Ermis

SOEN6411: A Simple Chat Server in Erlang
Presented to Dr. Constantinides

Simon Symeonidis
5887887

November 27, 2013

Contents

List of Figures

Listings

::Section 1::

Summary

This is my coursework for SOEN6411, on the *Erlang* programming language. Inside this document we discus the different powers of the programming language, the flexibilities, and other shortcommings.

For this case study the project I tackled was writing a very simplified version of an IRC server. By this we wish to allow different clients connect to a server and send messages. It follows that the server should allow facilities to broadcast messages to the clients.

Also I implemented a simple, and probably naive way of authenticating users, following a Token Based approach (the same way that Http servers with application understand authentication). This looks into a functionalities to produce a Key that is shared to authorized users, using *salt* and the *SHA* hashing method. This Key is later bound to the priviledged requests of the user to the server, so that the server can make sure that that issuing user is allowed to perform the augmented actions.

1.1 The name 'Ermis'

Is from 'Hermes' the messenger of the gods. Since in Erlang there is message based concurrency, I personally believe it's an adequate name for the project - not only due to the approach of the language's problem domain, but also that of this particular application we wish to implement.

1.2 Resources

I mainly used two very useful resources to get everything done in this case study. One resource is Fred Hébert's book [3], and the other was Erlang's online manual [2]. Joe Armstrong's Dissertation [1] also was a good read though I did not get to finish it.

::Section 2::

Introduction

Erlang is an interesting language as it has been notably the one which has given me the least trouble to program networked applications. On other programming languages that I have used, the traditional approach was to open up a socket at a port, and code from scratch the protocols on each side - sending and receiving. Erlang however abstracts this tedious process with specific language constructs. Erlang also comes with OTP *(Open Telecommunication Platform)*, which helps in the development of distributed applications.

Another distinguishable feature unseen in any other programming languages I have used, is that Erlang suppports *Hot Code Loading*, which allows running systems to update themselves *while running, on production configurations.* This opens a lot of interesting possibilities on the management and behavior of such applications.

2.1 Problem Statement

For this project, we'll implement a small and naive implementation of an IRC server. We will also write our own clients that will interface to the service, and use it.

Ultimately we want to represent the system in the following way, as shown in Figure 1. Nodes A, B, C connect to the server, and via the server they are able to talk to each other. That is, if A sends a message, it is relayed to the rest of the connected nodes - B, C in this case.

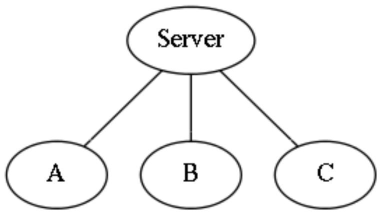

Figure 1: The System as Nodes

To increase the complexity for this project, we wish to include authentication, so that we can perform priviledged actions such as 'kick', and 'mute'. This will also require an authentication method 'auth'. The authentication method we wish to pursue, is simple 'token-based' authentication. The token to be generated is generated this way: the server stores a secret passphrase, and the user sends this passphrase via text. The server sees that the text is indeed the secret passphrase. On success, the server returns the current nano time, concatenates it to the secret passphrase (salt), and then hashes it using the *SHA* hashing algorithm. The server stores the Key in an ETS table, and sends the key back to the user - this is the shared secret between the two.

::Section 3::

Language Exploration

In this section we highlight the different parts of the language that we find unique, or distinguishing, in respect to others. We outline these differences atomically, and later on show how they tied together in order to help build the chat server.

3.1 Types

There exist some standard types in *Erlang* that are used in order to describe and solve problems. On a general note, not too many surprises are found here, except for the binary notation of data types. We list some data types.

Lists Lists are very similar to the lists in *Prolog*, as are the operations you can perform. To express a list you ensnare the wanted elements between square brackets. Here is an example: [1, 2, 3, 4, 5]. Lists can contain any other type as well.

Strings Strings are expressed as lists of integers in Erlang. Therefore we can use list operations to manipulate strings if we wish. For example getting the 'head' of a string would return the first character, and 'tail' would return the rest. So, for example, these two lists are equivalent, and if we checked for equality in an *Erlang* application, the result would evaluate to *true*: [97, 98, 99] =:= "abc".

