Guess Who?

Andrea Biagini
andrea.biagini5@studio.unibo.it

Filippo Gurioli
filippo.gurioli@studio.unibo.it

April 2024

The project aims to create a distributed system that manages the creation and progress of matches of the popular "Guess Who" game. Players can join the game using a desktop application that connects to a dedicated server, allowing users to instantiate matches. A user can register to save statistical data, such as the number of wins. The server will be internally divided into multiple services that appear as a single entity to clients. This way, a certain level of *fault tolerance* and *availability* is ensured: if one of the running services fails, another will take its place, and meanwhile, other users can still start new games. Each new game will be instantiated on the first functioning service that is currently hosting the fewest games. User data will be replicated multiple times on dedicated data structures that guarantee *consistency*, again to provide *fault tolerance*.

1 Goal

The goal of this project is to develop a distributed system composed by two distinct entities: the client and the server. More specifically, the application must be able to replicate data using multiple servers. This case of study is about "*Guess Who?*"; it has been chosen for the simplicity of its mechanics, that make it possible to the team to focus more on design and architecture aspects.

1.1 A brief description

Guess Who? is a known table game in which two players compete to guess the opponent's secret character. Every player has a table containing faces of the different characters. Alternatively, players ask *yes/no* questions to exclude characters in their table, trying to identify the opponent's secret character before him or her. The player who guesses the opponent's character first wins the match [3].

1.2 Requirements

To play a game, a player must use a client application from which it is possible to:

- log in or sign in to keep track of personal results;
- create a match;
- join a match;
- ask questions to the opponent to progress the game;
- answer to the opponent's questions;
- interact with cards representing every character, to isolate the suspects;
- send *guesses*, particular questions containing a character's name who the player wants to guess, that can lead to his or her victory or defeat.

In this scenario the server is totally transparent to the users, and it acts as coordinator to maintain a valid state of the match and to allow players to play it. Figure 1 highlights the user's use cases related to what has been just explained.

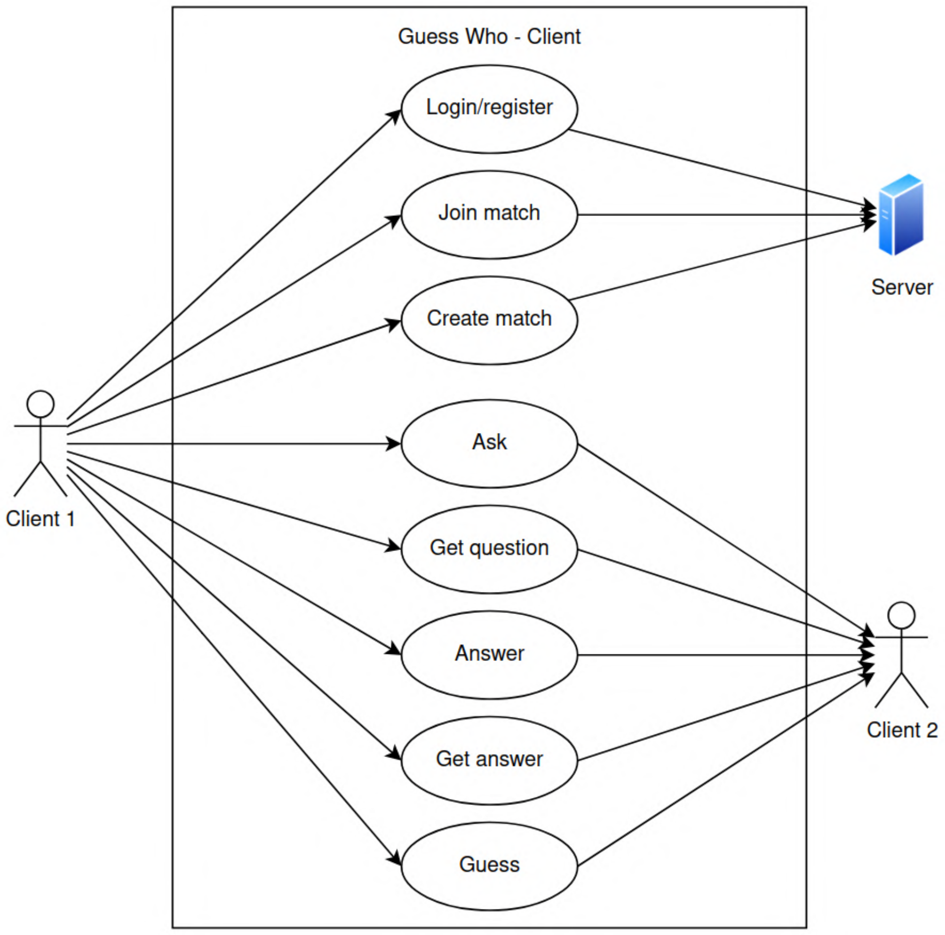


Figure 1: Use case diagram from the user's point of view.

1.3 Non-functional requirements

From the just mentioned requirements come the non-functional ones. More than one machine will be required to handle the server, to guarantee separation of concerns and scalability, and every actor in the system must be inter-connected. All edge cases must be also considered to give solidity to the system:

- what happens if a player tries to join a non-existing match?
- what happens if a player sends two consecutive questions?
- what happens if an answer is sent outside of a player's turn?

These questions find answer in the design (2) section and in the paradigms and architectures used in the project.

2 Design

In this part the design of the project, the specific structures used and the actors behavior is explained. To organize game servers and to satisfy the scalability requirement a *proxy* architecture is used, in order to an intermediate component between clients and servers. To guarantee data replication, the same principle is adopted, with a *gateway* component that redirects client calls to the internal servers. In conclusion, besides the *frontend-backend* division, another separation is applied inside the backend part. To keep things modular, backend is divided into **server**, concerned about the dynamic part of the system, and **serverData**, concerned about static resources. The former knows everything about user interactions regarding the game matches themselves, like question or answer requests, characters assignment and so on, while the latter maintains data about matches and users, like questions and answers history, data for every user, and so on.

Figure 2 summaries the just described design.

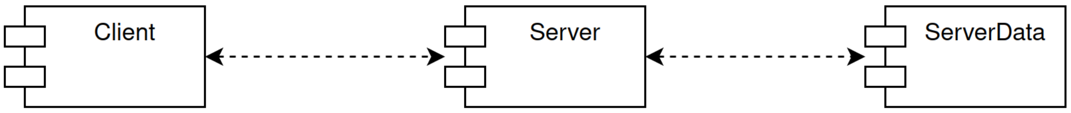


Figure 2: Macro-entities diagram

2.1 Interactions

It is now illustrated interactions modeling between entities. Gradually, client-server interactions are explained, and then the proxy-servers and gateway serverData ones.

