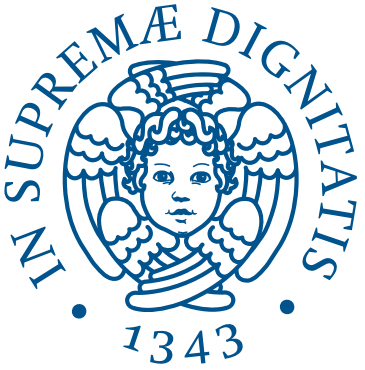

UNIVERSITÀ DI PISA

Computer Engineering

Distributed Systems and Middleware Technologies

Project Documentation

TEAM MEMBERS:
Tommaso Burlon
Francesco Iemma
Olgerti Xhanej

Academic Year: 2021/2022

Contents

1 — Project Specifications

AuctionHandler is a distributed web-app in which users can sell their goods by creating Online Auctions. Registered users have the possibility to join an ongoing auction in order to buy a good in case they beat the concurrence by setting an higher offer on a given limited time.

1.1 Use Cases

An *Unregistered User* can:

- Register to the service

A *Unlogged User* can:

- Login to the service

A *Logged User* can:

- View the list of ongoing Auctions
- Create a new Auction
- Join an ongoing Auction
- Logout
- After Joining an Auction:
  - Make an offer
  - View list of participants
  - View past offer history
  - View the remaining time
  - Wait until the end of the Auction and then exit
  - View Auction result

The *System* must:

- Remember registered users
- Remember ongoing auctions
- Remember auction participants
- Choose in a unique way the auction winner
- Remember offers history
- Synchronize the remaining time, the auction participants, the offer history for an auction for each user
- Synchronize the list of ongoing auctions for each user

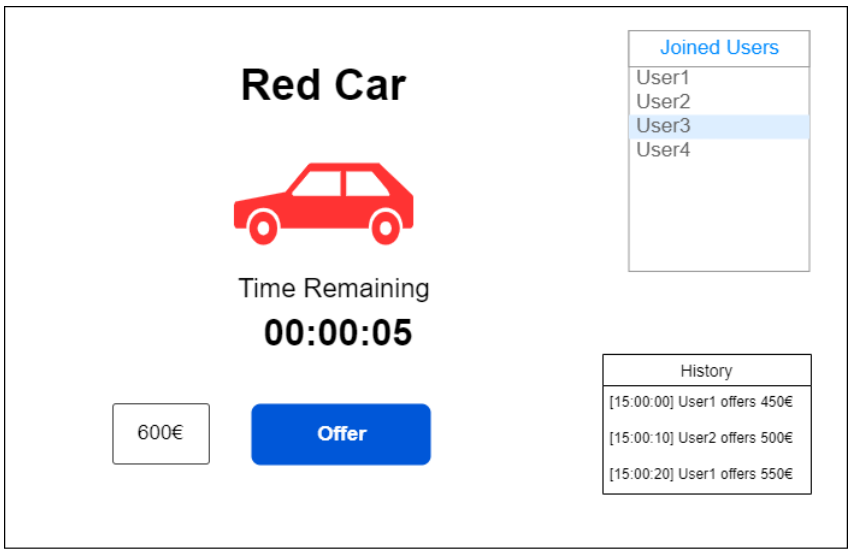

Figure 1: Mock-up of the main interface of the auction

1.2 Synchronization and Communication Issues

On the application we have the following synchronization and communication issues:

- Client nodes need to be synchronized with the same remaining time of the auction, the same offer history for a given auction, the same list of joined users on a given auction and the same list of available ongoing auctions.
- In case a client makes a valid offer, the server will be in charge of communicating to other clients nodes the information regarding the made offer.
- In case a client creates a new auction, the server will be in charge of communicating to other clients the information regarding the newly created auction.

1.3 Design Ideas

We were thinking of implementing the system in the following way:

- **Client Nodes**: User Interface via HTML/CSS, generated via Java Servlets and JSP. Each client node will have a dedicated Erlang node for communicating with the server
- **Server**: Made entirely in Erlang. The responsibility of the server will be performing persistent data storage and handling the communication and synchronization between different client nodes

2 — System Architecture

A graphical representation of the system architecture can be seen in figure 2.

