UNITRIP

Distributed systems and middleware technologies

Project documentation

Matteo Mugnai, Filippo Puccini

1 SOMMARIO

1. INTRODUCTION

Unitrip is a distributed web-app that allows users to create trips, which other people can join, in order to organize group trips quickly and easily; users can also add trips in their favourites list, in order to keep trace of trips in which they are interested. The architecture of the web-app is composed by a server part, written in erlang, and a client part, written in java, using tomcat server to deploy the application.

1.1 USE CASE

Unitrip provides three types of users: unregistered user, registered user and admin user.

An unregistered user can:

- Register to the application

A registered user can:

- Log-in to the application
- Create a new trip through an apposite form
- Visualize trips proposed by the community
- Visualize their own favourites trip list
- Visualize list of participants of a specific trip
- Join an active trip
- Leave a trip already joined
- Log-out
- Add/remove a trip to/from his favourite list

An admin can:

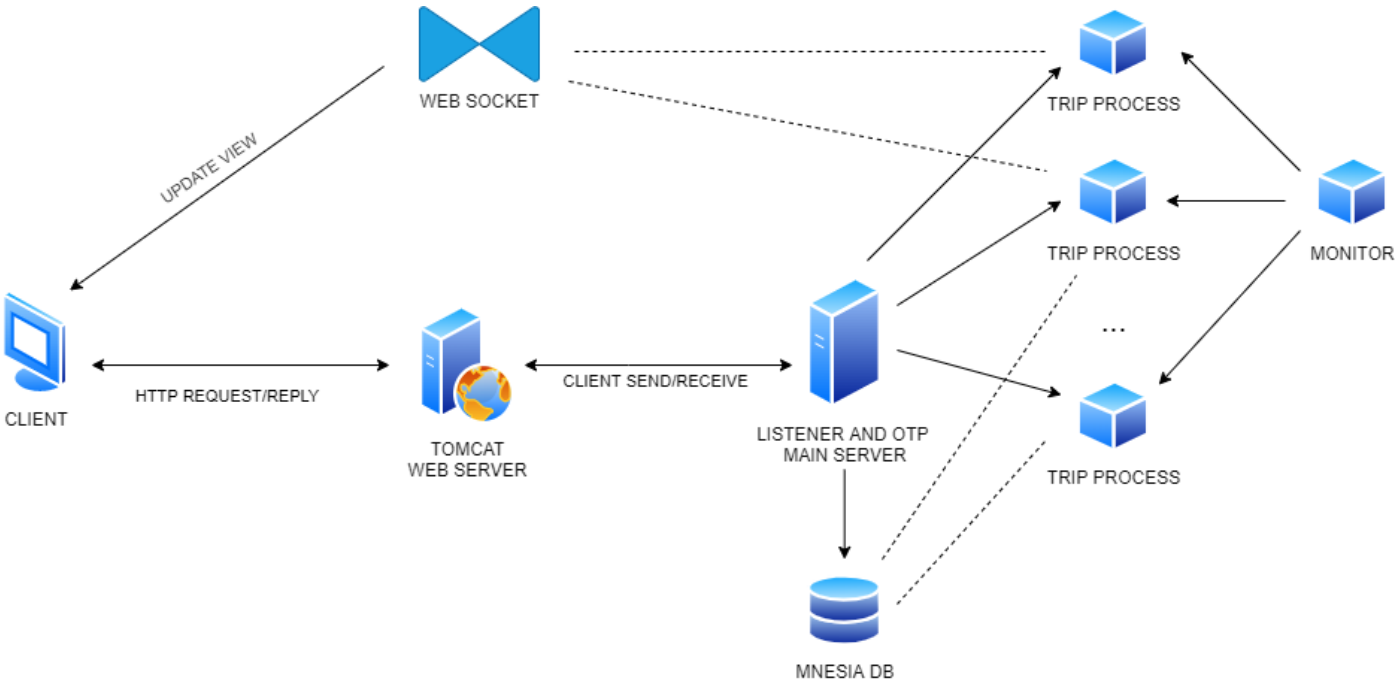
- Delete an ongoing trip

The application must therefore keep the information about the list of participants (and the user favourites list) and show any changes in real time; for each trip a countdown will be shown, after that the trip will be deleted from the list of active trips and will be no longer accessible.

