-> ok
```

Private Send loop. The main user process is in charge to get the input from the user , analyze it and make an action accordingly. 3 messages are supported, the other ones are ignored:

"bye" : leave the chat

"who" : get a list of the reachable users

"Recipient : Text" : to send the message "Text" to the user "Recipient" Recipient is any string that do not contain a : character. So a user who have chosen the name "Smart:Joe" is not reachable while "Smart Joe" is correct.

The sending loop stores 3 information:

Name : The user name is used to tag the messages so the recipent will know who wrote each of the received messages.

Me : the pid of the listen loop, in order to stop it whe the user leaves the chat. As far as the listen loop is linked to the send loop, it was also possible to exit the prcess using a reason different than normal, but I don't like to use error mechanism for a normal behavior.

Connected : a list of pair {Name,Pid} to avoid permanent accesses to the server to solve the Recipient address. A consequence is that if one user is disconnected for a while, the first message won't be delivered (The system don't propagate the user disparition).

Sending the message needs some steps, the execution is delegate to the helper function send message who is in charge to send the message if possible and update the pair list of [user,pid].

send_message/3 *

```
send_message(Message::string(), Sender::string(), Connected::connected_user_list())
-> connected_user_list()
```

Private First step of the send message operation responsible to extract the Recipent name from the user input.

send_message/4 *

```
send_message(Recipient::string(), Text::string(), Sender::string(),
Connected::connected_user_list()) -> connected_user_list()
```

Private Second step of the send message operation responsible to retreive the Recipient listen loop. If the Recipent is not connected yet to this user, it requests its pid to the server and adds it to the connected list

do_send/4 *

```
do_send(To::pid(), Text::string(), Sender::string(), Connected::connected_user_list())
-> connected_user_list()
```

Private Last step of the send message operation responsible to send the message. If the message is not acknowledged within 2 seconds, it removes the recipein from the connected user list.

split/1 *

```
split(Message::string()) -> {string(), string()} | {error, string()}
```

Private Extract the Recipient name from a string. It first splits the string into pieces using the charater : as separator. Then it constructs a tuple made of the first token (without leading and trailing blanks) and the rest of the input string (if there were multiple : charaters, the message is rebuilt).

show_people/1 *

```
show_people(Connected::connected_user_list()) -> ok
```

Private Request the list of the registered users. When the list is received, the function extract all the names and check if they are in the connected list and print the information accordingly.

gendoc/0

```
gendoc() -> ok
```

Generate the documentation

Overview

Generated by EDoc, Oct 5 2016, 12:08:19.