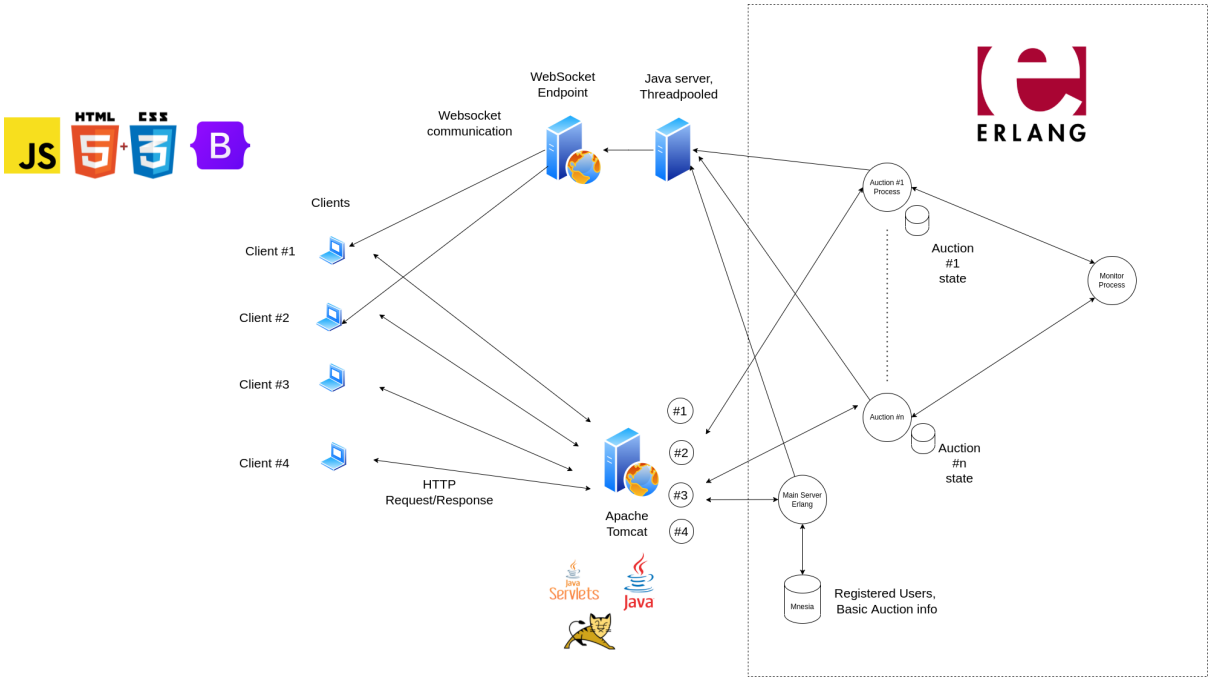

Figure 2: System Architecture graphical representation

We can divide the overall system in two part:

- The server side part: developed in Erlang, it is in charge of handling the request coming from the users and it has to handle the auctions and to maintain the global view of an auctions consistent among the users.
- The client side part: depeloved in Java, using EJB and JSP, and in Javascript for what regards the websocket. It is in charge of retrieving information from the server, create the GUI, update the GUI in order to let the user have a consistent view of the state of the auction and of the overall system.

2.1 Server Side

The Server side part is written in Erlang and contains the following components:

- The Main Server
- Auction Handler
- Auctions Monitor
- Supervisor
- MnesiaDB

2.1.1 Main Server

The main server is in charge of receiving requests from the client side and saving all the necessary information in the database.
Since robustness is required the main server is an OTP gen_server. The main server is actually composed by an endpoint and by the actual server. The endpoint is a sort of API used to make a call to the OTP gen_server:

1. The client sends a message to the endpoint
2. The endpoint receives the message, analyzes it and calls the function that uses the function call in order to make a request to the OTP gen_server
3. The OTP gen_server receives the request and calls the function handle_call that matches with the incoming message pattern.
4. The handle_call function will manage the request arrived from the client and sent back the required information (saving info in the database or spawning process if necessary)

The functionalities that the server exposes are:

- Register a new user

  The user will be registered (i.e. added to the database) if and only if its username is not already present in the database.