Given that the current domain is hardly message-based, the aspect of interactions was the first one considered. To maintain an incremental approach, the first set of interactions designed is the one related to the match itself. The set regarding user login and registration and other sets of interactions are considered later. As shown in Figure 3, the initial design was composed by little and simple interactions:

- `create` - creates a match and returns a unique code identifying it;
- `wait other player` - makes the player wait for a *join* action from the other player;
- `join` - lets a player join a match given its unique code;
- `send message` - sends a message to the other player, that can be of three kinds: *question*, *answer* or *guess*;
- `wait message` - makes the player wait for an *answer* message from the other player.

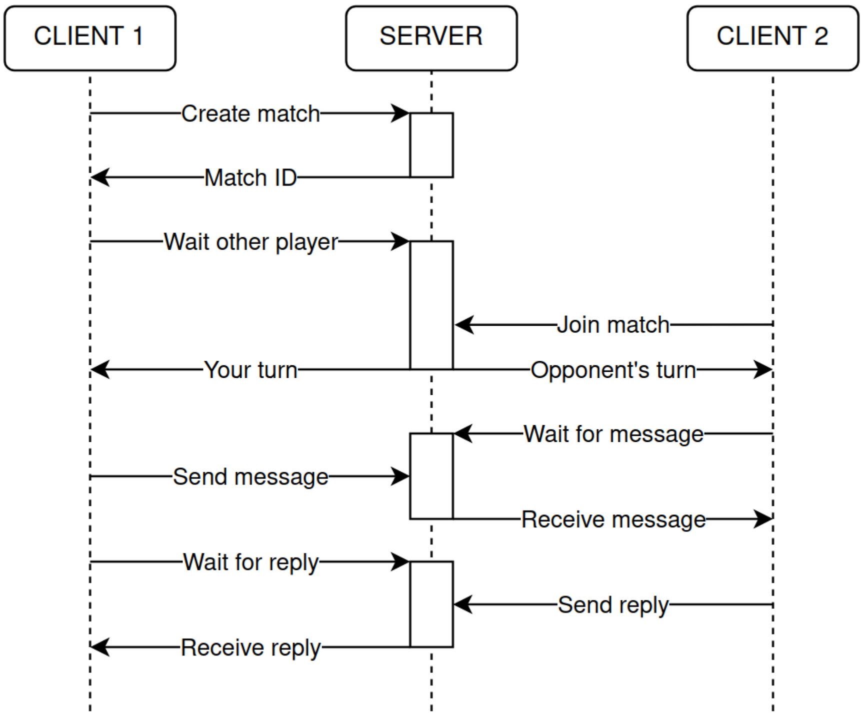


Figure 3: First draft of interactions between clients and server.

This first and simple part of the analysis already gives bases to the final result.
A relevant problem has been immediately detected, regarding the use of synchronous interactions: a client who sent a question has to keep the communication open until the other client sends his or her request. Keep requests open between a client and the server is generally considered a bad practice, as it leads to less security and waste of resources. From this partial representation various variants took place, until the final result has been reached. The final result is illustrated in Figure 4.

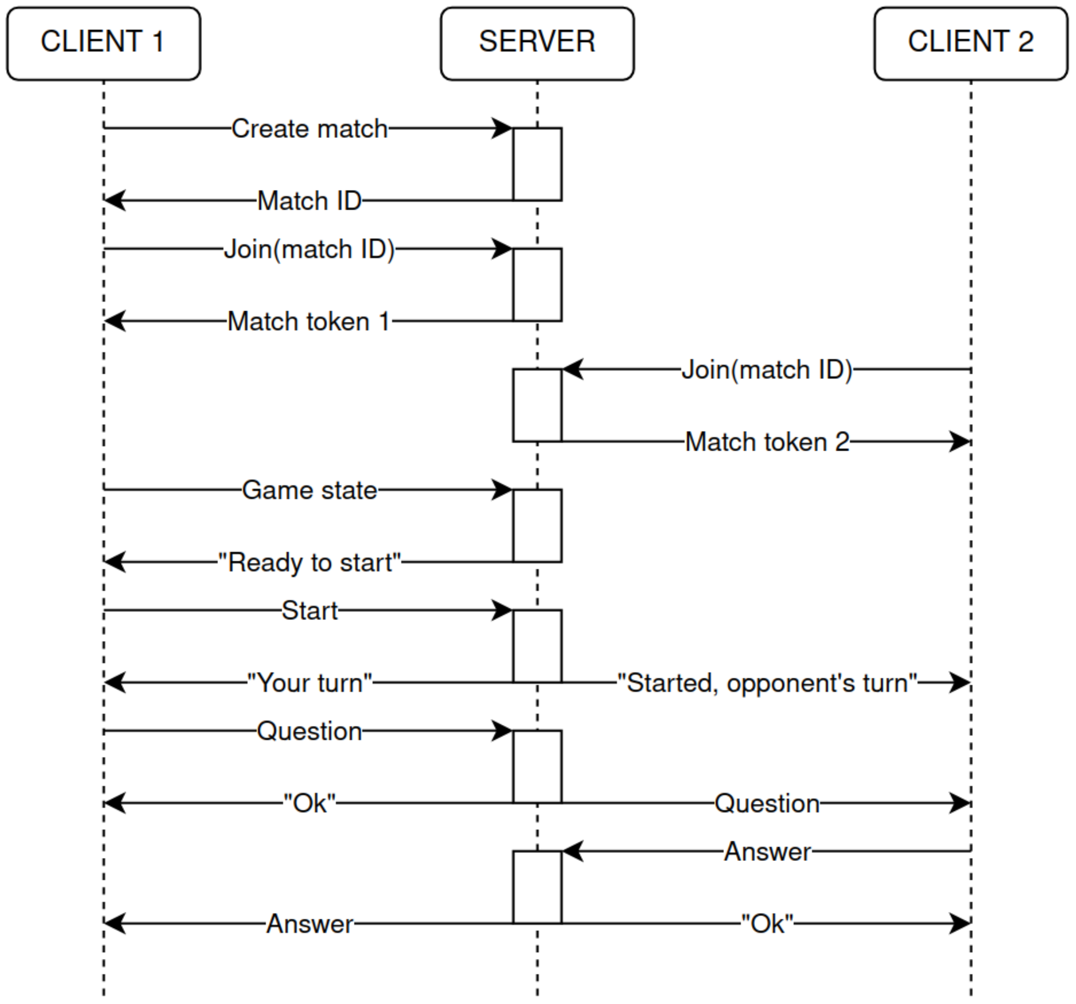


Figure 4: Final *Sequence diagram* of the application

To obtain a cleaner diagram, polling actions are left implicit, while some non-trivial responses from the server. The application performs polling requests to be kept updated about the current state of the match; it can be assumed that each client knows it at every moment.