These requirements involve in **communication and synchronization issues**: the system is in charge to communicate to users modifications in the list of participants and in the numbers of seats available, in addition it has to maintain update information about trips to display consistent data. To manage these problems, we exploit erlang for the server, which independently manages concurrency problems, and websockets for dynamic page updating. More detailed information on this topic will be provided later in this documentation.

2 SYSTEM ARCHITECTURE

Below you can see the overall architecture of the system:



As already described above, the architecture is divided into two distinct parts: client side and server side.

- **Server**: developed in erlang, it has the task of responding to clients requests, spawning new processes when users want to create new trips and keeping consistent information about trips. In addition, we have a monitor process on server side to keep trace of the state of the processes spawned by the main server.
- **Client**: developed using Java Servlets, Jsp and WebSockets. The client has the task of making the requests arrive to the server, directing the messages to the right process, which can be either the listener (which in turn will forward them to the main server), or one of the processes that manage the trips; it also has the task of creating and keeping the web page updated

So, a typical scenario could be that in which a user, after having registered and having logged in, decides to create a new trip, thus sending the request to the listener; this in turn will contact the main server, which will spawn a new process which, once started, awaits for requests. Now other users will be able to join this new trip: through the websockets, in real time, the number of participants will be shown, until there are no more places available. At the end of the trip, it will be removed from the list of active trips and will be saved in the database, while the corresponding process will be terminated.

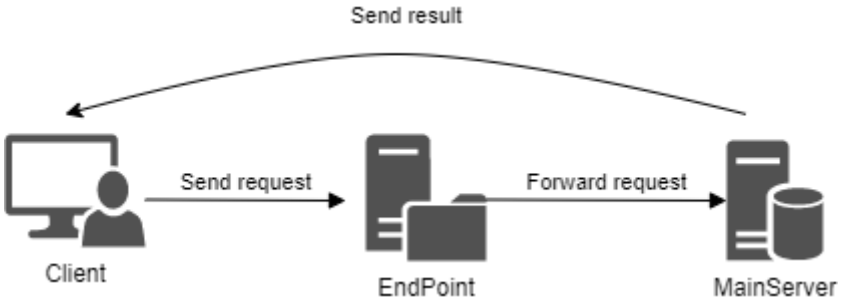
2.1.1 The server

The server side is composed by six different Erlang modules:

- Erlang server app
- Supervisor
- Monitor
- Trip
- Loop Server
- MnesiaDB

2.1.2 Erlang Server App

This module is the main server of the application and implements an OTP gen_server (which guarantees us reliability and efficiency) with the relatives callback functions. As shown in the figure below, it is in charge to communicate with the loop server, an endpoint useful to avoid a direct channel between client and main server, and compute operations based on the request type of the client. These operations are performed by the handle_call functions (callback functions), called by the OTP gen_server.



The services that the main server offers are:

- *Register new user*: adding a new user in the database.
- *Login*: check of the correctness of the password.
- *Create trip*: spawn a new process that is in charge to maintain information about the trip and direct communicate with the client.
- *Get active trips*: operation performed when a user go to the homepage of the web application; returns the list of the active trips, ordered by their popularity.
- *Get trip by name*: take all the information about a trip, given his name.
- *Get user favourites list*: returns a list of the favourites trips of a given user.
- *Delete trip*: permitted only to the admin, delete a specific trip.

The state of the server keeps a Pid list of active processes, useful for the implementation of get trips; if the main server should be shut down for some reason, when it is restarted, the necessary processes are spawned and the Pid of the new active processes is updated in the database, also updating the server state.

2.1.3 Loop Server

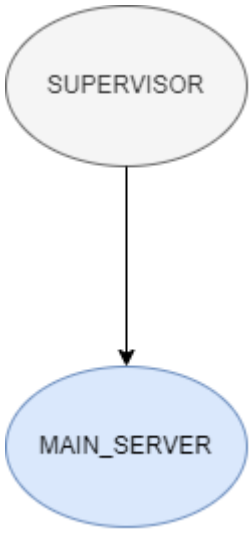
As mentioned before, this module implements the loop server, an endpoint useful to act as a link from the client to the main server, so that the main server always receives messages in the right format. Below you can see a small snippet of the code.

```
39  listener_server_loop() ->
40   receive
41    {From, register, {Username, Password}} ->
42     io:format("[LISTENER] Received a request for registration.~n"),
43     Result = erlang_server_app:register_request(Username, Password),
44     From ! {self(), Result};
45    {From, login, {Username, Password}} ->
46     io:format("[LISTENER] Received a request for login.~n"),
47     Result = erlang_server_app:login_request(Username, Password),
48     From ! {self(), Result};
```