Atoms Erlang supports atoms, very similar to those of Prolog. However [3] stresses that they should not be dynamically created, as atoms are limited. Atoms are useful when implementing protocols for communicating nodes as they can be used to identify the type of requests between said nodes. Atoms are expressed by typing in a token that starts with a lowercase alphabetic character. Examples are: *cat*, *bluescreen123*, *kittens.*

Tuples Erlang supports tuples. A tuple can contain any other data structure, as well as other tuples inside itself. To express tuples, we ensnare elements between curly brackets. Here is an example: {a,b,c,[1,2,3]}.

Bit Syntax / Binaries Special notation that allows the users of the programming language to express bit-strings exists. To express bitstrings, we ensnare elements with two angled brackets. Here is an example: $\langle\langle 12, 34, 51\rangle\rangle$.

Combinations As previously stated, we can combine different data types in order to express solutions to problems. For example, the following combines all aforementioned types:

$$[\langle\langle 1,2,3\rangle\rangle, \{"fred", [1, kitten, 3]\}, 3]$$

3.2 ETS Tables

ETS (Erlang Term Storage) tables are an extremely useful feature in the language, but results in side effects. But in my opinion sometimes side effects are desirable for improving the usability of programming languages - Haskell in this aspect is much more pure, but as a result a little harder to program in. And in this respect, ETS tables provide for a better programming experience with *Erlang.*

Tables work as simple key value stores, and one can store any data structure inside. For example we could store a key, with a value of an array. We can also store a key, with a value of a 3-tuple, who's first element is an atom, the second a list, and the last a 2-tuple of atoms.

What makes this extremely favorable into whole new level is that we can later use Erlang's powerful pattern matching to find and restore keys. We will demonstrate this as an example. But first let us first investigate how we create an ETS, as a simple *set* (Listing 1).

```
% @doc create a table
make_table() ->
  TableId = ets:new(mytable, [set,public,named_table]).
```

Listing 1: Creating a simple ETS table as a Set

Key-Value pairs in ETS tables in Erlang, are stored as a 2-tuple, like {key, value}. Listing 2 demonstrates this.

```
% inserting a pair inside the table
% return true on successful insert
ets:insert(TableId, {mykey, ["puppies", "kittens", "ferrets"]}).
```

Listing 2: Inserting a value in the ETS table

And now we can lookup the key with the method shown in Listing 3.

```
ets:lookup(TableId, mykey).
% output: [{mykey,["puppies","kittens","ferrets"]}]
```

Listing 3: Lookup a value in the ETS table

3.3 Erlang's Pattern Matching

We look at how we can express powerful patterns in Erlang, that can produce concise code by matching. This results in easier readability for us, the programmers. Pattern matching in Erlang might remind you of Prolog, since its roots were located there.

Now if we store different values using different data structures, and we're not sure at any given time what to expect from a lookup, we could use Erlang's pattern matching in order to quickly decide what to do on each case. As you can see in Listing 4 we can use this powerful form of expression, with the combination of guards.

```
-module(pattern-matching).
-author("lethaljellybean@gmail.com").
-export([main/1]).

main(_ARGS) ->
  TableId = ets:new(mytable, [public,set,named_table]),

  ets:insert(TableId, {mykey, ["ferrets", "puppies", "kittens"]}),
  io:format("~s~n", [what_do(TableId)]),

  ets:insert(TableId, {mykey, {"lasagna", ["cheese", "yummy", "so nice"]}}),
  io:format("~s~n", [what_do(TableId)]),

  ets:insert(TableId, {mykey, value}),
  io:format("~s~n", [what_do(TableId)]),

  ets:insert(TableId, {mykey, {one, two, three}}),
  io:format("~s~n", [what_do(TableId)]),

  ets:insert(TableId, {mykey, {one, two, three, four}}),
  io:format("~s~n", [what_do(TableId)]).

what_do(TableId) ->
```

```
  case ets:lookup(TableId, mykey) of
  [] ->
    "nothing found";

  [{_, Val}] when is_atom(Val) ->
    "You have a value of only an atom!";

  [{_, {_, _}}] ->
    "You a tuple of two!";

  [{_, {_, _, _}}] ->
    "You a tuple of three!";

  [{_, Val}] when is_list(Val) ->
    "You have an array!";

  _ ->
    "Unmatchable."
  end.

%% Output
%%
%% You have an array!
%% You a tuple of two!
%% You have a value of only an atom!
%% You a tuple of three!
%% Unmatchable.
```

