Università di Pisa

Master Of Science in Computer Engineering and Artificial Intelligence

DS Auction

DS Auction

Student:
Megan Maremmani
Eleni Zwedu Belayhun
Ettore Di Donato

Academic Year 2025-2026

Table of Contents

Problem Description

The project consists of designing and implementing a working distributed software system. The following fundamental requirements must be satisfied.

Fundamental Requirements

- Completion of a functioning distributed software system.
- The project must address synchronization/coordination and communication issues.
- These issues must be clearly identified and discussed in the specification document.
- Any technological solutions may be used, provided they are justified during the presentation.
- The system must include at least one component implemented in Erlang.
- A working instance of the distributed application must be demonstrated across multiple nodes.
- All project documentation must be delivered to the instructor during the presentation.

1 DS Auction

DS Auction is the distributed web-application of your dreams.

In particular, DS Auction, is a **distributed web application** where Registered Users can sell their retro/vintage games and consoles creating their own online auctions where other users can bid concurrently with other users to buy them while also chatting together within the limited amount of time given by the auction itself.

2 System Requirements

In the following section, the requirements of the system are shown.

2.1 Functional Requirements

2.1.1 Unregistered User

- The system must allow a non registered user to browse the application.
- The system must allow a non registered user to register to the application with name, surname, password, age, email and payment information.
- The system must allow a non registered user to only view the ongoing auctions, bid history and chat history.

2.1.2 Registered User

- The system must allow a registered user to browse the application.
- The system must allow a registered user to login to the application with username and password.
- The system must allow a registered user to see the auctions and to bid in them.
- The system must allow a registered user to use the application's chat.
- The system must allow a registered user to send a request to sell an item for auction with: name, description, starting price, type, imageURL and status: all conditions are mandatory.
- The system must allow a registered user to see the bid and chat history of the specific auction.
- The system must allow a registered user to see the remaining time before the auction of an object ends.
- The system must allow a registered user to see the results.

2.2 System

- The system must remember the registered users.
- The system must organize the items for the auction.
- The systems must start the timer for the auction.
- the system must remember ongoing auctions.
- The system will choose automatically the winner after the timer ends based on the bids.
- The system will reset the timer at 30s each time a bid is made in the last 30 seconds.
- The system must synchronize the list of ongoing auctions for each user and the remaining time for each auction.
- The system must review the items sent by the user in order to have all the details filled correctly.

2.3 Non functional Requirements

- The system must be always available.
- The system must be fast responsive.
- The System must handle the bids correctly and in a serialized way.
- The system must handle the auctions correctly with no consecutive bids by the same user.

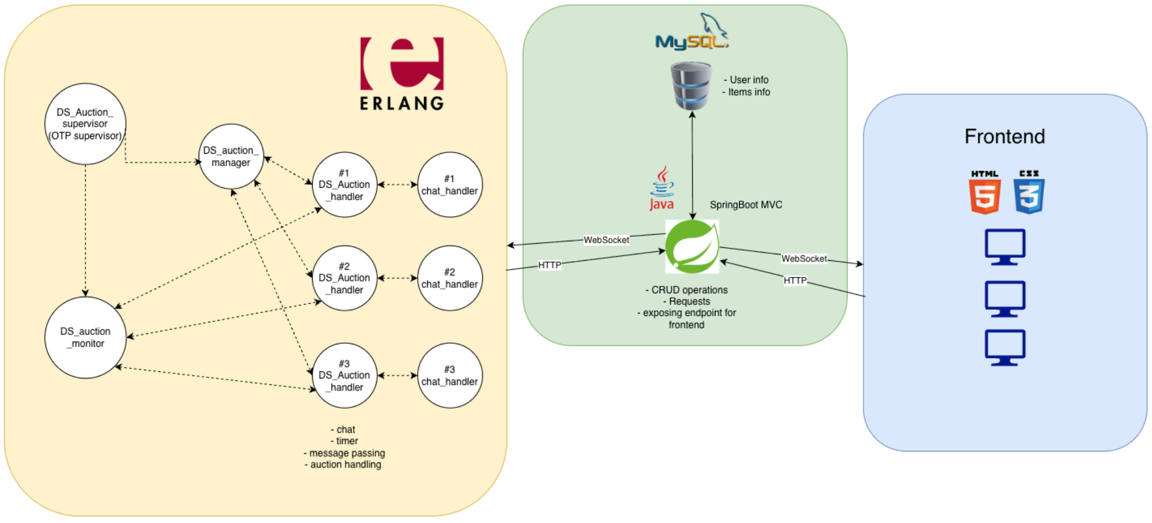
3 Synchronization and Communications issues

On the web application there are several synchronization and communication issues to address and to handle.