- Login

  The username and password are checked and if both are correct a positive reply is sent back

- Create Auction

  A new auction is created (added to the database) if and only if another auction with the same name is not already present in the database. When the auction is created a new process (i.e. The Auction Handler) is spawned and this process is added to the processes that the Auctions Monitor has to monitor (for more detail about the monitoring see 2.1.3). From now on for all the information regarding that auction the client has to contact the auction handler at the pid present in the reply message.

- Retrieve Auction PID

  Return back to the user the PID of the handler that manages the auction with the name passed to the server in the payload of the request. This is very useful for instance when the auction handler crashes and it is re-spawned. In that cases the PID changes and so the client has to know the new one.

- Retrieve Active Auctions

  Return back the list of all the active auctions (They are saved in the database).

- Retrieve Passed Auctions

  Return back the list of all the passed auctions (They are saved in the database).

- Update Win

  Save the username of the auction's winner. This functionality is used by the auction handler in order to store in the persistent storage the name of the winner.

- Update PID

  Change the PID of the auction handler associated to a particular auction. This is used by the auction monitor when an auction handler crashes and so it is re-spawned with a new PID.

2.1.2 Auction Handler

The auction handler's role is to manage an auction maintaining its state. Its functionalities are:

- Maintain the time that remains for the end of the auction.

  Each seconds a clock message is sent by the auction handler to itself in order to emulate the passing time. When the remaining time is 0 (the information about the remaining time are kept into the handler state) then the auction handler has to decide the winner. After this decision the handler kills itself.

- Remember which users are online at any moment

  Each time a new user enters the auction or leave it, we update the handler state and we notify the client through the Java Listener.

- Register the offers

  When a new offer arrives the handler saves it into an ordered list[1] (the higher offers is at the beginning of the list) in its internal state. Notice that the offer is registered only if the offer amount is higher w.r.t. the minimal amount.

- Decide the winner

  The winner is the one whose offer is in the first position in the list of the offers. If the list is empty then this means that no offers have been made and so there is no winner. When the winner is decided the client and the main server are notified.

2.1.3 Monitor And Supervisor

It is possible to see a graphical representation of the monitoring relationships among processes in figure 3. The monitoring is performed by **My Supervisor** and **Auctions Monitor**.

[1] The list is composed by tuples e.g. {"UserA",10}

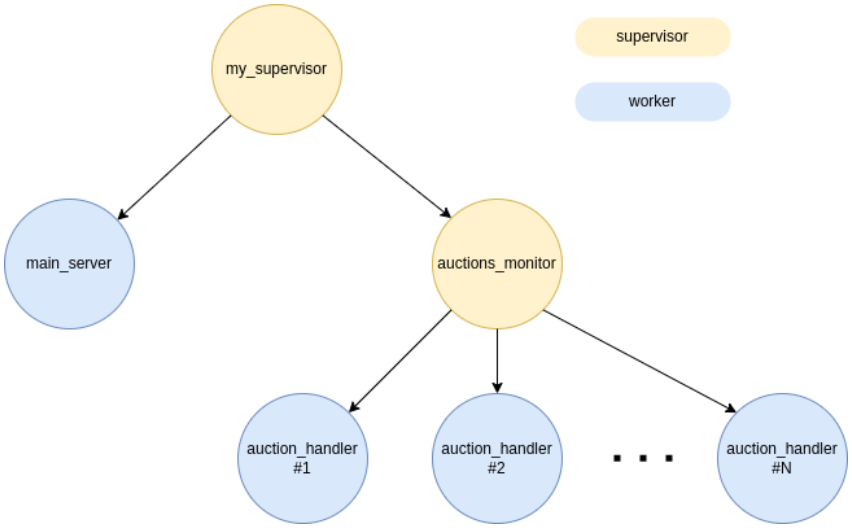

Figure 3: Graphical representation of how the monitoring is implemented. Notice that in the draw the arrow starts from the monitoring process and ends to the monitored process.