Listing 4: Pattern Matching

3.4 Recursion

Recursion exists in Erlang and the way it is invoked is nothing out of the ordinary. Here is a simple function that calculates factorials. Listing 5 shows us this.

```
-module(defacto).
-author("lethaljellybean@gmail.com").
-compile(export_all).

% Base case
factorial(0) -> 1;
% Recursive case
factorial(N) ->
  N * factorial(N - 1).
```

Listing 5: Factorial Implementation in Erlang

3.5 Spawning Processes

In order to spawn a process, we need to use **spawn**. This function takes in three arguments.

1. the module where the function of interest exists
2. the function name as an atom
3. the argument list for that function, if the function is parameterized

So in that respect, Listing 6, demonstrates how to spawn a process with the given function. The **?MODULE** is a value that is replaced by the current module name. If for example the module was renamed in the future, this application would still run, as opposed to an example that would use the hard coded name of said module.

```
-module(spawnone).
-author("lethaljellybean@gmail.com").
-compile(export_all).

main(_ARGS) ->
  spawn(?MODULE, func1, []),
  spawn(?MODULE, func2, [one,two]),
  spawn(?MODULE, func3, []),
  timer:sleep(5000),
  io:format("program end.~n").

func1() ->
  timer:sleep(2000),
  io:format("function terminates!!~n").

func2(Arg1, Arg2) ->
  timer:sleep(3000),
  io:format("Another one bites the dust!~n").

func3() ->
  timer:sleep(2500),
  io:format("I am in between!~n").
```

Listing 6: Spawning in Erlang

The output is shown in Listing 7.

```
1> c(spawnone).
spawnone.erl:16: Warning: variable 'Arg1' is unused
spawnone.erl:16: Warning: variable 'Arg2' is unused
{ok,spawnone}
2> spawnone:main([]).
function terminates!!
I am in between!
Another one bites the dust!
program end.
ok
```

Listing 7: Output of spawned processes

3.6 Using Receive

Another excellent feature of Erlang, is how message passing is performed. We are given tools in the language to approach this implementation detail with ease. Think of a receive block as a **case of** block. We demonstrate such code in Listing 8, inside the **rcvloop** function. The output may be observed in Listing 9. To send messages to a process that is listening, we use the exclamation '!' mark, place the listening process id to the left of it and the message on the right.

Notice that inside the receive block, we call the function recursively. This is to keep the process listening. If this is omitted, like it is in the case of 'sleep', then the process stops listening, and terminates.

```erlang
-module(rcvtest).
-author("lelthaljellybean@gmail.com").
-compile(export_all).

main(_ARGS) ->
  io:format("Spawning process...~n"),
  MyProcess = spawn(?MODULE, rcvloop, []),

  io:format("Going to start sending stuff now...~n"),
  MyProcess ! {self(), coffee},
  MyProcess ! {self(), coffee},
  MyProcess ! {self(), tuna},
  MyProcess ! {self(), {pizza, burger}},
  MyProcess ! {self(), imhungry},
  MyProcess ! {self(), [mousaka, beans, bread]},
  MyProcess ! {self(), kitten},
  io:format("Terminating hungry process...~n"),
  MyProcess ! {self(), sleep},

  io:format("Received messages : ~p~n",
    [erlang:process_info(self(), messages)]).

rcvloop() ->
  receive
  {Pid, tuna}            ->
    Pid ! "Tuna!? OM NOM NOM",
    rcvloop();
  {Pid, coffee}          ->
    Pid ! "YES! COFFEE! GULP GULP GULP",
    rcvloop();
  {Pid, [mousaka|_]}     ->
    Pid ! "Mousaka always comes first.",
    rcvloop();
  {Pid, {pizza, burger}} ->
    Pid ! "PROGRAMMING POWER FOR REELZ",
    rcvloop();
  {Pid, imhungry}        ->
    Pid ! {here, have, some, dolmades},
    rcvloop();
  {Pid, sleep}           ->
    Pid ! "goodbye";
  {Pid, _}               ->
    Pid ! "I can't eat that.",
    rcvloop()
  end.
```