- **Race Conditions on Bids:** One of the main issues is to handle bids that happen at the same time, so the system must serialize the requests to have a consistent state.
- **Time issue:** All users must have the same coordinated timer for the auctions and their general remaining time.
- **Distributed Timer Coordination:** An issue arises also from the final 30 seconds of the auction where the timer is reset to 30 seconds and all users must see the same change at the same time.
- **Bids** All user must see the valid bids and their history and the server needs to communicate the correct bid happened to all users participating in the auction.
- **State Propagation:** All users must see all the ongoing bids and the server must communicate with all the users to ensure they all see also the new auctions happening that have been accepted by the admin.
- **Sequential bids:** the system must guarantee that the same user can't make consecutive bids, it must look at the state of the auction and the current bid.
- **Winner determination when auctions ends:** the system must determine and announce the winner consistently across all nodes when the timer ends.
- **Message Ordering in Chat and Messages at the same time:** Messages need to be properly handled, users can send messages at the same time but all the messages need to have the correct order. Without this there could be network delays and this could cause messages to appear out of context (For example an answer appearing before the question).

4 System Architecture

In the following figure is the over-all architecture of the system.



The architecture, in particular, can be divided in:

- **Frontend**: Made with HTML and CSS to have a personalized and ad hoc option for the application.
  Currently it is only used as a mock-up for the project GUI interface.
- **Database**: a MySQL database was chosen to ensure proper handling of the auction information, such as users and items, and to have ACID atomic operations. MySQL workbench was used to have the proper handling of the items and the users.
- **Client side**: The client side was developed in Java using Java SpringBoot MVC (Model, View, Controller) since it is a very practical tool for the handling of the HTTP requests and Websocket endpoints. It also communicates with the DB in order to get the items. It is responsible for handling the information flow from the client to the Erlang server and vice versa.
- **Server side**: The server side was implemented using Erlang and it is in charge of managing the users' requests and to handle the logic of the auctions, timing among the nodes and the chat of the different auctions so that all users can have a pleasant and consistent experience while using the application.

The Server Side written in Erlang is divided in two containers, the Manager on container **10.2.1.48** and the worker is on **10.2.1.13**. The Java/SpringBoot side is deployed on container **10.2.1.25** and the DB is deployed on container **10.2.1.47**.
To access the containers OpenVPN can be used with the **studenti2025.ovpn** file and all the codes to make the project work are on the gitHub repository (along with other useful files): `https://github.com/Ector55/DS_Auction`.

4.1 Server-Side Implementation (Erlang)

As stated in the previous paragraph, the server side was written in Erlang (version 28.3.1) and contains the following components:

- DS Auction handler
- DS Auction monitor
- DS Auction supervisor
- DS Auction manager
- Chat handler

4.1.1 DS Auction supervisor

This is the main supervisor that ensures that the system remains operational.
This module implements the `supervisor` behaviour to supervise the *Monitor* and the *Manager* processes.

The following functions are called:

- `start_link/0`: Starts the supervisor process and registers it
- `init/1`: Initializes the supervision flags with a `one_for_one` strategy, allowing 5 restarts within 10 seconds.
  - It uses the **one_for_one strategy**: If the Manager or the Monitor crashes, only the dead process is restarted, not the entire system.
  - First, it starts the monitor and then the manager: the order is important because the manager needs to have the monitor ready to function properly.
  - If one of these critical processes dies, it is automatically restarted, making the server highly stable.
  - Defines the child specifications:
    - **Child 1**: `DS_auction_monitor` .
    - **Child 2**: `DS_auction_manager`

4.1.2 DS Auction monitor

This module separates the business logic (handled by the Manager) from the technical supervision logic of the auction processes, it implements the Erlang OTP gen_server behaviour.

- It gets the request to monitor the auctions from the manager.
- If an auction process terminates (whether normally or due to a crash), the Monitor captures the 'DOWN' event and/or the message.
- Its main purpose is to forward the message to the manager to free the slot: the Monitor watches if they die, while the Manager decides what to do when they die.

In particular:

- **Record Definition**: `#state`: Contains `monitored_auctions`, a Map mapping `MonitorRef` → `{Pid, AuctionId}` top map the state of the auctions.
- `start_link/0`: Starts the monitor gen_server and registers it.
- `init/1`: Initializes the server with an empty state map and logs the start.
- `handle_cast/2`: `{add_to_monitor, Pid, AuctionId}`: Creates an Erlang monitor (`erlang:monitor/2`) for the given PID, stores the reference in the state map, and logs the action.
- `handle_info/2`: Handles the crash or termination of a monitored process and notifies the manager about them.

