Final Report Template
UNO-P2P: A Cheater-proof Peer-to-Peer implementation of a simple card game.

Final Report for the Distributed Systems Course [2025/2026]

Luka Finn Born
lukafinn.born@studio.unibo.it

March 6, 2026

When it comes to playing games online, cheaters represent a major problem. Especially, peer-to-peer architectures are vulnerable to cheating as the game state and logic are handled purely by untrusted peers, rather than a single trusted server [5]. However, using P2P approaches can increase scalability as this usually implies smaller bandwith requirements for the service provider. In this project, we develop and design a system that provides a (mostly) cheating-proof method for implementing a card game in a P2P fashion. As a proof-of-concept, we develop a card game, similar to the game 'UNO'. Further, we design a distributed architecture that integrates this approach into a more complex system that additionally provides functionality for a leaderboard that tracks the win statistics of the players.

Disclaimer

During the preparation of this work, the author(s) used the following tools/services.

- *Grammarly* for grammar correction.
- *DeepL (only the translator)* to help with finding words/synonyms while writing the report.
- *Zed AI Edit Predictions* for smart autocompletions while writing code.
- *Zed AI Agent* to accelerate the development of some individual functions/classes, aswell as some unit-tests. No LLM was used to write the report or design the software architecture.

After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the final report/artifact.

1 Concept

In this project, we develop a simplified version of the popular UNO game, available as a web service. The users can play online with strangers and view a leaderboard that tracks the number of games won.

The intended user-base are regular people who like to play card games in their free time. To interact with the system, they can simply use their computer, phone, or any other device that supports Browsers with a modern implementation of web standards.

The system only needs to store credentials of the users, aswell as track the number of wins. In the current draft, we do not provide a role system and treat all users equally.

1.1 Game Rules

In the beginning, each player draws 7 cards and an initial card is put on the discard pile. To win the game, the players need to be the first to get rid of all of their cards. The cards are kept secret between players. Each card has one of four colors: Red, Blue, Green, or Yellow, along with a number from 1 to 9.

Similar to the original UNO game, this is a turn-based game. Here, the turn order needs to be externally defined and is not given directly by the game rules.

Each turn goes as follows:

1. The player optionally draws new card(s).
2. The player either
   - plays a card by putting it on the discard pile. This is only possible if either the number or the color of the card matches the current topmost card on the discard pile.
   - skips the turn.

1.2 Why is distribution needed?

Our distributed P2P architecture for game sessions enables easy scalability without requiring much computing power and network traffic on the host side.

Regarding the distributed nature of the other components, the biggest advantages are fault tolerance and availability. Even if one component completely fails, the other parts of the system are still mostly usable. The caching servers for the leaderboards enable better load balancing and faster response times for the users.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

Upon deployment, there should be an HTTP endpoint that the user can use to launch the client inside their browser.

Inside the client, there should be options for logging in and creating new accounts. When the user has logged in, they should not be required to re-enter their credentials for every request to the backend server. The user should also have access to a leaderboard where the number of wins of players is tracked.

When a user logs out or changes their password, all active tokens should be invalidated after a short timespan.

To start a game with other players, there should be functionality for creating rooms along with a server browser where players can join open sessions.

2.2 Non Functional Requirements

As our goal is to design this system so that players can play with strangers without having to worry about cheating, it is a hard requirement to prevent cheating at all cost. Even though we are using a peer-to-peer architecture for the game sessions, we require every peer to not be able to know the current cards of the other users. This minimizes the risk of users gaining an advantage by using a manipulated client software. Further, peers should not be able to manipulate the card-drawing process in order to influence the drawn cards.

2.3 Implementation Requirements

The user interface of the application should be able to run in a web browser environment. This makes the application easily portable across platforms. This also allows us to serve the client as a Webpage, removing the barrier of entry that would be posed by having to install a software package.

The restriction to the Browser ecosystem, of course, also imposes constraints on the set of viable frameworks and programming languages. As most browsers are only capable of running JavaScript and WebAssembly, we are locked to these languages or languages that compile/transpile down to them.

Since browsers cannot act as traditional servers with reachable listening ports, WebRTC is required to handle the signaling and NAT traversal necessary for direct peer-to-peer data exchange. The implications of this requirement are discussed in section 4.