2.1.4 Supervisor and monitor

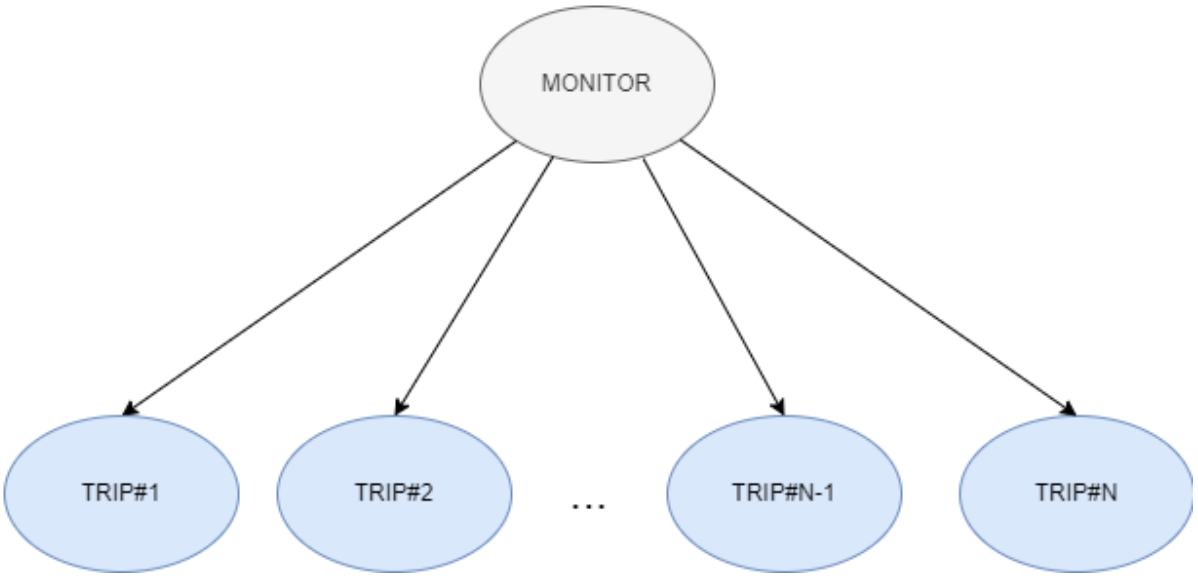
The supervisor and the monitor help us to improve the reliability of the entire application and are respectively in charge of monitoring the main server and the processes that manage the trips created by the users.

To implement the supervisor, we used an OTP supervisor behaviour, setting some flags according to our needs; in particular we set the main server process as permanent: this means that it will always be restarted when necessary, adopting a one for one policy. The supervisor process has only one child, that is the main server, and the whole application can be started by its **start_link ()** function.



The monitor, on the other hand, takes care of the trips processes, restarting them if necessary. It is a custom monitor that maintain in his internal state a list of the processes monitored and, based on the exit reason of the monitored processes, we can have two distinct scenarios:

- If the exit condition is normal or killed, the Pid of the process concerned is deleted from the internal server state; this can happen if the trip expires or if the administrator decides to delete it.
- If the exit reason is not one of these two, a read is performed in the database to retrieve the information on the trip crashed and the monitor spawns a new process: then it updates the trip Pid in the database with the new value, keeping all other attributes unchanged.



2.1.5 MnesiaDB

For storing all the information regarding users and trips we make use of Mnesia, which permits us a fast key-value lookup and an internal handling of concurrent access to the data. Mnesia is started from the main server, and contains two table:

- **User**: contains users data, has Username, Password and Favourite_Lists as attributes .
- **Trip**: contains trips data, has TripName, Pid, Organizer, Destination, Date, Seats, Partecipants, User_Add_To_Favourite

Below you can see the structure of the records :

```
18  -record(user, {username, password, favorites}).
19  -record(trip, {name, pid, organizer, destination, date, seats, partecipants, user_add_to_favorites}).
```

Because we want to keep data in a persistent way, so even there is a crash in our server the record shouldn't be lost, the tables are saved using the **disc_copies** option, thus we maintain the record store in memory and on the disc. Every time the server starts Mnesia it waits if the table are present, otherwise we create the table again.