4.1.3 DS Auction manager

It will spawn the auction handler processes to start the auctions and then it requests the items to be loaded from java so it(java) can fetch them from the DB. It's an OTP **gen_server** behaviour. In Particular:

- State definition and constants
  - Defines constants for `JAVA_NODE`, `WORKER_NODE`, and `POLL_INTERVAL` (5000ms).
  - `#state`: Contains `active_slots`, a map of currently running auctions to keep track of which slots are busy/free.
- `start_link/0`: Starts the manager gen_server and registers it.
- `init/1`:
  - Calls `start_permanent_slots()` to make sure that the 3 worker slots exist and it requests java the item to be auctioned and it has 3 permanent active slots at a time and maps them to check if they are active or not.
  - (`check_for_auctions`): Every 5 seconds, it checks if the slots are still available or if they are occupied.
- `handle_info/2`:
  - `check_for_auctions`: If the slots are full, it stays idle; otherwise it calls `fetch_auctions_rpc` to get items from Java and `load_items_into_slots` to assign them that sends a synchronous request to java to retrieve the new items for the auctions.
  - When it receives the data of a new auction, it loads the item into an available idle handler and chat process and delegates to the monitor the role of monitoring both of them.
  - `{auction_ended, ...}`: Handles normal auction termination. Notifies Java of the result (sold/unsold), removes the auction from active slots, spawns a new idle handler, and registers it with the Monitor.

  - `{'DOWN', ...}`: When an auction is finished or has crashed, it senses that it is gone and will remove the Pid from the map and searches for new ones.

- **Private Helper Functions**:
  - `start_permanent_slots/0`: Iterates through IDs 1,2 and 3. Checks if `auction_N` exists on the remote worker node; if not, spawns it and monitors it.
  - `get_free_slots/1`: Returns a list of slot IDs (1, 2, or 3) that are not currently in the `active_slots` map.
  - `load_items_into_slots/3`: Recursively assigns fetched items to free slots.
  - `find_slot_by_pid/2`: Searches the active slot map to find the AuctionID associated with a given PID.
  - `notify_java/2`: Sends messages to the Java mailbox (`java_listener`). Handles `sold` (with winner/price) and `no_bids` cases.

4.1.4 DS Auction handler

The auction handler handles each of the different auctions (one handler per auction) while maintaining the state of the auctions. It also handles the bids correctly and the timing.
In Particular:

- `#state`: The process registers itself with a unique name and sets its initial state with price, duration, item name and starts the timer: it sends a clock tick every second to itself to keep track of time (parameters: `auction_id`, `current_bid`, `high_bidder`, `time_remaining`, `bids_history`, and the other parameters to get the state of the auctions.) and sends it to java as well.
- `start/1`: Registers the process as `auction_[ID]` and enters the `idle_loop`.
- `idle_loop/1`: Waits for a load item message and then initializes the state with item details and starts the clock timer before entering the main `loop`.
- It waits for the different types of messages in order to handle the different kinds of bids, times and the different requests.
- The bid handling is managed depending on the time of the bid and the user, in particular:
  - If the auction time is completed, the bid request is declined
  - If a user bids consecutively, the offer is rejected (as a chosen rule).
  - If the offer is valid, meaning that we have a different user from the previous one and a higher amount, the offer is accepted
  - if a valid offer comes in the last 30 seconds of the auction, the timing will be reset at 30 seconds.
- When the bid is accepted as the highest, it will update the bidding history
- `handle_winner/1`: At the end of the auction timer, the unique winner is calculated and broadcasted. It stops the chat, notifies Java of the winner (or lack thereof), sends the result to the Manager, unregisters the name, and terminates the process.

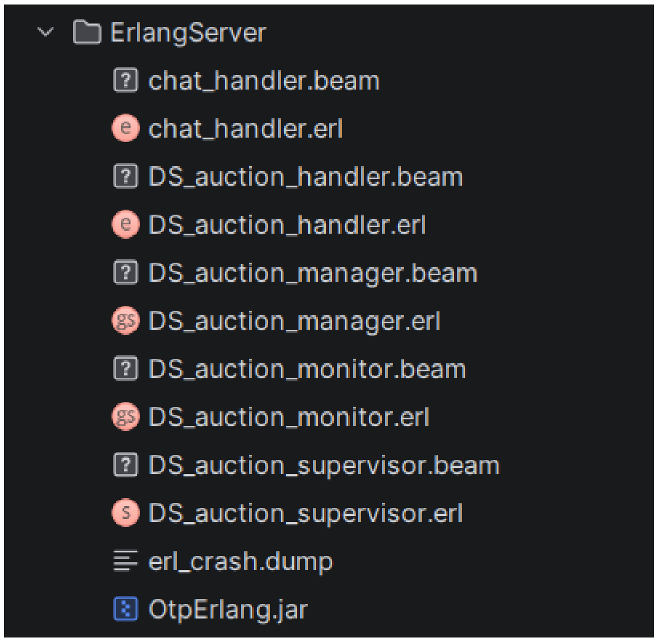
- The Handler is also constructed in a way that it communicates with the java nodes and sends periodic updates regarding the state and offers and formats the different data in tuples to send to java.
- When an auction ends, it stops the process and sends a stop message to the chat. It also informs Java that the process has ended, communicating the winner and the final price.
- The module also implements the Cristian's algorithm to ensure the synchronization between the Erlang processes:
- **Time Synchronization API**:
  - `get_server_time/1, 2`: Fetches the raw system time from the auction process.
  - `sync_time/1, 2`: Implements Cristian's algorithm. Calculates Round Trip Time (RTT) and Offset to synchronize the other processes clock with the master.