Regarding the backend, however, there are not many restrictions. In principle, one could use any language that has access to system APIs and (ideally) has a rich ecosystem of libraries for networking.

2.4 Relevant Distributed System Features

This section describes relevant distributed systems features that this system implements.

Scalability: We would like this system to be easily scalable so we can easily cope with a growing user base. Our peer-to-peer architecture enables us to offload most of the potentially growing amount of requests and traffic to the clients.

Fault Tolerance: When a single component of the system fails, the others are still usable to some degree. No component fully depends on others. How quickly the system has to recover from a failure depends on the type of the component. For example, even if

the complete leaderboard system fails, the other components will still be usable. If only the main leaderboard server fails and the distributed caching nodes still remain online, only the functionality for adding new records to the leaderboard will be unavailable. One component that can only cope with short outages is the user management server, which hands out the JWTs to the users. When this server is offline, it will no longer be possible to log in or create new accounts. However, previously issued JWTs will still be valid and can be used for communication with the other components. To favor security, we, however, prefer using relatively short expiration times for the tokens, which means that the complete system may get unusable if the user management server is offline for a long period of time.

Performance/Bandwidth: Using our P2P architecture during the games, we try to minimize the required network traffic on the server side. This minimizes the requirements for hardware and bandwidth, potentially saving costs.

Interoperability: Since the server and client both operate in different ecosystems and are developed in different languages, we require mechanisms for interoperability. However, this is mostly limited to serialization and deserialization.

Security/Trust: To manage the leaderboard system, we require authentication mechanisms to ensure that players can only submit game results for matches in which they have participated. Additionally, we employ measures during the game, in our P2P protocol, to ensure that no participants can cheat by using manipulated clients.

Transparency: Our system does not require explicit hiding of the distribution details, as the client can directly communicate with the individual components of the system.

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis.

3.1 Architecture

In general, we mix two different styles of architecture. For the game sessions, we use a peer-to-peer architecture where the players communicate directly with each other while minimizing communication with any infrastructure that needs to be hosted by the deployers.

For the user management server, we implement a classic client-server architecture, as we require a very high degree of consistency here. Regarding the leaderboard server, we utilize one centralized server for writing new records into the database along with multiple distributed caching instances. This helps us to avoid write conflicts while still providing a high degree of availability for reading the leaderboards.

3.2 Infrastructure

Apart from the P2P communication during the game, our system consists of 3 main components:

1. **User Management:** The component that handles user registrations and logins. This component is implemented using a centralized instance without any replication or caching to ensure the highest level of consistency and partition tolerance. Upon login, a JSON Web Token (JWT) is handed out to the user, which they can use to authenticate themselves with the other components. This enables the other components to authenticate and authorize users without needing to actively communicate with the user management server, avoiding a single point of failure and thereby increasing availability.

2. **Matchmaking & Signaling:** This component is used to start new game sessions or assign users to game sessions that have not started yet. The matchmaking component also handles the connections establishment between the peers. This component needs to know the public key of the User Management service, but apart from that, there are no other dependencies on other components. When the session starts, the matchmaking server distributes a signed handle that specifies the turn order and the initial card on the discard pile. This handle can later be used for submitting the outcome of the game to the leaderboard service.

3. **Leaderboard:** The Leaderboard component manages a leaderboard that tracks the number of wins of the players. For writing new records into the leaderboard, we use a single centralized node to ensure consistency and avoid conflicts. The leaderboard needs access to the public keys of the user management server and the matchmaking servers to authenticate the user and verify that the game session for which this record is being submitted exists and that the player has participated in the game. For reading the leaderboard, our system architecture provides caching nodes, reducing latency and improving availability.

The reasoning behind the centralized architecture of the User Management Service and the Leaderboard service are explained in more detail in section 3.6.

Figure 1 shows a visualization of the different components and the dependencies between each other.

3.2.1 Component Discovery

The discovery between the individual peers is performed by the matchmaking server, which acts as a signaling channel for exchanging the ICE messages.

In order for the user frontend to discover the other non-peer components, such as the user management server, the URLs of the components need to be provided manually to the web client using CLI arguments. However, when actually deploying this software to production, a more dynamic approach would be to use different subdomains for the individual components (e.g., 'eu.matchmaking.p2p-uno.com').