Some server responses contain a *match token*, an alphanumerical string, unique in the context of the match, that identifies the two players, and makes the server able to handle them correctly. It is generated and communicated when a player joins a match, and it is used in client requests to let the server know from which of the two player the message is. Also, a client that does not know any of the two match tokens can't interact with the match.

Another token is additionally used, for what concerns the user login; this is the *login token*, a string ciphertext (encrypted with a secret key) whose plain-text is basically a JSON containing a username and the date and time until which its login is valid. This token is attached to requests that need it using a custom HTTP header; the server decrypts the login token and checks the expiration date: the request process continues only if this check succeeds.

Regarding the interaction between proxy and server, an effort has been made to distribute the load of requests by dividing matches across multiple servers that handle the computation of responses. In this architecture, the proxy assigns matches to one of the servers and redirects requests based on a map that associates each match's identifier with the server to which it was assigned. The proxy also manages login and checks the login token to filter out any invalid requests. To assign a match to a server, the proxy generates a list of all servers sorted by the number of games already assigned (in ascending order). It then selects the first server in the list and checks its availability by performing a ICMP ping request: if the server responds, a match is created on it, and the *matchID-serverID* entry is saved in the proxy's map; if the server is offline and does not respond, the next server in the list is chosen, and the process is repeated.

This architecture aims to distribute the load of requests; however, it still presents the challenge of having the proxy as a central element, without which the system does not function. Despite this, in the event of a proxy crash, users may experience only a brief period of service unavailability, and in the *worst-case scenario*, they may need to log in again (if the newly restarted proxy has not set the correct date and time). All matches data remains intact, as it is managed by other services outside of the proxy.

Lastly, it is important to specify how interactions between servers and the data server occur. As mentioned, a gateway is used as an intermediary between the two entities. Specifically, it can mask the IP addresses of the various serverData, displaying only its own in the public network. The immediate advantages include increased security, as the ports for the servers are not directly exposed, as well as cleaner organization, with a single IP address for sending requests. However, this also brings some disadvantages, such as an evident bottleneck at the gateway itself. Since all requests must pass through this structure, it is logical to assume that it is easier to saturate it, which could cause the system to fail. There are both theoretical and practical solutions to this problem, although they will not be implemented in this project. For a brief analysis of these solutions, please refer to the section "future work" 6.1.

Figure 5 illustrates how the gateway and serverDatas interact.

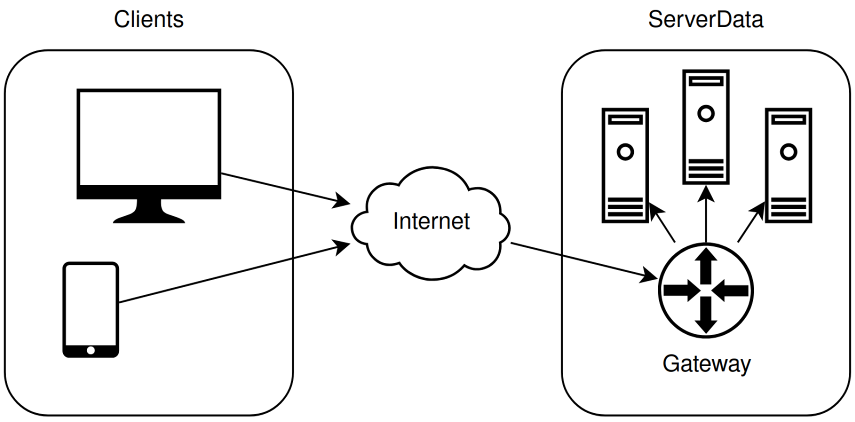


Figure 5: Interactions between the gateway and serverDatas.

2.2 Structure

As previously specified, to enable the two players to communicate and progress in their match, a client-server structure was built to coordinate the players during the match. If a peer-to-peer structure had been chosen, issues related to data access and protection would have arisen, which are avoided with the adopted structure. Thanks to the server, data is entirely delocalized and accessible only through client identification, ensuring security and integrity. However, the chosen architecture also presents some challenges: using a server creates additional overhead for communication between users, as it must always pass through a medium entity.

Having clarified the reasons behind choosing a client-server structure, the modeling of the entities involved is presented, as briefly mentioned at the beginning of this section.

2.2.1 Backend

First, the backend is analyzed. Being an essentially independent module, it is easier and more immediate to describe, as it is possible to abstract from the actual implementation of the frontend. Its role is, in fact, to expose APIs for managing interactions between clients, as well as for handling static resources.

The APIs have been divided into 3 main groups:

- **Users - API**: allows managing user information, such as registration, login, modification of personal data, etc.
- **Manage - API**: allows managing matches, such as creating a new match, adding a player to a match, removing a player from it, etc.
- **Game - API**: allows managing the match itself, such as sending questions, answering questions, starting the game, etc.

For each of these main groups, the main routes that allow access to the various functionalities have been listed:

1. **Users**:
   - `POST /register`: to register a new user;
   - `POST /login`: to log in;
2. **Manage**:
   - `POST /create`: to create a new match;
   - `POST /join`: to join a match;
   - `POST /leave`: to leave a match;
   - `GET /getAllMatches`: to get the list of currently created matches;
3. **Game**:
   - `GET /state`: to get the current state of the match;
   - `POST /question`: to send a question;
   - `POST /answer`: to answer a question;
   - `POST /guess`: to make a guess;

Once the APIs to call and their corresponding responses were established, the focus then shifted to modeling the client.

2.2.2 Frontend

On the client side, a component-based architecture has been adopted. This allows for a modular structure, making it easier to add new features and modify existing ones. It is important to emphasize that a component structure does not necessarily have to map the server APIs *one-to-one*; there will be components that call multiple APIs as well as components that do not call any. The function of the frontend is to display the game to end users and to allow them to interact with it in the simplest and most intuitive way possible. With these premises, the team wishes to not focus too much on the graphical aspect, but rather to concentrate on the interactions between the various components and the server.

The flow of the frontend application has been divided into 3 phases:

1. **Login/Registration**: allows the user to register or log in;
2. **Lobby**: allows the user to create a new match or join an existing match;
3. **Game**: allows the user to interact with the actual match, sending questions, answering the opponent's questions, eliminating suspects, etc.

By the end of the third phase, the intention is for the game not to end but to return to the *Lobby* phase, allowing the user to create a new match or join an existing one.

In figure 6 the model of the game flow is represented.

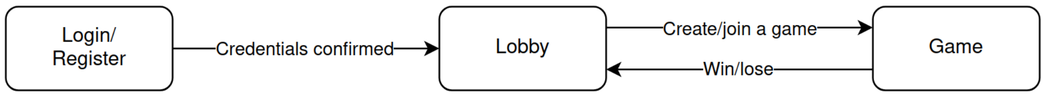


Figure 6: Model of the client-side game flow

These 3 main components will then be decomposed into sub-components, which will actually implement the calls to the server. In the register/login component, for example, there will be server calls for registration and login, while in the Game component, there will be calls to send questions, answer the opponent's questions, etc.