4.1.5 DS Auction chat handler

The chat handler is a module that manages the chats in each of the different auctions.

- Each Auction has its own chat process, and every user participating in the auction can see the chat.
- Registers the process as `chat_[ID]`.
- Enters `loop` with an empty list.
- When a user sends a message, every client connected in that auction can see the message, and it also sends the message to java
- It monitors the connecting clients; if a participant leaves or crashes, the handler removes them from the broadcast list:
- `loop/2`:
  - `{join, ClientPid}`: Monitors the new client PID and adds it to the participant list.
  - `{post_message, ...}`: Broadcasts the message to all Erlang participants and sends it to the Java node.
  - `{'DOWN', ...}`: Removes a crashed or disconnected client from the participant list.
  - `stop`: Closes the chat when the auction ends and notifies all participants.

The following image contains the server organization:



```
ErlangServer
    chat_handler.beam
    chat_handler.erl
    DS_auction_handler.beam
    DS_auction_handler.erl
    DS_auction_manager.beam
    DS_auction_manager.erl
    DS_auction_monitor.beam
    DS_auction_monitor.erl
    DS_auction_supervisor.beam
    DS_auction_supervisor.erl
    erl_crash.dump
    OtpErlang.jar
```

4.2 Client-Side Implementation

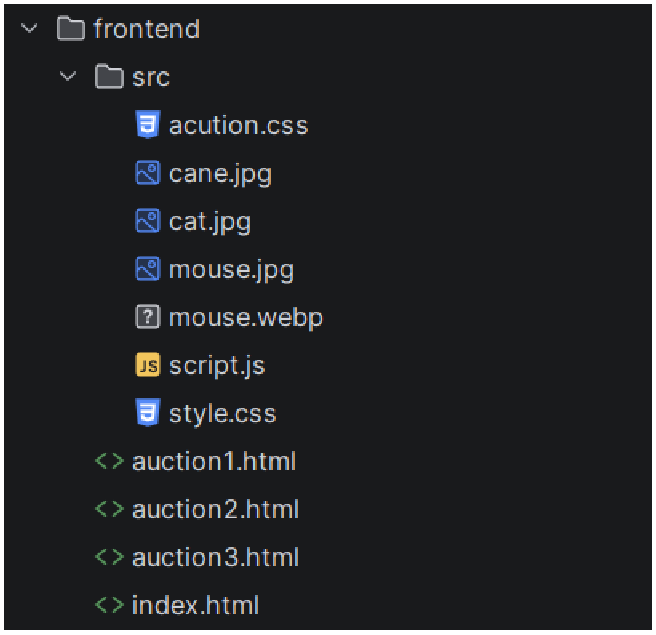
The client side was implemented using SpringBoot MVC, a HTML based Frontend and a MySQL type of database.

4.2.1 Front-end

The front-end layer was custom-developed to realize a specific visual identity and user experience. It is currently only used as a mock-up in this project.

The front-end resources are organized within the following directory structure:

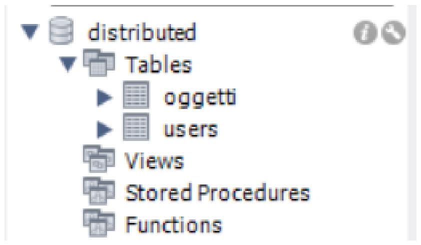
- **frontend/**: Contains the core HTML structures (`index.html`, `auction1.html`, etc.) that define the skeleton of the various application views.
- **frontend/src/**: It stores the presentation and static logic, including:
  - `style.css` and `auction.css`: Custom stylesheets used to define the layout, responsiveness, and aesthetic themes.
  - Static images and media used to represent auction items and elements.



```
frontend
    src
        acution.css
        cane.jpg
        cat.jpg
        mouse.jpg
        mouse.webp
        script.js
        style.css
    auction1.html
    auction2.html
    auction3.html
    index.html
```

4.2.2 MySQL Database

As stated previously, a relational database was implemented via MySQL.
The database schema is designed to ensure data integrity and efficient retrieval of information regarding the two primary entities: users and items.



- **Users**: with their attributes, such as name, surname, username (unique), email, age, and payment information
- **items**: with their attributes such as title, description, starting price, photo, type, and status.