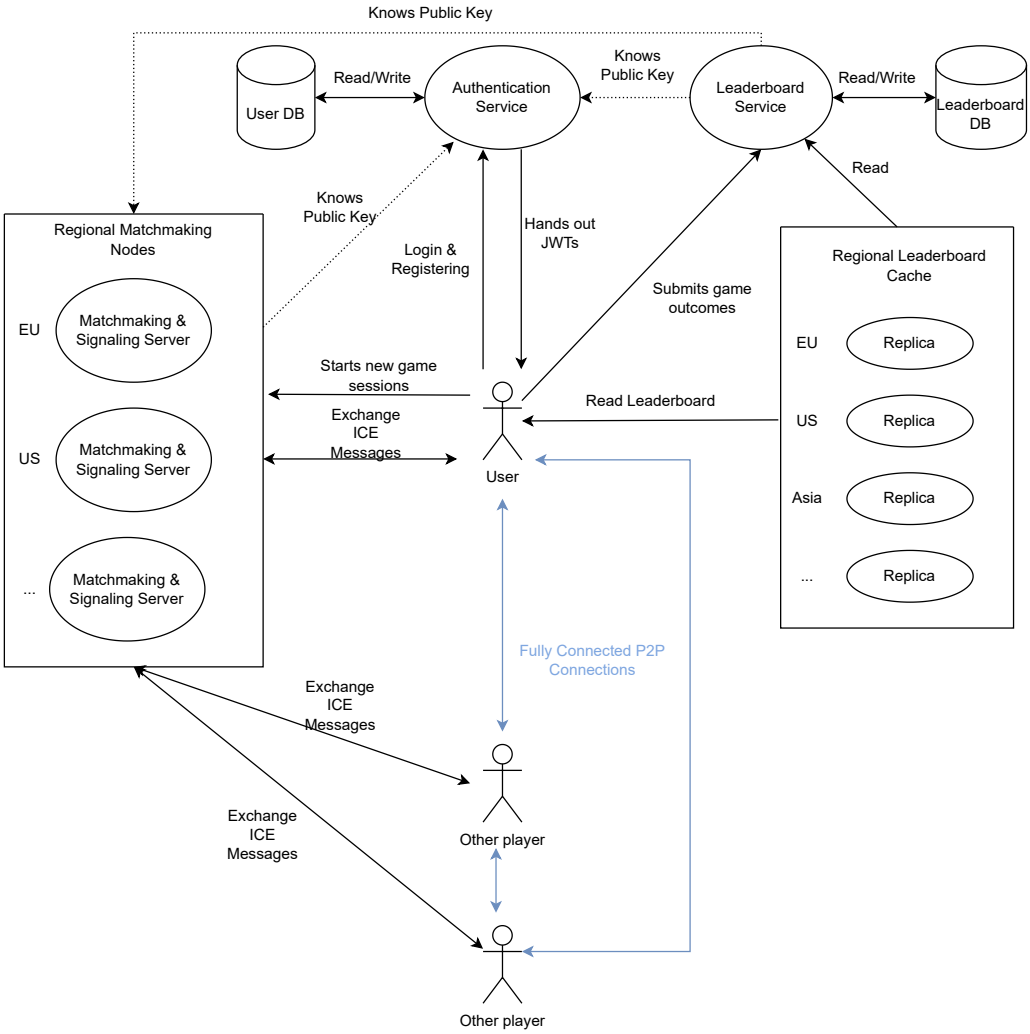

Figure 1: Overview of the different components and their role in the system.

3.3 Modelling

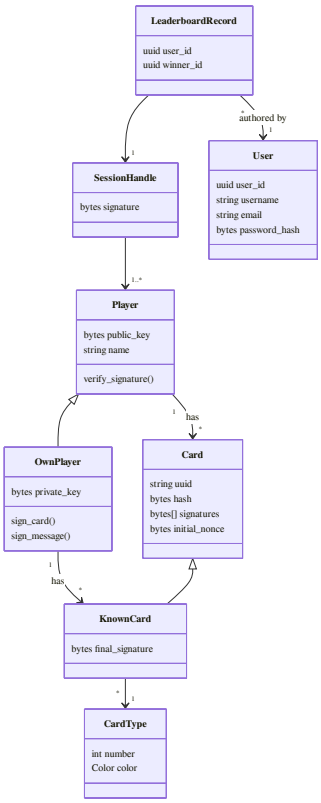

Figure 2: Class diagram of the software system.