For a more detailed description of the calls to the backend, please refer to section 3.

2.2.3 Backend - Data

As previously mentioned, to ensure greater robustness of data, the replication functionality has been separated into a distinct module. This section will not address how this functionality is guaranteed; for that, please refer to the section ”Interactions” (2.1). Instead, we will outline which data is replicated and how it can be accessed.

To manage data, a minimal API will be used, allowing for simple and fast access. Therefore, a REST API has been chosen, enabling data access through HTTP calls.

The data that will be replicated are:

- **Proxy data** - data related to users, such as passwords, emails, and names;
- **Matches data** - data related to server management, completely transparent to users but useful for managing matches. This includes information about which server hosts which match and how many matches have been created;
- **Matches Server data** - data related to match management, such as its current state, the questions asked, the answers given, or the current turn.

Once established which resources we want to track, the team proceeded to model the REST APIs, outlined below.

Under the route `proxydata`, there are:

- `GET /getUser`: allows retrieving user data;
- `POST /registerUser`: allows registering a new user;
- `DELETE /deleteUser`: allows deleting a user;

Under the route `matchdata`, there are:

- `GET /getMatch`: allows retrieving data of a match, i.e., all data related to the association between matches and the servers hosting them;
- `GET /getAllMatches`: allows retrieving the list of currently created matches;
- `POST /registerMatch`: allows creating a new match;
- `DELETE /deleteMatch`: allows deleting a match;

Finally, under the route `matchserverdata`, there are:

- `GET /getMatch`: allows retrieving data related to a match, specifically the current state of the match;
- `POST /registerMatch`: allows creating a new match;
- `DELETE /deleteMatch`: allows deleting a match;
- `GET /getPlayers`: allows retrieving the list of players currently in a match;
- `GET /getTurn`: allows retrieving the current turn;
- `POST /changeTurn`: allows changing the turn of the game;
- `POST /startMatch`: allows starting a match;
- `POST /addMessage`: allows adding a message to the match history;
- `GET /getMessages`: allows retrieving the list of messages from the match;

In the implementation phase of the project, small adjustments have been made to adapt the designed APIs to their *real-life* usage.

3 Implementation Details

This section will explain non-trivial software specifics.

3.1 Frontend

To implement the frontend, Unity was used, a cross-platform *game engine* that allows the creation of 2D and 3D games. It is well-suited to the needs of the project as it speeds up development and allows for the creation of complex games in a straightforward manner.

As previously described, the client is divided into 3 main groups: *Login*, *Lobby*, and *Game*.

Upon launching the application, once the "Access" button is clicked, a login request is sent with the entered credentials. If the login is successful, the client stores the login token and is redirected to the lobby. Once in the lobby, the user can:

- create a new match;
- join an existing match;

In the first case, a creation request will be sent, and upon receiving a response, the user will be guided into the Game component. In the second case, the user will be directed to a waiting component (not included in the previous list for brevity), where the player can enter the ID of a match and thus join it (if the match exists and it is not full of players).

Finally, once inside the match, the player will be able to interact with the match itself by sending questions, answering the opponent's questions, discarding suspects, and so on. To manage these interactions, the following process is followed:

1. When both users are present, the creator (marked as *owner*) will be allowed to start the game, triggering a start message that will be received by the other player;
2. Upon starting, the client will request the server to assign them their character, that is, the character that the opponent must guess;
3. Once received, the players will begin polling the server to stay updated on the current state of the match;
4. The match state indicates which player should start, for whom the client application will display an interface to ask a question to the opponent;
5. Once sent, a request is launched to the server, which will save the question asked, and through the opponent's polling, they will be notified that it is their turn to respond;
6. To respond, a "yes" and a "no" button will be displayed, which will trigger the sending of a message to the opponent;
7. The match then ends when one of the two players attempts a *guess* question, to which the opponent must respond affirmatively, or else, when a player ends its guess attempts.

4 Deployment Instructions

To start the developed system, it is necessary to have *Docker* and the *Docker Compose* utility installed, to manage the various components.

For ease of development and testing, the software has been packaged into Docker images: specifically, one image for the proxy, one for the servers, one for the gateway, and one for the data management servers. In a Docker Compose configuration file, the execution parameters for the various containers to be run are specified: thus, to start the system, it will be sufficient to use the command `docker compose up`.

A simple Makefile has been written to quickly manage the system startup, image generation, and their management on the *docker.io* platform. To download all the necessary images for starting the system, use the command `make pull`; Docker Compose is bootable using the command `make`.

5 Usage Examples

In this section, screenshots of the software in action are provided. It should be noted that the screens will only pertain to the frontend, as it is the only part with a graphical interface.

In the first image (7), the login screen can be seen, where the user can enter their credentials to access the game.

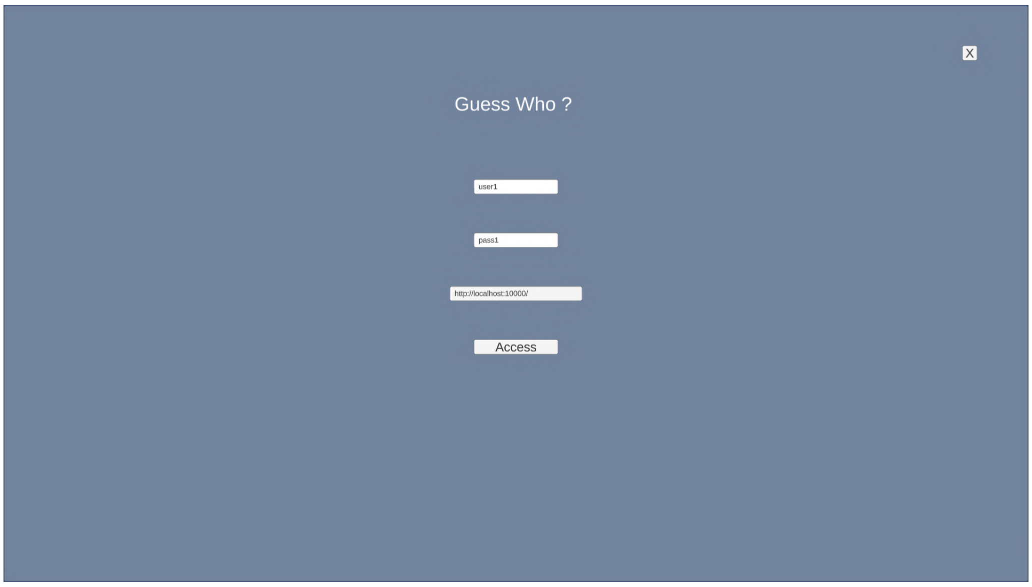


Figure 7: Login screen

Subsequent to logging in, users will be redirected to the lobby screen (8), where it can choose to create a new match or join an existing one.