The data of the trips are updated every time they are modified (for example when a new participant is added to a trip, or if a user adds that trip to their favorites list); in fact, in case the process crashes, we must have consistent information through which to create a new process that replaces the previous one.

```
21  start_mnesia() ->
22    mnesia:create_schema([node()]),
23    %% application:set_env(mnesia, dir, "~/mnesia_db_storage"),
24    mnesia:start(),
25    case mnesia:wait_for_tables([trip, user], 5000) == ok of
26      true ->
27        io:format("[MNESIA] Mnesia started correctly. ~n"),
28        ok;
29      false ->
30        io:format("[MNESIA] New schema created. ~n"),
31        mnesia:create_table(user,
32          [{attributes, record_info(fields, user)}, {disc_copies, [node()]}]),
33        io:format("[MNESIA] Table user created. ~n"),
34        mnesia:create_table(trip,
35          [{attributes, record_info(fields, trip)}, {disc_copies, [node()]}]),
36        io:format("[MNESIA] Table trip created. ~n")
37    end.
```

2.1.6 Trip

The trip module has the task of exposing the services related to a trip, and to keep the count of the users who have put it in the favorites list, to measure its popularity in the community.

When a user wants to create a new trip, a request is sent to the main server, which, after checking the name chosen for the trip (which is unique), starts the new process by calling the **init_trip ()** function, which in turn calls the **listener_trip ()**, which exposes the services; moreover, every second the process sends a message to itself to check that the expiration date has not been exceeded: if that's true, the process ends and the data are stored on Mnesia. The internal state of the process maintain all the information about the trip: it is useful to avoid unnecessary read on the database and to speed-up the application.

The services offered by this module are:

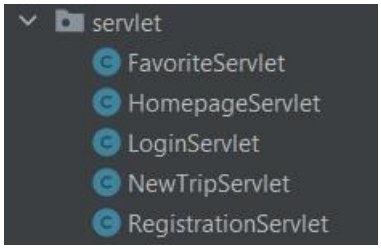
- *New participant*: add a new participant to the trip, if there are enough seats available.
- *Delete participant:* delete a participant and update available seats.
- *Add to favorites*: add the trip to the user favourite list.
- *Delete from favorites*: delete the trip from the user favourite list.
- *Get participants*: returns the list of participants in the trip.
- *Get seats available*: returns the number of seats available.

2.2 THE CLIENT

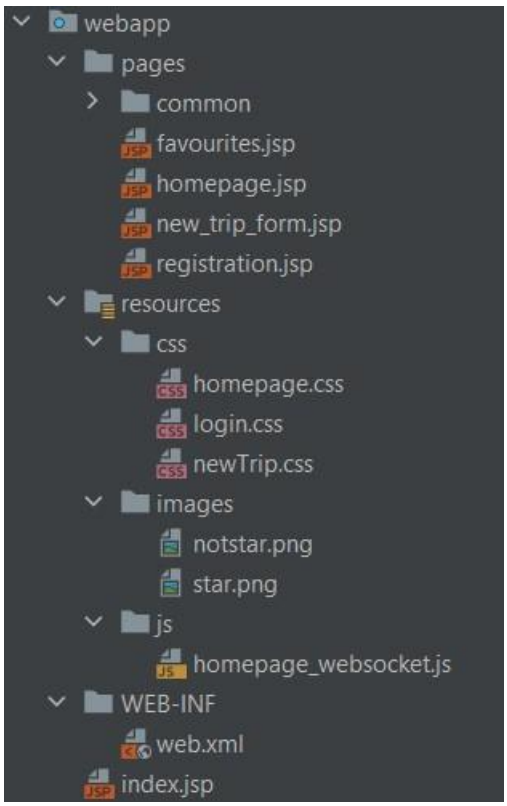
The web application shown to the user is generated via Apache Tomcat web-server, the page update is managed in real-time via web-sockets, the front-end part is done with Bootstrap (for the static content). The dynamic content is loaded combining Java Servlets and Jsp pages.

2.2.1 Tomcat Web Server, Java Servlets and Jsp pages

In Apache Tomcat web-server HTTP requests arriving from the user are processed by Java Servlets:



As shown in the above figure, there is one Java Servlet for each page of the web-application. From the generated page is possible to follow some hyperlinks to find other pages of the web-app.
The Webapp folder is organized in the following way:



The actual HTML is generated via JSP technology, with some scriptlets, for each page, to have a logic separation between the "View" part and the rest of the application.