My supervisor is the main supervisor and due to its importance and the robustness required is developed using an OTP supervisor behaviour. The process that it monitors are main_server and the auctions monitor, both of them are *permanent*. A monitored process is said to be permanent if it is always restarted when necessary.
The restart strategy is *one_for_one* since with this stratey we are able to provide the service even if one between main server and auctions monitor crashes. One for one means that if one of the monitored process crashes then only the crashed one is restarted. In our scenario we can have two possibilities:

- Main Server crashes

  If the main server crashes before the auction monitor we can just restart it and we have no side effect.

- Auctions Monitor crashes

  If the auctions monitor crashes and it is restarted then all the auctions created before the crash are not monitored anymore. Anyway that auctions are still reachable by the user and, moreover, the situation is temporary and when the old auctions are terminated we have the standard situation in which all the auctions handler are monitored.

Thus as we can see with this strategy in both cases we are still able to provide the correct service.

The auctions monitor is a custom monitor which is in charge of restart an auction handler when it crashes. To do so it saves in its internal state all the necessary info, in this way when a process crashes abnormally, the auctions monitor restart it and it notifies the main server in order to update the pid with the pid of the new re-spawned auction handler.
Notice that when the auction handler is re-spawned the state of the auction is the initial one (the state we would have just after the creation of the auction).

Another thing to highlight is that if the auction handler terminates correctly the auctions monitor does nothing. In fact each auction handler kills itself when the auction ends.

2.1.4 MnesiaDB

The module MnesiaDB is the one in charge of managing the persistent storage. Mnesia has three options regarding the storage persistence (the following three definition are taken from the documentation[2]):

- *ram_copies*. A table can be replicated on a number of Erlang nodes. Property ram_copies specifies a list of Erlang nodes where RAM copies are kept. These copies can be dumped to disc at regular intervals. However, updates to these copies are not written to disc on a transaction basis.

- *disc_copies*. This property specifies a list of Erlang nodes where the table is kept in RAM and on disc. All updates of the table are performed in the actual table and are also logged to disc. If a table is of type disc_copies at a certain node, the entire table is resident in RAM memory and on disc. Each transaction performed on the table is appended to a LOG file and written into the RAM table.

- *disc_only_copies*. Some, or all, table replicas can be kept on disc only. These replicas are considerably slower than the RAM-based replicas.

The option chosen is *disc_copies* since makes the database resilient against crashes and allow us to restart the processes without issues. Moreover since the database is kept also on RAM we have also quicker operations (w.r.t. the case disc_only_copies).

In the database there are two tables:

- User

  The attribute of this table are *name* and *password*. It is in charge of remember the users registered to the application.

- Auction

  The attribute of this table are *name, duration, startingValue, imageURL, creator, pid, winner*. It is in charge of remember all the info regarding an auction. An important attribute is *winner*, this attribute is used to distinguish between passed auction and active auction: if the auction is active winner is equal to none, instead if the auction is finished, winner contains a string with the name of the winner or a particular string indicated that there is no winner (if the auctions has received no offers).

A particular feature of Mnesia db is the possibility to use *Atomic transactions*. An atomic transaction ensures that all the operations inside the transaction are performed in an atomic way. This feature is used for the user registration and for auction creation. In both cases we need to register the user or create the auction only if there is not an auction/user with the given username/auction_name. To ensure this we need a read on the database and then a possible write. If the two operations are not done in an atomic way (i.e. inside the same transaction) we may have the following situation:

[2]https://www.erlang.org/doc/man/mnesia.html

- Alice tries to register itself with the username "john-doe"
- READ (triggered by Alice): There are no user whose username is "john-doe" so I can perform the write
- Bob tries to register itself with the username "john-doe"
- READ (triggered by Bob): There are no user whose username is "john-doe" so I can perform the write
- WRITE (triggered by Alice): I write "john-doe" in the table *user*.
- WRITE (triggered by Bob): I write "john-doe" in the table *user*. This write will cause an inconsistency because we will have two users registered with the same name.