Users have a **unique** username, and it will be shown during the auctions and the status of the items is updated every time they are used:

- **PENDING**: While it is waiting to be selected
- **ACTIVE**: When its auction is ongoing
- **SOLD**: When the auction ends and the item is sold

4.2.3 Spring Boot MVC: RESTful API and HTTP Requests

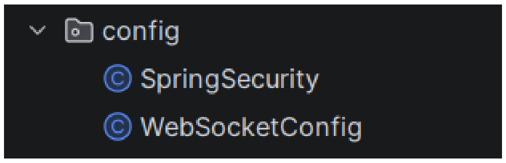
Since springBoot MVC was used for the project, client interactions are handled with RESTful HTTP API , using Spring Boot controllers

- User actions that require immediate validation, such as user authentication, fetching the initial list of active auctions, or submitting a new bid—are executed via standard HTTP `GET` and `POST` requests.
- For example, when a bid is sent, the client sends an HTTP `POST` request to the `AuctionController`. The controller delegates the operation to the `ErlangService` (which performs the RPC call to Erlang) and keeps the HTTP connection open until it can return a success or error HTTP status code to the user.

4.2.4 Config files

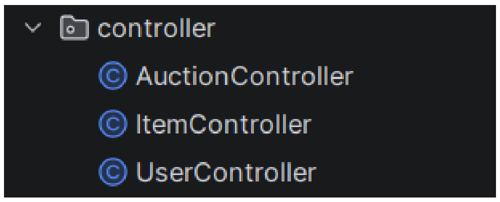
In order to give our web application a security layer, SpringBoot security was implemented.

- `SpringSecurity.java`: Configures the security layer, defining authentication rules and authorized access to specific URLs (for example, login, registration, and auction pages).
- `WebSocketConfig`: enables real-time communication between the server and connected clients. It allows clients to subscribe to auction events (bid notifications, status changes, etc.) and send actions (place bids) without constant polling.



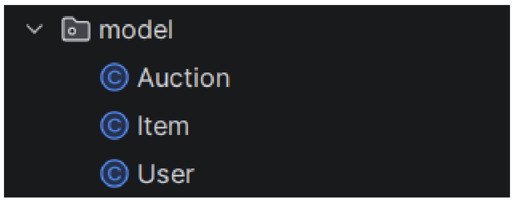
4.2.5 Controller

- `AuctionController.java`: Manages the requests related to the auctions, coordinating the display of active auctions.
- `ItemController.java`: Provides endpoints for managing auction items, including the creation and retrieval of the items.
- `UserController.java`: Handles user related operations such as registration and authentication flows.



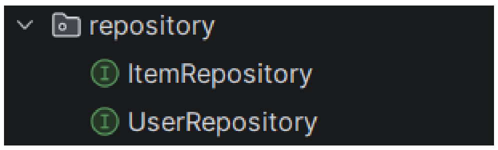
4.2.6 Model

- `Auction.java`: Represents the auction entity, containing details like current price, time remaining, and histories for both bids and chat messages.
- `Item.java`: Defines the structure of the item being sold, including its name, starting price, and status.
- `User.java`: This class manages user information, including credentials and roles for system access.



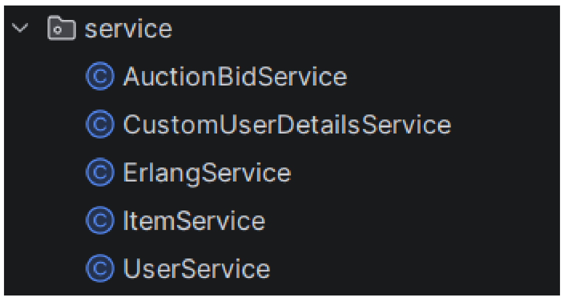
4.2.7 Repository

- `ItemRepository.java`: Interface for performing CRUD (Create, Read, Update, Delete) operations on the `Item` database table.
- `UserRepository.java`: Manages persistence logic for `User` entities, allowing the system to find and save registered participants.



4.2.8 Service

- `ErlangService.java`: The core layer that manages the JInterface node, listening for Erlang messages and sending commands to the Erlang server.
- `AuctionBidService.java`: Handles the synchronization of auction states between Erlang and Java, managing the local cache of bid and chat histories.
- `ItemService.java`: Contains business logic for items, such as activating the next pending item for a new auction slot.
- `UserService.java`: Implements user-specific logic, including password encoding and user retrieval.



4.3 Erlang-java communication

The system architecture uses an hybrid approach, with Java Spring Boot and the actor model of Erlang.
This interoperability is facilitated by the **JInterface** library, which enables the Java application to function as a native node within the Erlang distributed network.

4.3.1 Node Configuration

The connection between the two environments is made with standard Erlang node handshake:

- The Java runtime is identified as `java_node@10.2.1.25`, while the Erlang backend operates as `auction_service@10.2.1.48` and the worker on `10.2.1.13`.
- **Authentication**:There is a shared secret (*cookie*) for the session.
- **Mailbox Management**: Java initializes a persistent `OtpMbox` named `java_listener`.