Listing 8: Receive Block Example

Inside the *main* function, you might notice that we send the tuples in the form of {self(), _}. The reason we do this is because we want to supply the process id of the process that is sending the messages to *rcvloop*, so that it can send back the required results (Eg: when sending the 'tuna' atom, we expect to receive the string "Tuna!? OM NOM NOM").

```
11> c(rcvtest).
{ok,rcvtest}
12> rcvtest:main([]).
Spawning process...
Going to start sending stuff now...
Terminating hungry process...
Received messages : {messages,["YES! COFFEE! GULP GULP GULP",
                               "YES! COFFEE! GULP GULP GULP",
                                "Tuna!? OM NOM NOM",
                                "PROGRAMMING POWER FOR REELZ",
                                {here,have,some,dolmades},
                                "Mousaka always comes first.",
                                "I can't eat that."]}
```

Listing 9: Receiving Messages (After Sending Them)

3.7 Hot Code Loading

A notable aspect of *Erlang*, is that it provides us with a feature called **hot code loading**: an application that is running, written in *Erlang* has the possibility of updating itself, whilst running, with no downtime. This is due to one of the use cases of the language that required it to provide mechanisms for services that would never go down - something that was required in telephony.

::Section 4::

Dissecting the Problem

In this section we will briefly explain how we decompose the problem. We talk about smaller uncertain parts of the software, and combine everything for a solution in the end.

4.1 Printing While Getting Input

This is somewhat of a trivial aspect, but is required in order to provide a nice interface in the client application. The client needs to read input from the user, but print the messages on the standard output as soon as they are received. Other applications usually achieve this by the use of threads. In *Erlang* we are able to use *spawn* in order to achieve this. Listing 10 is the main entry point of the *Ermis* client. We spawn the process, and store its process id to a variable which is later passed to the main loop of the client.

```erlang
% @doc main entry point for application. Prompts the user for
%   information about nnickname and where to connect to.
main(_ARGS) ->
  Nick       = rmn(io:get_line("Enter nickname : ")),
  ServerName = rmn(io:get_line("Enter server locations : ")),
  NickAtom   = list_to_atom(Nick ++ "@localhost"),
  Location   = list_to_atom(ServerName),
  TableId    = ets:new(mytable, [set,public,named_table]),
  Receiver   = spawn(ermisclient, preceiver, [TableId]),
  net_kernel:start([NickAtom, shortnames]),
  io:format("Trying to connect to : ~p...~n", [Location]),
  case net_kernel:connect(Location) of
  true ->
    register(shell, self()),
    {shell, Location} ! {Nick ++ " joined the server!", node(), Receiver},
    program_loop(Location,Receiver,TableId);
  _    -> io:format("Problem connecting... ~p~n")
  end.
```

Listing 10: Printing Received Messages Asynchronously

Listing 11 shows the process that is spawned in order to print messages to the standard console as soon as the messages are received. The first pattern matching is to see if the server is sending back the authentication token that is later discussed in section 4.2. The next match fires if whatever is sent back is not an authentication success, and therefore prints the results on the console.

```erlang
% @doc individual process that runs in the background, and prints a message
%   whenever someone connects to the server / is kicked / says something
preceiver(TableId) ->
  receive
  {token, Token} ->
    io:format("Authenticated with key ~p~n", [Token]),
    ets:insert(TableId, {key, Token}),
    preceiver(TableId);
  Rcv ->
    io:format("~s~n", [Rcv]),
    preceiver(TableId)
  end.
```