In practice we have a problem known as TOCTOU (Time Of Control Time Of Use) which consists in the fact that I can change the table after the check done during the read, thus when I perform the write the state of the table is changed because another operation has been performed between the read and the write.
The solution to this problem is the use of atomic transactions provided by Mnesia DB.

2.2 Client Side

The web application shown to the user is generated via Apache Tomcat web-server, the page update is managed asynchronously via web-sockets, the front-end part is done with Bootstrap (a pretty famous CSS & Javascript framework) and Javascript "vanilla".

2.3 Web-server with Apache Tomcat

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

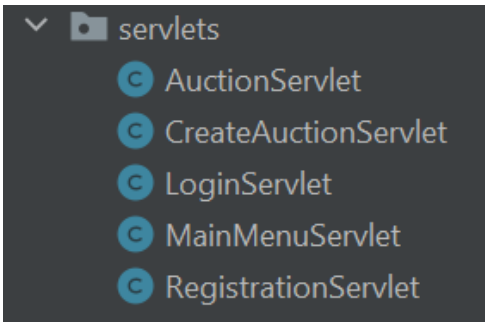

Figure 4

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:

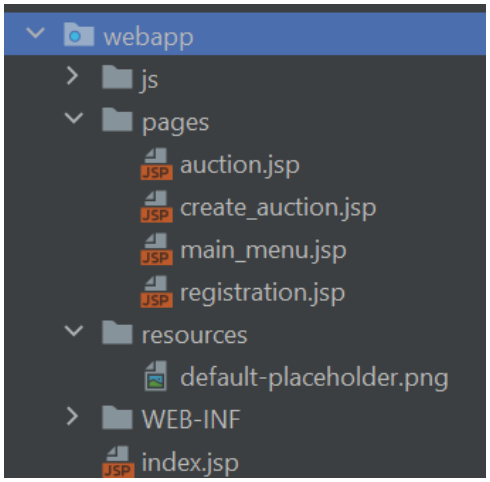

Figure 5

The actual HTML is generated via JSP technology, with some script-lets, for each page, in order to have a logic separation between the "View" part and the rest of the application. In example the *main_menu.jsp* scriptlet contains some logic to generate different cards by iterating the list of auction returned by the Erlang server.

In order to add some layers of security to the application a couple of Filters were added:

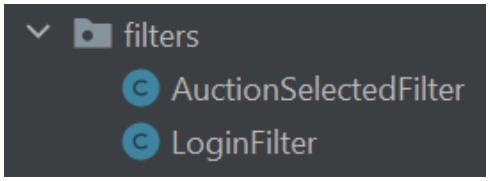

Figure 6

- **LoginFilter** is responsible for checking that the request to some "protected" pages arrives with users already logged in (this can be assured by analyzing the session of the particular user, indexed by a cookie).
- **AuctionSelectedFilter** prevents a logged user to access an auction without having pressed the related button. This is done because at the same time multiple auctions can be running at the same time.

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:

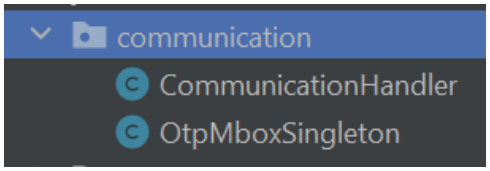

Figure 7

With the usage of Jinterface library: for each client is adopted a different mailbox of one single "Erlang" node, obtained from the cookie of the client. Then a request/reply communication happens with the Erlang Main Server and if needed with the related Auction Handler, in case an auction page is requested. The communication will return a result to communicate if the request of the client has succeeded or not (i.e. a client registration to the service may fail in case of a duplicate username).