4.3.2 Synchronous and Asynchronous Message Passing

Communication uses both synchronous RPC-style calls and asynchronous message passing, depending on the required interaction:

- When a user submits a bid, the `ErlangService` creates a temporary mailbox, constructs an `OtpErlangTuple` (with user ID, bid amount, and username), and sends it to the `DS_auction_handler`.
- Java waits for a definitive `bid_accepted` or `bid_rejected` response before replying to the HTTP request, depending on the cases discussed in the previous paragraphs and auction rules.
- For state changes (like auction closures, new chat messages, or timer extensions), the Erlang backend sends tuples directly to the `java_listener` mailbox.
- A dedicated Java thread continuously polls this mailbox, allowing the system to persist results via the `ItemService` and broadcast real-time updates to the clients without blocking the main application flow.

4.3.3 WebSockets

To guarantee instantaneous state propagation and eliminate the network overhead and latency that happens with traditional HTTP polling,WebSocket protocol was used, to prevent constant pooling. This model allows the backend to push updates (such as new bids or chat messages)without having to refresh the page each time something happens, keeping the session open until it finishes.

4.3.4 Clock Synchronization

Given the distributed nature of the application, there could be clock drift between the client's local machine and the Erlang server that can lead to discrepancies in the auction countdown timers.

To mitigate this synchronization issue, the system implements a streaming of the Erlang time to Java so that all clients will be able to see the server's time instead of their local times.

5 Resolution of Synchronization Issues

As stated in the previous paragraph, these are the summarized synchronization and communication issues and how they are resolved and handled by our application.

- **Race Conditions on Bids**
- **Time issue** All users must have the same coordinated time
- **Bids** All user must see the valid bids and their history
- **State Propagation**
- **Sequential bids**
- **Winner determination when auctions ends** nodes when timer ends.
- **Message Ordering in Chat and Messages at the same time**

5.1 Solutions

The solution for these problems in concurrency and synchronization is handled by the Erlang language and the WebSockets.

The core logic is implemented entirely in Erlang where each auction is encapsulated within a distinct process (`DS_auction_handler`). Erlang processes possess a mailbox where incoming messages (such as bids) are queued. The process handles these messages sequentially in a standardized loop. Consequently, race conditions are eliminated natively by the language: even if two bids arrive at the exact same millisecond, they are serialized by the mailbox and processed one after the other.

The system acts as a centralized source of truth to solve synchronization issues because the Auction state is handles also in erlang and it rejects invalid logic locally before the state is propagated. The Erlang handler acts as the master clock. It decrements the remaining time internally and provides a synchronization API implements **Cristian's Algorithm** to align client clocks with the server, compensating for network latency. When the timer ends elrnag also selects the winner of the auction and notifies all java nodes.

On the other hand, the commuication and propagation to clients is handled via the Java Spring node. The Erlang Manager and Handlers give updates to the Java node, like bids, chat and winner, via `JInterface`. Then the Java service uses **WebSockets** to immediately broadcast these updates to all subscribed frontend clients.

This architecture ensures that while clients may be distributed, they all visualize a view of the system that is consistent with the serialized logic executed by the Erlang server.

6 User Manual

In the following section there are the different options and actions that can be done in the web-application and the rules we set for the auction application and handling.

6.1 Auction Rules

For the Auction service, the following rules are implemented:

- Registered users can login and participate in an Auction.
- When the auction starts a timer is also set and users can bid on the item as well as write messages in the chat
- When bidding as a registered and logged user:
  - A user can't make two consecutive bids, the second one will be rejected
  - if the bid is lower than the previous one, it will be rejected
  - if he/she tries to bid after the time ends, the bid will be rejected
  - if the bid is higher than the previous one it will be accepted
  - if the bid is done in the last 30 seconds of the auction, the timer will be reset
- The auction winner will be displayed at the end of the auction and will be unique
- Non registered user can't bid or write in the chat

In the following section some mock-ups of the application are shown

6.2 Home page

This is the homepage of the system as soon as you open the web application.

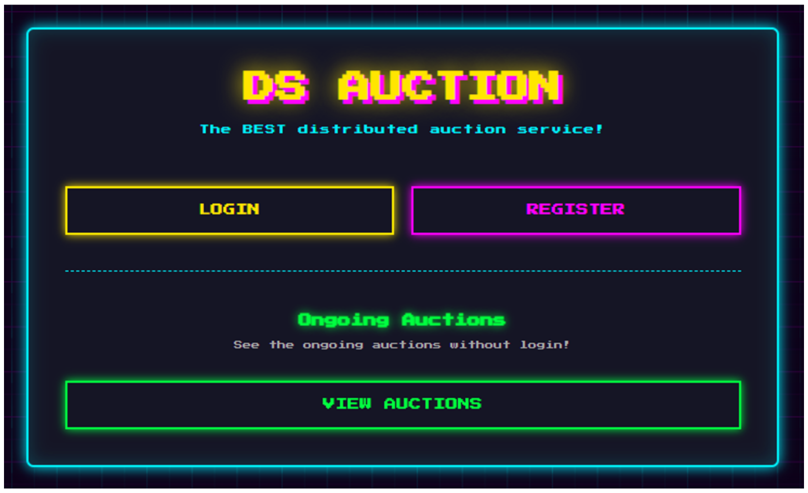

DS AUCTION
The BEST distributed auction service!
LOGIN
REGISTER
Ongoing Auctions
See the ongoing auctions without login!
VIEW AUCTIONS