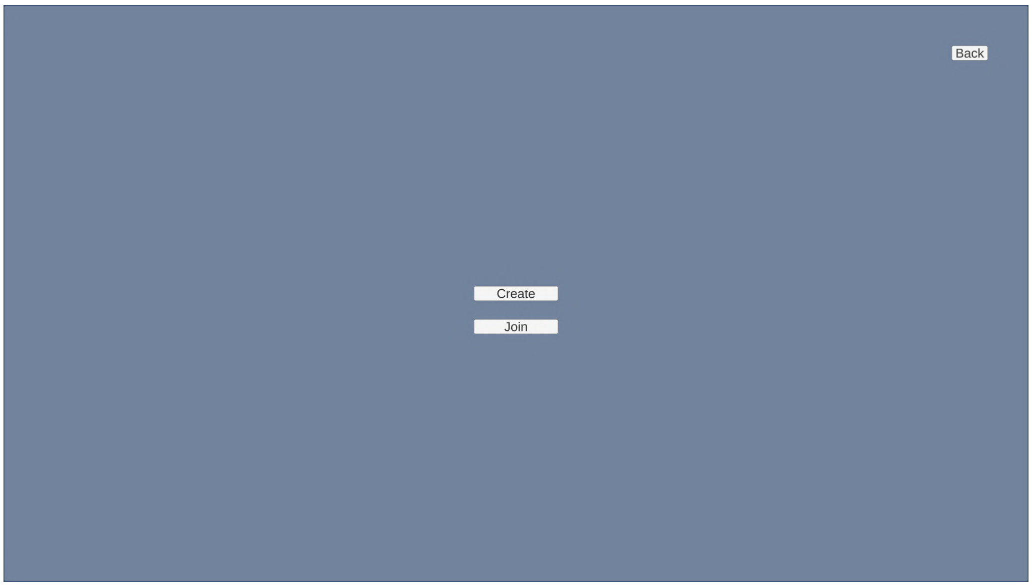


Figure 8: Lobby screen

In this example, it is assumed that the user has chosen to create a new match. Once the button is clicked, the application will transition to the game screen (9), where the user must wait for their opponent to join before starting the match.

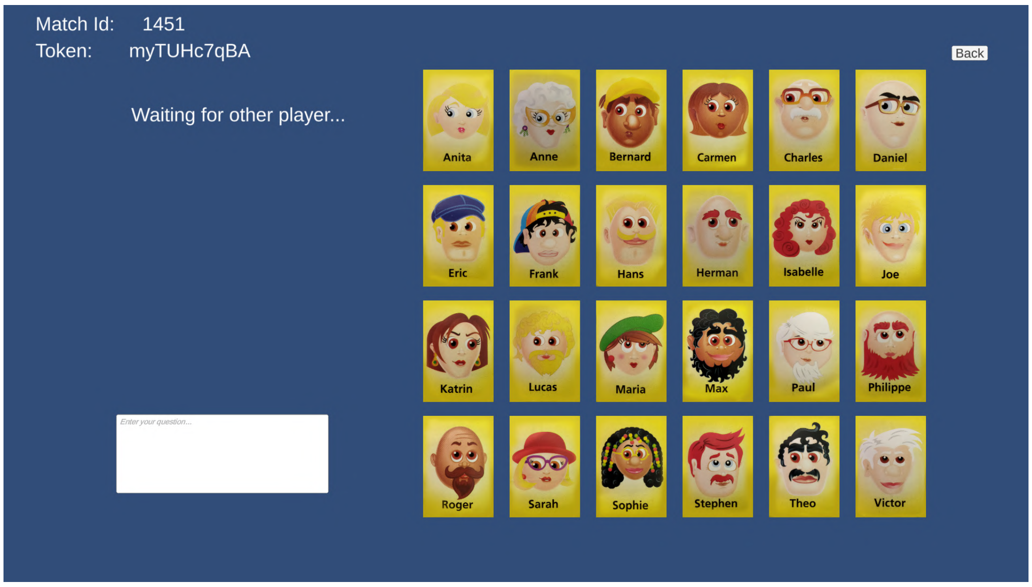

Match Id: 1451
Token: myTUHc7qBA
Waiting for other player...

Figure 9: Waiting for player screen

Once this is done, the first user to enter the match ID in the search bar in the lobby section to join a match (10) will start playing with the user.

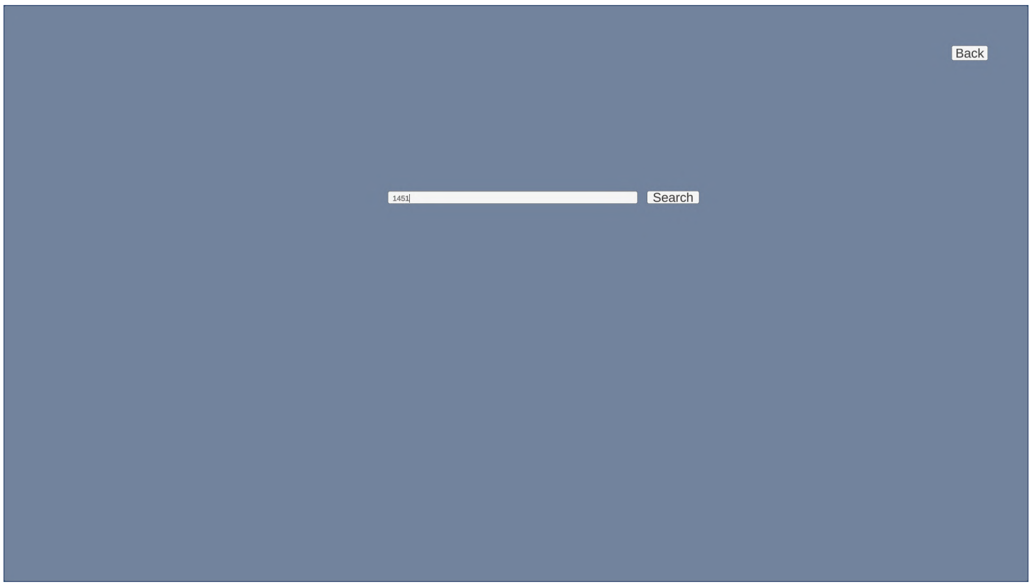


Figure 10: Join match screen

Now that both players are present, the owner can start the match (11). At this moment, the server will randomly choose which of the two players goes first, displaying the question screen (13) to the first player and the waiting screen (12) to the opponent.