In order to add a layer of security to the application a couple Java Filter was added (responsible for checking that requests to "protected" pages arrives from users that are already logged in).

In order to update the "Model" part of our application some requests to the Server-Side part in Erlang needs to be done. This is dealt in the communication package.

The Jinterface library is used: for each client there is a different mailbox contained in one single "Erlang" node, which is unique since obtained from the session token of the client. Then a request/reply communication happens with the Erlang Main Server or with the related Trip process. The communication will return back a result in order to communicate to the user if the request made has succeeded or not.

2.2.2 WebSocket

In order to update the application state without requesting periodically a web-page, Web-socket technology is used.

```java
@ServerEndpoint(value="/homepage_endpoint/{username}", decoders = HomepageDecoder.class, encoders = HomepageEncoder.class)
public class HomepageEndpoint {

    private static final Set<HomepageEndpoint> homeEndpoints = new CopyOnWriteArraySet<>();
    private Session session;
    private static final HashMap<String, String> users = new HashMap<~>();

    @OnOpen
    public void onOpen(Session session, @PathParam("username") String username) {
        // Metodo eseguito all'apertura della connessione
        System.out.println("[MAIN MENU ENDPOINT] OnOpen " + username + " is entering");
        this.session = session;
        homeEndpoints.add(this);
        users.put(session.getId(), username);
    }

    @OnMessage
    public void onMessage(Session session, @PathParam("username") String username, Message message) throws EncodeException, IOException {
        // Metodo eseguito alla ricezione di un messaggio
        // La stringa 'message' rappresenta il messaggio
        // Il valore di ritorno di questo metodo sara' automaticamente
        // inviato come risposta al client. Ad esempio:
        System.out.println("[MAIN MENU ENDPOINT] OnMessage");
        System.out.println("[MAIN MENU ENDPOINT] Trip list is going to be broadcast");
        System.out.println("[MAIN MENU ENDPOINT] Name of the trip: " + message.getName());
        System.out.println("[MAIN MENU ENDPOINT] Action : " + message.getAction());
        message.setUser(username);
        broadcast(message);
    }
```

HomepageEndpoint: responsible for updating the state of the all the trips w.r.t. the list of participants and the available seats counter. It receives a json message from related javascript function every time Join or Leave buttons are clicked. Then OnMessage method is executed and relative updates are broadcasted to all client connected to this WebSocket (an user is connected to this WS every time homepage.jsp page is loaded). Broadcast action send back messages that are captured from javascript onMessage event that is responsible to commit modifications.

3 SYNCHRONIZATION AND COMMUNICATION ISSUES

As mentioned before, the main problems about synchronization and communication in our web-app are:

- The updating of the seats and the list of participants, that must be show in real-time to online users.
- The remaining time displayed for each trip must be consistent among all the users.
- The information regarding active trips must be consistent (seats, list of participants, popularity, expiration date, …).

In order to solve these types of problems and ensure a good degree of parallelism to handle several requests from different users, the server has been implemented entirely in Erlang. This programming language makes it very easy to solve this type of problem, thanks to its internal features:

- **Processes do not share memory**: thanks to the actor model, processes communicate by asynchronous message passing.
- **Fast, high-performance code**: Erlang is a functional programming language, so function has no side effects and are idempotent, lack of state during computation, lazy evaluation is exploited.
- **Independent to network structure:** Processes can be created without knowing the structure of the nodes making up the network.

In fact, the choice of spawning a process for each trip was also dictated by the fact that in this way the number of requests to be handled by the main server drops considerably, guaranteeing greater stability and speed of the system.

In addition, to ensure consistent data operations, we exploit Mnesia transactions, in order to guarantee mutex access to the data: those are ACID operations capable of ensuring Atomic and Isolated statement. Thanks to the transactions, there is no risk for critical races on the data between processes.

To address the problem of catching real-time modifications of users regarding seats and list of participants we make use of WebSocket technology to monitor number of available seats and avoid possible inconsistencies between users. In other words Websockets keep updated and synchronized the homepage of all the users that are accessing simultaneously to our webapp. In this way an user who clicks on join button is sure to book really the seat, in fact available seats counter is always updated and this prevent a situation in which we can have possible races between users to book the last places of a trip. Websockets keep fresh also the list of participants of all trips posted on the homepage. With this solution multiple users, with different sessions, connected to the service at the same time can view real time modifications on homepage.