Listing 11: Receiving Loop of the Ermis Client

4.2 Generating a Salt

There are probably many more sophisticated ways to create salts for hashes in enterprise applications. However due to the simplicity of this application, and required clarity for the demonstration, we implement something simpler. We do this by taking the nano-time by calling the *now()* function in Erlang, converting it to a string, and concatenating it to the passphrase that we define in the module. Listing 12 demonstrates how we use pattern matching, a lambda for string conversion (due to the use of *fun(X)*), and concatenation to return everything as one element.

```
% @doc to be used with the crypto stuff
salt() ->
  {A, B, C } = now(),
  [K, L, M] = lists:map(fun(X) -> integer_to_list(X) end, [A, B, C]),
  K ++ L ++ M.
```

Listing 12: Generatin Salt

When the user tries to authenticate, a request is sent to the server in the receive block of Listing 13, and a check is made to see if the passphrase is correct. If this is indeed the case, the *crypto* module of erlang is used to hash the passphrase with the salt, and is sent back to the client.

```
% @doc main serving function
serve(RcvList, AuthKey, TableId) ->
  receive
  {authenticate, Usr, Other, Pass} ->
    io:format("Authentication request: ~s~n", [Pass]),
    SanitizedPass = rmn(Pass),
    case SanitizedPass =:= ?SECRET of
    true ->
      Key = crypto:hash(sha, SanitizedPass ++ salt()),
      % We send back the key on successful auth
      Other ! {token, Key},
      ets:insert(TableId, {key, Key})
    end,
    serve(RcvList,Key,TableId);

    ...
```

Listing 13: Ermis Server

4.3 Priviledged Behavior

Once the server has received an authentication request, it generates a token, sends it to the user, and remembers it by storing it in an ETS table (section 4.2 talks about this). When the user wishes to perform an action that requires priviledges, the user must send the command, along with the token in order for the server to accept this behavior. Listing 14 shows how the request is sent by the user, and Listing 15 shows how the server handles the mute request.

```
% @doc Main loop of the application
program_loop(ServerLoc,Receiver,TableId) ->
  Cmd = rmn(io:get_line("> ")),
  [H | _] = re:split(Cmd, "\s+", [{return,list}]),
  case H of
  "/kick" ->
    WhoToKick = rmn(io:get_line("which user? : ")),
    {shell, ServerLoc} ! {kick, node(), Receiver, get_key(TableId), WhoToKick}
        ,
    program_loop(ServerLoc,Receiver,TableId);
  "/mute" ->
    ToMute   = rmn(io:get_line("User to mute: ")),
    ToMuteAt = list_to_atom(ToMute),
    {shell, ServerLoc} ! {mute, node(), Receiver, get_key(TableId), ToMuteAt},
    program_loop(ServerLoc,Receiver,TableId);
```

Listing 14: Ermis Client Mute

```
% @doc main serving function
serve(RcvList, AuthKey, TableId) ->
  receive

  ...

  {mute, User, Other, Key, UserToMute} ->
    io:format("request to mute user : ~p~n", [UserToMute]),
    case Key =:= get_key(TableId) of
      true ->
        io:format("authenticated - muting user ~s~n", [UserToMute]),
        Ret = ets:lookup(TableId, muted),
        case Ret of
        [] ->
          ets:insert(TableId,{muted,[UserToMute]}),
          serve(RcvList,Key,TableId);
        _ ->
          [{_,Prev}|_] = ets:lookup(TableId, muted),
          case lists:member(UserToMute, Prev) of
          false ->
            NewList = [UserToMute|Prev],
            ets:insert(TableId, {muted,NewList})
          end,
          serve(RcvList,Key,TableId)
        end;
      false ->
        io:format("You are not authenticated to do that~n"),
        serve(RcvList,Key,TableId)
    end,
    serve(RcvList,Key,TableId);

  ...
```