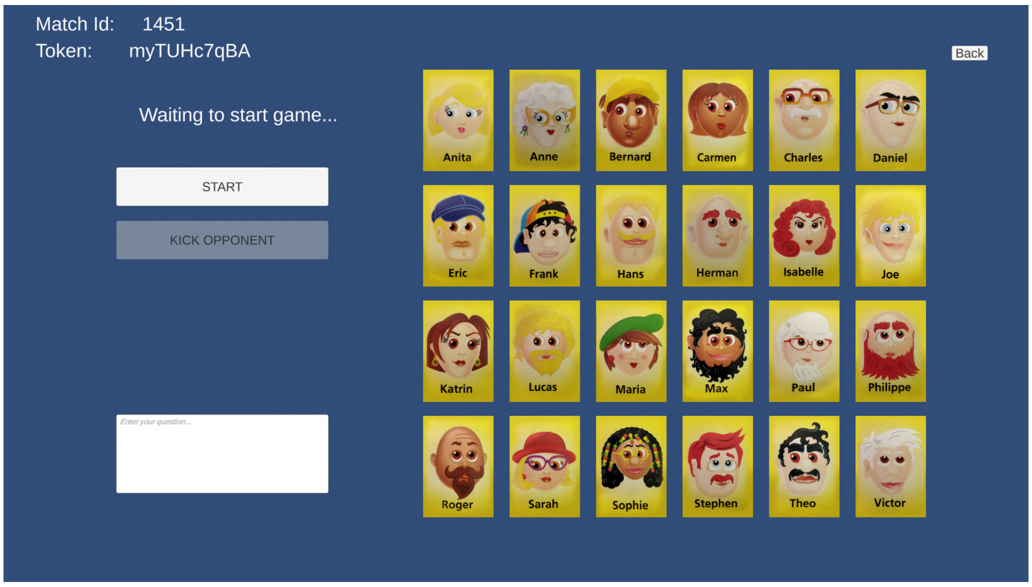


Figure 11: Waiting for match start screen

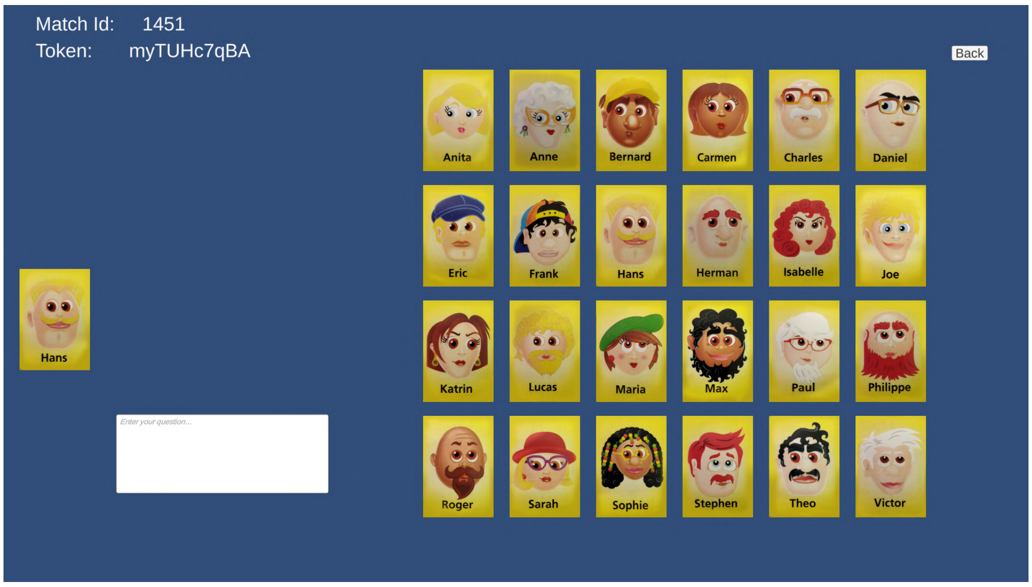


Figure 12: Waiting for turn screen

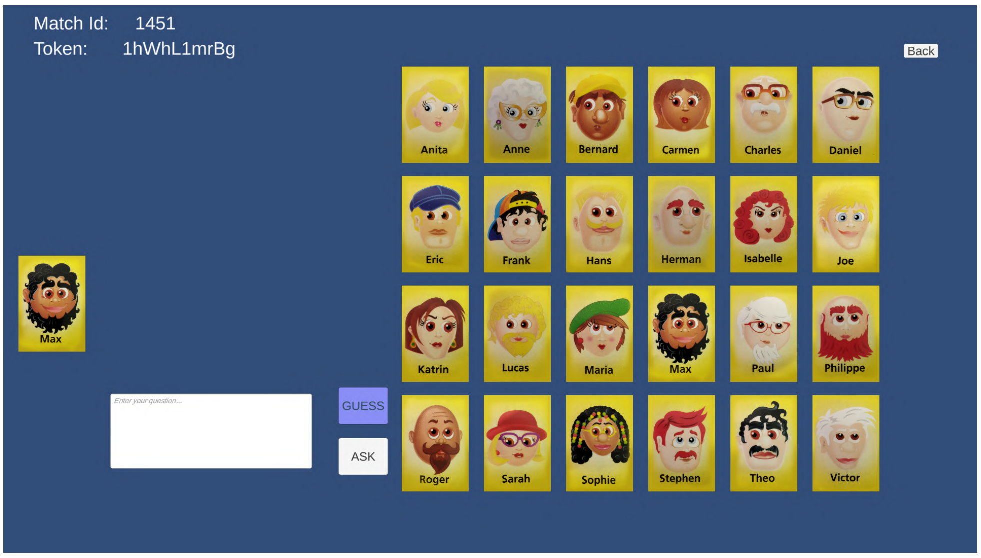
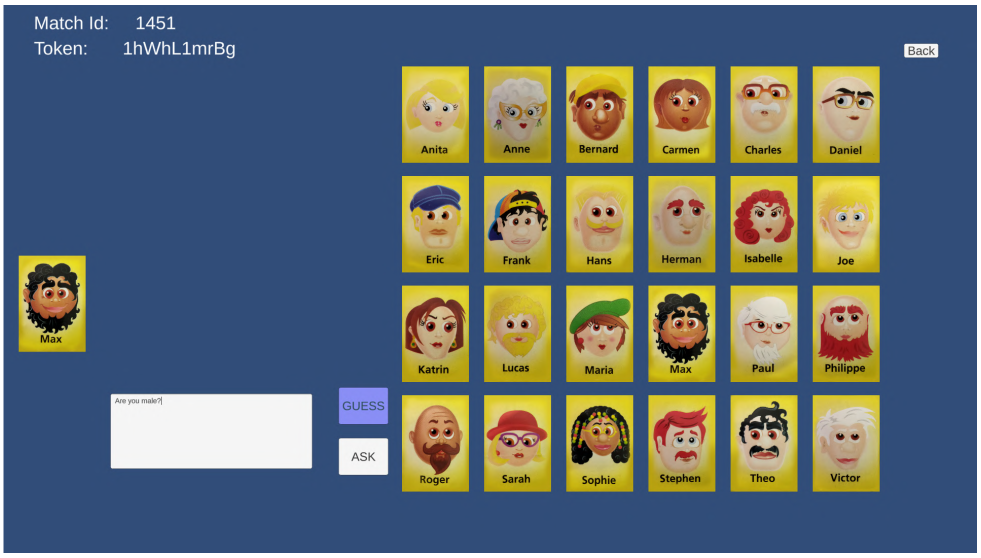


Figure 13: Question screen

The first user then enters a question and clicks the "ask" button to send it to the opponent (14). The opponent will receive the question and can respond with a "yes" or a "no", while simultaneously formulating its own question (15).
'yes'



Are you male?