Figure 1: Homepage

6.3 Registration and Login Phase

Users can then register in the **registration page**, which includes all the mandatory fields that a user must insert to register correctly:

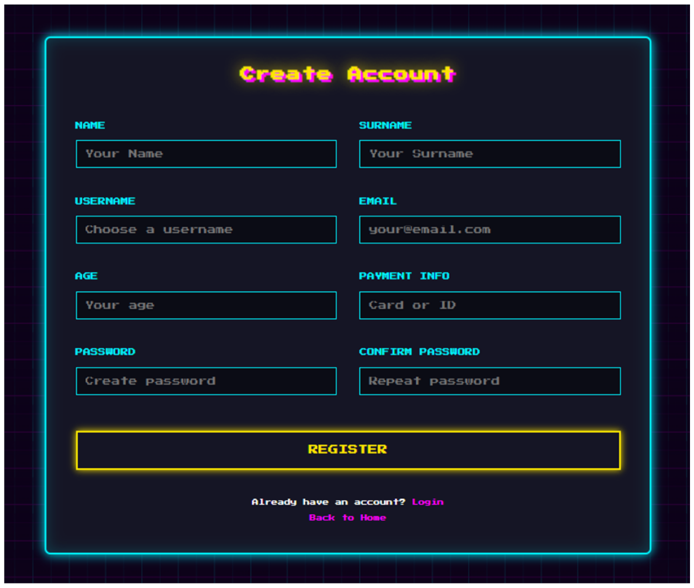


Figure 2: Registration Page

After registration, users can **login** with username (which needs to be unique) and password so that he/she can have access to the chat and auctions.

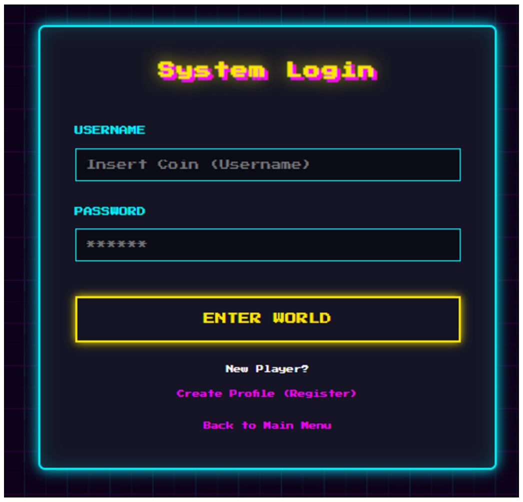


Figure 3: Login page

6.4 Browsing and Creating Auctions

Later on he/she can view the ongoing auctions and decide if he/she wants to participate or not in a specific auction.

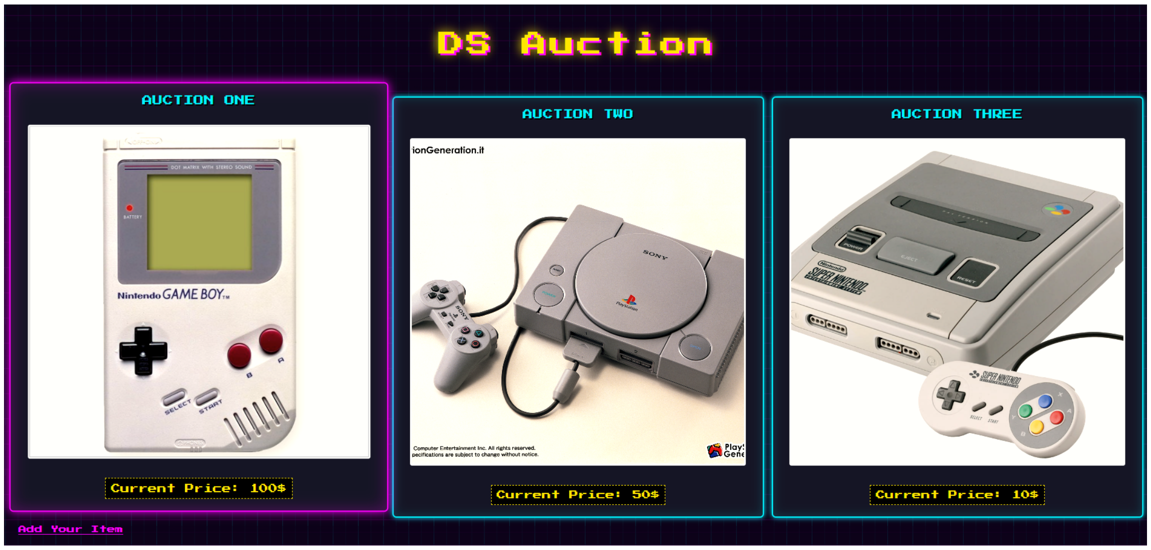


Figure 4: Auction List

A user can also create its own auction by compiling its dedicated section, filling all the necessary details.