2.4 Web-sockets

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

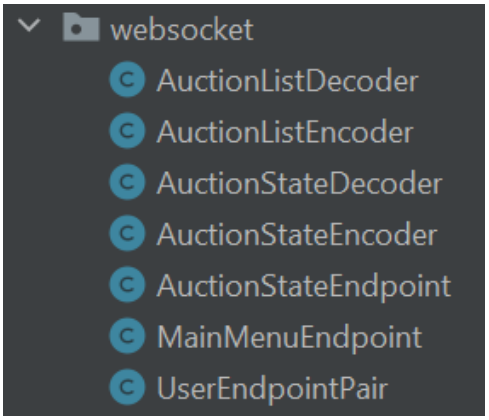

Figure 8

As we can see in the image above two different web-socket endpoint are used:

- **MainMenuEndpoint**: responsible for updating the state of the main menu w.r.t. the list of active and inactive auctions. Clients connected to this endpoint are registered through the usage of a Java static *CopyOnWriteArraySet* for dealing with concurrent changes of the connected client list.
- **AuctionEndpoint**: responsible for updating the state of the auction w.r.t. the list of joined users, the list of valid bids, the auction time, and the winner election. Clients connected to this endpoint are registered through the usage of a Java static ConcurrentMap¡String, Set¡UserEndpointPair¿¿ data structure. With the latter, for each auction name is associated a list of client endpoints, in order to correctly send messages only to clients of a particular auction, with a publish/subscribe fashion.

2.5 Java Listener

There's one actor missing in the whole picture, the Java Listener. The web-socket state update happens withe the following sequence of steps:

1. A client requests a change to the state of the model (i.e. creates an auction)
2. Apache Tomcat intercepts the requests and start the communication with the related Erlang server-side which actually stores the application state

3. The Erlang server-side sends in general **two messages for each requests**: one is fore the servlet itself for giving an ACK to the made request, one is sent to the **java listener** for triggering the state change to other clients connected (without need of a page refresh)

4. Java listener broadcast the state change to all interested users via web-socket

The advantages of using this approach are the following:

- Since messages are sent by the erlang server-side (which actually is the source of truth of the web-application state), the state returned is consistent.
- Thanks to web-sockets, there's no need to periodically sending request to Apache Tomcat. **Page updates happens only when there is a state change**, avoiding a considerable amount of computations.
- Some events are triggered by erlang side itself and not by the client (i.e. auction ending and winner election), so this approach update the web-application state correctly by simply avoiding to send one message to Apache Tomcat

The **cost** of this approach is the implementation of the Java Listener.

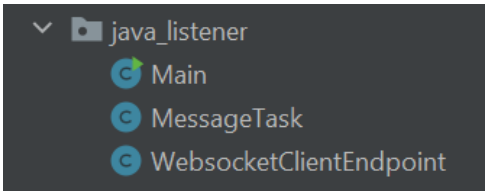

Figure 9

The Java Listener is implemented with a listener Jinterface node for getting requests from Erlang server-side. For handling multiple requests in parallel, threads are used: in particular, for avoiding the overhead due to the creation and termination of threads, **thread-pooling** is used via ExecutorService interface that implements a fixed thread pool. The task (called MessageTask) computed by each thread is related with sending a message (with the state information) to the correct web-socket endpoint.

2.6 User Interface

In order to ease the UI developing a framework like **Bootstrap** has been used, in order to create a responsive web-application in a simple way, with a set of ready-to-be-used css classes, without the need of reinventing the wheel by defining our set of css classes for getting the same result.

A couple of javascript scripts are then defined:

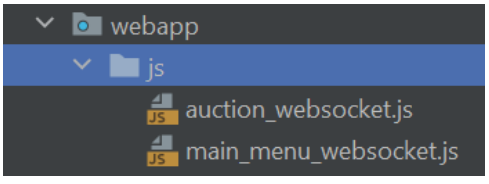

Figure 10

- **main_menu_websocket**: Responsible for connecting with the main menu websocket endpoint and to update the DOM of the HTML page when a message arrives.
- **auction_websocket**: Responsible for connecting the client to the auction websocket endpoint, to update the DOM of the HTML page when a message arrives and to update locally the timer.

For what concern form validation, **browser-default validation** was adopted (i.e. some fields are required to be added, some fields, like the auction duration, must contain a number).