Figure 14: First question

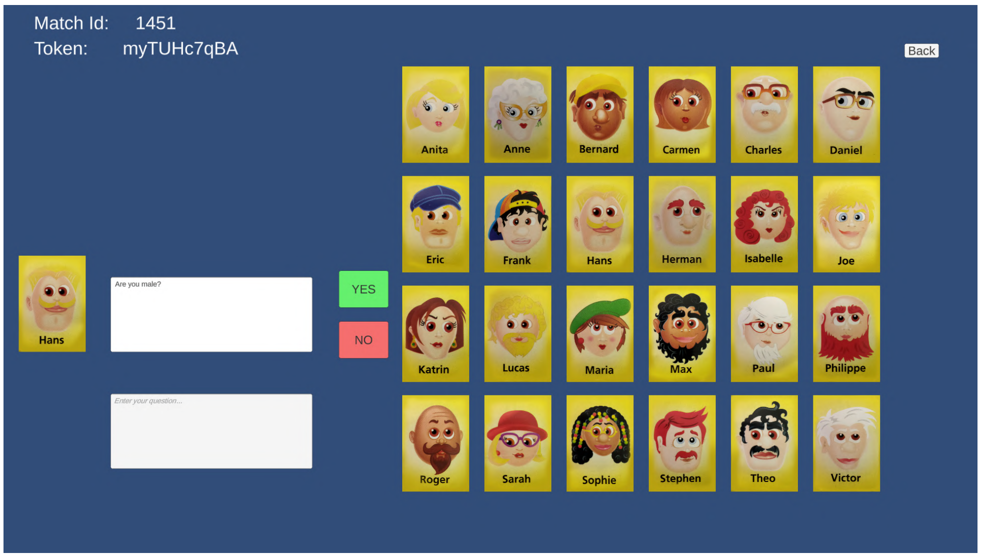


Figure 15: First answer

After receiving the answer, the first user can mark from the GUI characters not corresponding to the information obtained from the answer (16).

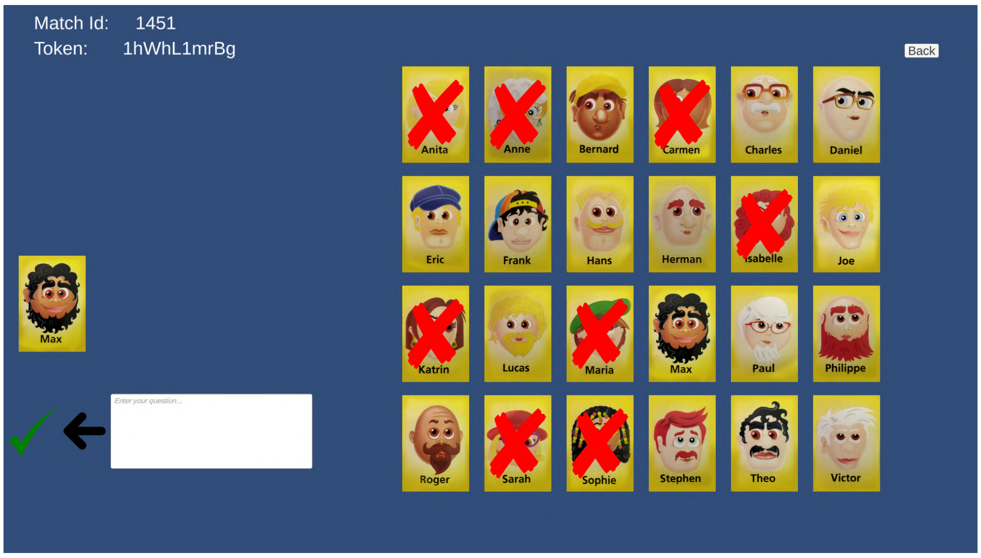
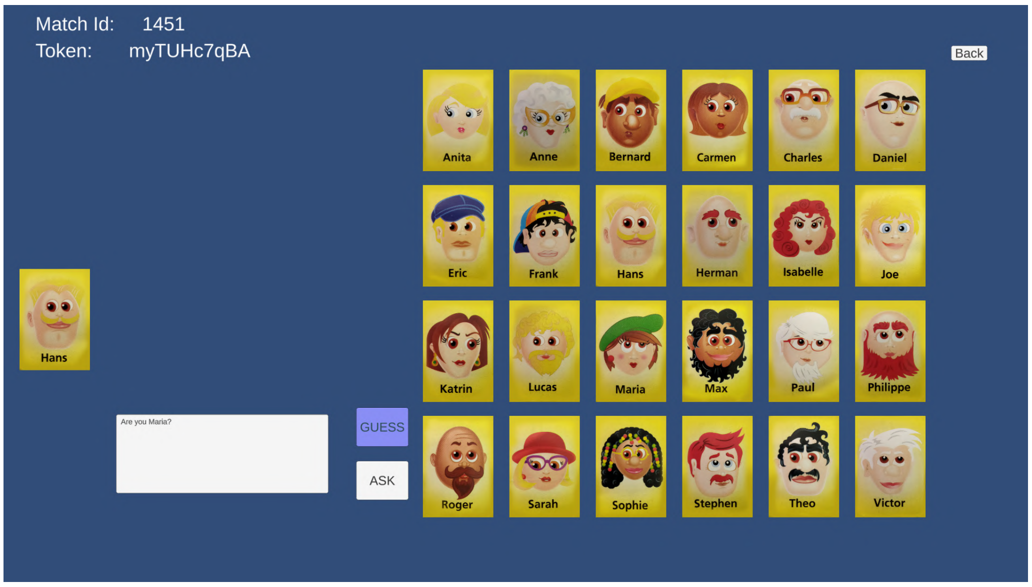