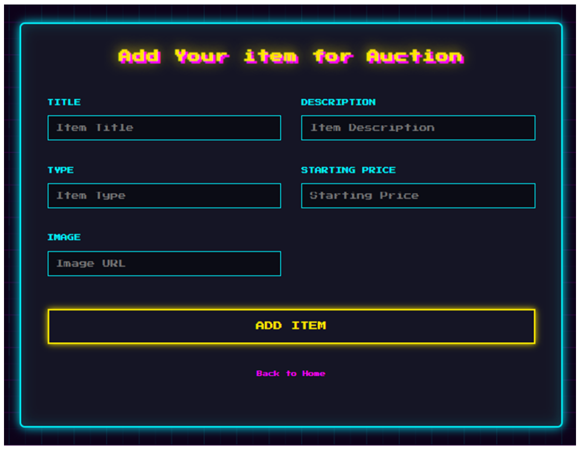


Figure 5: Enter an item to be auctioned page

6.5 Participating in an Auction (Bidding & Chat)

This is how an ongoing auction looks, there is the chat on the left, the bidding button, exit button to exit the auction and the history of the bids.

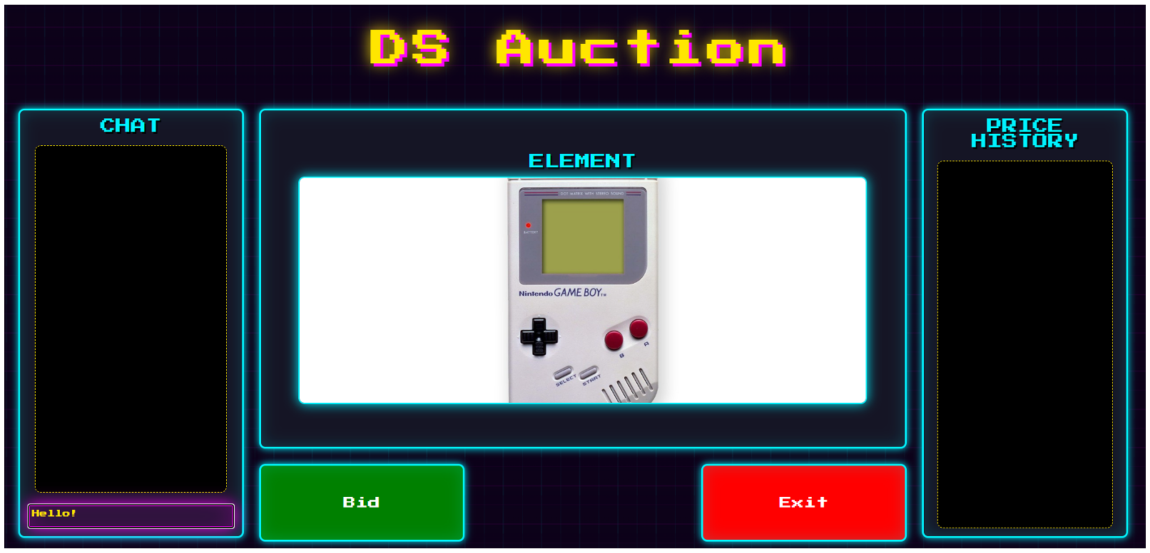


Figure 6: Enter Caption

6.6 Auction Conclusion and Winner Election

At the end of the auction you can see a new page where the winner is announced and the details of the auction are shown.

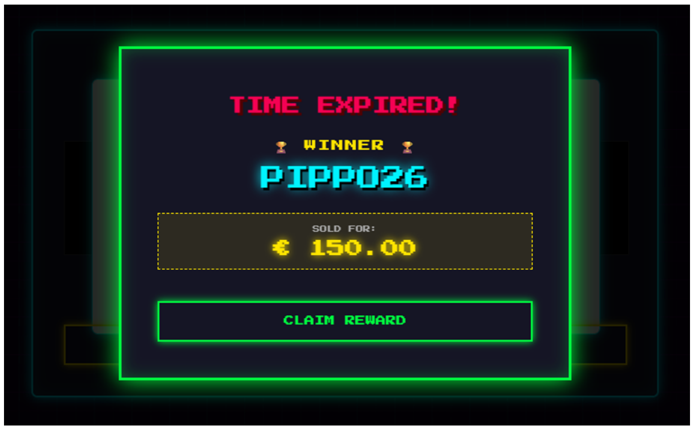


Figure 7: Enter Caption