Listing 15: Ermis Server Mute

5 Final product

Here the listings 16, and 17 are the client and server respectively, which were implemented with the given specifications.

```
-module(ermisclient).
-compile(export_all).
-author("lethaljellybean@gmail.com").

% @author
%   Simon Symeonidis
%   5887887
% @doc
%   This is a small chatting application using distributed erlang, that was
%   prepared as a small case study for the comparative programming language
%   course (SOEN 6411), and presented to Dr. Constantinides.

% @doc main entry point for application. Prompts the user for
%   information about nnickname and where to connect to.
main(_ARGS) ->
  Nick       = rmn(io:get_line("Enter nickname : ")),
  ServerName = rmn(io:get_line("Enter server locations : ")),
  NickAtom   = list_to_atom(Nick ++ "@localhost"),
  Location   = list_to_atom(ServerName),
  TableId    = ets:new(mytable, [set,public,named_table]),
  Receiver   = spawn(ermisclient, preceiver, [TableId]),
  net_kernel:start([NickAtom, shortnames]),
  io:format("Trying to connect to : ~p...~n", [Location]),
  case net_kernel:connect(Location) of
  true ->
    register(shell, self()),
    {shell, Location} ! {Nick ++ " joined the server!", node(), Receiver},
    program_loop(Location,Receiver,TableId);
  _    -> io:format("Problem connecting... ~p~n")
  end.

% @doc helper func to remove newline remove newline
rmn(Str) ->
  re:replace(Str, "\n", "", [global,{return,list}]).

% @doc Main loop of the application
program_loop(ServerLoc,Receiver,TableId) ->
  Cmd = rmn(io:get_line("> ")),
  [H | _] = re:split(Cmd, "\s+", [{return,list}]),
  case H of
  "/kick" ->
    WhoToKick = rmn(io:get_line("which user? : ")),
    {shell, ServerLoc} ! {kick, node(), Receiver, get_key(TableId), WhoToKick}
        ,
    program_loop(ServerLoc,Receiver,TableId);
  "/mute" ->
    ToMute   = rmn(io:get_line("User to mute: ")),
    ToMuteAt = list_to_atom(ToMute),
    {shell, ServerLoc} ! {mute, node(), Receiver, get_key(TableId), ToMuteAt},
    program_loop(ServerLoc,Receiver,TableId);
  "/auth" ->
    io:format("trying to authenticate ... ~n"),
    Passphrase = rmn(io:get_line("Passphrase : ")),
    {shell, ServerLoc} ! {authenticate, node(), Receiver, Passphrase},
    program_loop(ServerLoc,Receiver,TableId);
  "/names" ->
    program_loop(ServerLoc,Receiver,TableId);
  "/quit" ->
    {shell, ServerLoc} ! {disconnect, node(), Receiver},
    erlang:disconnect_node(node()),
    io:format("bye.~n");
  _ -> % Saying stuff here
    {shell, ServerLoc} ! {Cmd, node(), Receiver},
    program_loop(ServerLoc,Receiver,TableId)
  end.

% @doc individual process that runs in the background, and prints a message
%   whenever someone connects to the server / is kicked / says something
```

```
preceiver(TableId) ->
  receive
  {token, Token} ->
    io:format("Authenticated with key ~p~n", [Token]),
    ets:insert(TableId, {key, Token}),
    preceiver(TableId);
  Rcv ->
    io:format("~s~n", [Rcv]),
    preceiver(TableId)
  end.

% @doc get the token that was given from the server
get_key(TableId) ->
  case ets:lookup(TableId, key) of
  [] -> nil;
  [{_, Token}|_] -> Token
  end.
```