Figure 16: After the first answer

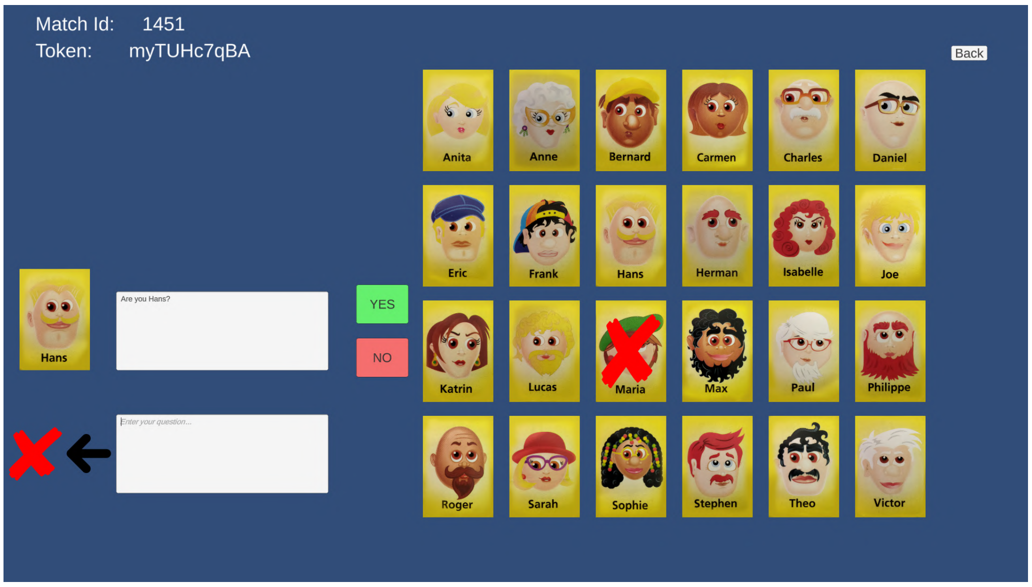
During its turn, the user can also perform a special question called *guess* (17 and 18). This question allows the player to guess the opponent's character; if the answer is affirmative, the match ends with the victory of the player who asked the question;

otherwise, if the user has not ran out of guess attempts, the match continues with the opponent's turn.


Match Id: 1451
Token: myTUHc7qBA

Are you Maria?

Figure 17: *Guess* question


Match Id: 1451
Token: myTUHc7qBA

Are you Hans?

Enter your question...

Figure 18: Answer to a *Guess* question

6 Conclusions

The project of creating a client-server application to play *Guess Who?* online has achieved significant results thanks to the implementation of a distributed architecture. This approach has highlighted several strengths, demonstrating how proper design of distributed systems can improve the efficiency, availability, and fault tolerance of the application. Below, the main advantages encountered are summarized:

1. **Separation between management servers and data servers**:
   - The separation between management servers and data servers has allowed for a clear distinction of tasks, improving the modularity and maintainability of the system.
   - Management servers are responsible for communicating with clients and data servers, acting as intelligent intermediaries, while data servers serve as centralized databases to store all necessary information.
2. **Multiple instances for each server type**:
   - The multiple instantiation of each server type has ensured greater service availability, as the presence of redundant servers allows continued operation even in the event of hardware or software failures.
   - Optimal resource management has been achieved by balancing the load among various servers, improving the overall efficiency of the system.
   - Fault tolerance has been significantly increased, reducing the risk of service interruptions.
3. **Use of a proxy for communication between client and server cluster**:
   - The inclusion of a proxy between clients and the server cluster has facilitated communication mediation and redirected requests to the least overloaded server, improving system performance and reducing response times.
   - This approach has allowed for intelligent load management, preventing bottlenecks and enhancing the user experience.
4. **Use of a gateway for communication between server clusters and data server clusters**:
   - The gateway has played a crucial role in balancing the load of requests among the servers, further improving the system's efficiency.
   - Write requests have been evenly distributed across all data servers, ensuring that each database holds the same set of information.
   - For read requests, the gateway has used a *round robin* strategy, distributing requests fairly among the data servers, reducing the load on each server and improving response times.

In conclusion, the implementation of a distributed architecture for this application has demonstrated significant benefits in terms of availability, efficiency, and fault tolerance. The division of tasks between management servers and data servers, the multiple instantiation of servers, and the use of proxies and gateways for load balancing have contributed to creating a robust and high-performing system. These results highlight the importance of careful design in distributed systems, capable of effectively responding to the needs of an interactive and real-time online application.

6.1 Future work

This section is dedicated to possible future developments and improvements that could be made to the project.

First, it is important to highlight the issue of the *bottleneck* discussed in the "Interactions" section 2.1, which is generated by the use of a central node for redirecting requests. Some of the solutions proposed in the literature to mitigate this problem include:

- horizontal scalability of the gateway, which involves the use of multiple nodes to distribute the load;
- microservices architecture, which allows differentiating between the various services offered by the gateway in order to distribute the load more efficiently;
- request caching, which involves temporarily storing responses to the most frequent requests in order to reduce response times and load on the servers.

Other possible future developments primarily concern gameplay features. Since gameplay functionality was not the main focus, the team manly concentrated on achieving the *minimum available product*, and therefore it might consider adding:

- the ability to modify the number of guess attempts allowed for each match (currently, this can be set globally by modifying a file on the server);
- a system that checks guess questions to ensure that the responses are truthful and that the opponent does not respond deceptively.

6.2 What we have learned

6.2.1 Filippo Gurioli

During the development of this project, I had the opportunity to deepen my knowledge regarding distributed architectures and the associated challenges. In particular, I learned to design and implement a distributed system, using various technologies and tools to ensure scalability, efficiency, and fault tolerance. I gained skills in load balancing, resource management, and the design of complex systems. While I was already familiar with the realm of REST APIs and web services, this project allowed me to gain a different practical experience, as I had to adhere to significantly higher and more complex standards.

6.2.2 Andrea Biagini

This project allowed me to gain a deeper understanding of how to design a distributed system and highlighted the importance of precise and thorough analysis and design. It also enabled me to approach the characteristic problems of this type of architecture with greater seriousness: I grasped the concepts of fault tolerance, scalability, and availability, and I discovered the importance of making decisions on the key trade-offs, particularly those arising from the CAP theorem. I practiced using Docker and rediscovered the convenience of having a clean, organized, and as frictionless as possible project. Additionally, I became familiar with the RESTful architecture, exploring it with a fairly high level of detail.

References

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy.* San Val, 1995.

[2] Giovanni Ciatto, Alfredo Maffi, Stefano Mariani, and Andrea Omicini. Smart contracts are more than objects: Pro-activeness on the blockchain. In Javier Prieto, Ashok Das Kumar, Stefano Ferretti, António Pinto, and Juan Manuel Corchado, editors, *Blockchain and Applications*, volume 1010 of *Advances in Intelligent Systems and Computing*, pages 45–53. Springer, 2020.

[3] Wikipedia. Guess who? 2024.