Listing 16: Ermis Client

```
-module(ermisserver).
-export([main/1]).
-author("lethaljellybean@gmail.com").

-define(SECRET, "1337sp33k").

% @author
%   Simon Symeonidis
%   5887887
% @doc
%   This is a small chatting application using distributed erlang, that was
%   prepared as a small case study for the comparative programming language
%   course (SOEN 6411), and presented to Dr. Constantinides.
main(_ARGS) ->
  SrvAt = ermis@localhost,
  io:format("Start server at ~p...~n", [SrvAt]),
  net_kernel:start([SrvAt, shortnames]),
  register(shell, self()),
  TableId = ets:new(mytable, [public,set,named_table]),
  ets:insert(TableId, {muted,[]}),
  ets:insert(TableId, {key, crypto:hash(sha, salt())}),
  serve(TableId).

% @doc interface function for the backend function
serve(TableId) -> serve([], noadmin, TableId).

% @doc main serving function
serve(RcvList, AuthKey, TableId) ->
  receive
  {authenticate, Usr, Other, Pass} ->
    io:format("Authentication request: ~s~n", [Pass]),
    SanitizedPass = rmn(Pass),
    case SanitizedPass =:= ?SECRET of
    true ->
      Key = crypto:hash(sha, SanitizedPass ++ salt()),
      % We send back the key on successful auth
      Other ! {token, Key},
      ets:insert(TableId, {key, Key})
    end,
    serve(RcvList,Key,TableId);
  {kick, Usr, Other, Key, UserToKick} ->
```

```
    case Key =:= get_key(TableId) of
      true ->
        io:format("authenticated - kicking user ~s~n", [UserToKick]),
        UsrToKickAt = list_to_atom(UserToKick),
        erlang:disconnect_node(UsrToKickAt);
      false ->
        io:format("You are not authenticated to do that~n")
     end,
     serve(RcvList,Key,TableId);
  {mute, User, Other, Key, UserToMute} ->
    io:format("request to mute user : ~p~n", [UserToMute]),
    case Key =:= get_key(TableId) of
      true ->
        io:format("authenticated - muting user ~s~n", [UserToMute]),
        Ret = ets:lookup(TableId, muted),
        case Ret of
        [] ->
          ets:insert(TableId,{muted,[UserToMute]}),
          serve(RcvList,Key,TableId);
        _ ->
          [{_,Prev}|_] = ets:lookup(TableId, muted),
          case lists:member(UserToMute, Prev) of
          false ->
            NewList = [UserToMute|Prev],
            ets:insert(TableId, {muted,NewList})
          end,
          serve(RcvList,Key,TableId)
        end;
      false ->
        io:format("You are not authenticated to do that~n"),
        serve(RcvList,Key,TableId)
    end,
    serve(RcvList,Key,TableId);
  {disconnect, Usr, Other} ->
    io:format("~p disconnected ...~n", [Usr]),
    NewRcvList = lists:delete(Other, RcvList),
    erlang:disconnect_node(Usr),
    serve(NewRcvList,AuthKey,TableId);
  {Text, Usr, Other} ->
    case user_is_muted(TableId, Usr) of
    false ->
      io:format("~p: ~s~n", [Usr,Text]),
      case lists:member(Other, RcvList) of
      false ->
        NewRcvList = [Other|RcvList],
        relay_all(NewRcvList, atom_to_list(Usr) ++ ": " ++ Text),
        serve(NewRcvList,AuthKey,TableId);
      _ ->
        relay_all(RcvList, atom_to_list(Usr) ++ ": " ++ Text),
        serve(RcvList,AuthKey,TableId)
      end;
    true ->
      Other ! "You have been muted!!"
    end,
    serve(RcvList,AuthKey,TableId)
  end.

% @doc remove newlines
rmn(Str) ->
  re:replace(Str, "\n", "", [global,{return,list}]).

% @doc to be used with the crypto stuff
salt() ->
```

```
  {A, B, C } = now(),
  [K, L, M] = lists:map(fun(X) -> integer_to_list(X) end, [A, B, C]),
  K ++ L ++ M.

% @doc interfacing function for relay_node function
relay_all(RcvList, Text) ->
  relay_node(RcvList, Text).

% @doc relay a message to all currently connected nodes
relay_node([]   , _) -> [];
relay_node(Nodes, Text) ->
  [H | T] = Nodes,
  H ! Text,
  relay_node(T, Text).

% @doc get the token that was given from the server
get_key(TableId) ->
  [{_, Token}|_] = ets:lookup(TableId, key),
  Token.

% @doc true if the user is muted
user_is_muted(TableId, UsrAtom) ->
  [{_,List}] = ets:lookup(TableId, muted),
  lists:member(UsrAtom,List).
```

Listing 17: Ermis Server

References

[1] Joe Armstrong. Making reliable distributed systems in the presence of software errors. 2003.

[2] Ericsson. Erlang Online Manual, 2013. http://www.erlang.org/doc/.

[3] Fred Hébert. Learn You Some Erlang (for great good!), 2013.