Figure 2 shows a class diagram describing the most important domain entities.

- which **domain entities** are there?
  - LeaderboardRecord: A record submitted to the leaderboard by the user.
  - SessionHandle: A handle that contains the players that participated in the session.
  - User: Contains the relevant user data (e.g. username, password) for authentication.
  - Card: Holds the relevant data for a card. Note that the signatures array does not yet include the final signature that is used to determine the card type.
  - KnownCard: Card with all signatures and a known card type.
  - Player: Represents the state of a player during a game.
  - OwnPlayer: Represents the own player during a game. The main difference to the regular Player is that the OwnPlayer has KnownCards instead of Cards.

- how do *domain entities* **map to** *infrastructural components*?
  - The LeaderboardRecords are submitted by users to the Leaderboard and stored there.
  - The SessionHandles are generated by the matchmaking server and given to the players at the start of the game.
  - The Users are stored on the User management server.
  - Cards and KnownCards only exist during the game and are managed by the peers.
  - Each peer also has a OwnPlayer instance for signing messages and cards.
  - In each peer, the other players are represented as Player.
- which **domain events** are there?
  - User registers.
  - User requests new JWT using credentials.
  - User creates a session.
  - User connects to session.
  - Session initiator requests to start the session.
  - The session starts.
  - A player plays a card.
  - A player requests a card draw.
  - A player skips their turn.
  - User submits a record to the leaderboard.
- which sorts of **messages** are exchanged?

  Among the peers, mostly only events are published. Assuming the game state is consistent among the peers, each peer can derive from its own state whether it should broadcast a message to the other peers. Therefore, we do not require any *queries* for the regular game flow.

  However, for communication with the other components the user sents requests (e.g. for logging in or submitting game outcomes).
- what information does the **state** of the system comprehend
  - User authentication data.
  - The current game state:
    - The top card on the discard pile.
    - The cards on the own hand.
    - The cards on the other players hands.

- Open sessions.
- Win statistics of the players.

3.4 Interaction

The client/peer software is the only component that actively interacts with other components.

3.4.1 Joining & Creating a Session

Figure 3 shows a full overview of the interactions between components when joining or creating sessions. When creating a new game session, the player sends a request to the matchmaking server, which then opens a new session that is initially set to an idle state, where it waits for new players to join. The matchmaking servers provide an endpoint for fetching sessions that are currently in an idle state. To join a session, the user can then send a request to the matchmaking server. If the join request is valid, the user is added as a 'trial Player' and receives the id of the session, a 'proof-nonce', along with the names and public keys of the other players that have previously joined.

The next step is to establish the Peer-to-Peer connections between the new player and the current players. For the communication between each pair of players during this phase, signaling channels are opened. Using this signaling channel, the peers now exchange the ICE messages in order to establish a connection. To prove that the player has connected to all other players, the new player lets all other players sign the 'proof-nonce' using their private key. Afterwards, once the player has proven to the matchmaking server that it is connected to the other players by sending their signatures of the challenge nonce, the state of the joining player will be upgraded from the 'trial' state to a verified state. We do this to ensure that we initially have a fully connected network. When the session is started, each peer receives a session handle that contains all players that participate in the session (i.e., all players that have joined and are in the verified state), the turn order, the initial card on the discard pile, and the session id, along with a signature of this payload. After the match, this session handle is then used to add the outcome of the match to the leaderboard. This ensures that only players who actually participated in the game session can submit the outcome.

3.4.2 Drawing a Card

When drawing a card, the player generates a nonce and sends it around to each player. Each of the other players signs the nonce with their private key and forwards it to the next one. The last player then sends it back to the player who initiated the card-drawing procedure. This player then also signs the nonce but publishes only a hash of the nonce with all signatures to the other players. Figure 4 shows a sequence diagram of those interactions. Section 3.9.1 explains this process in more detail.

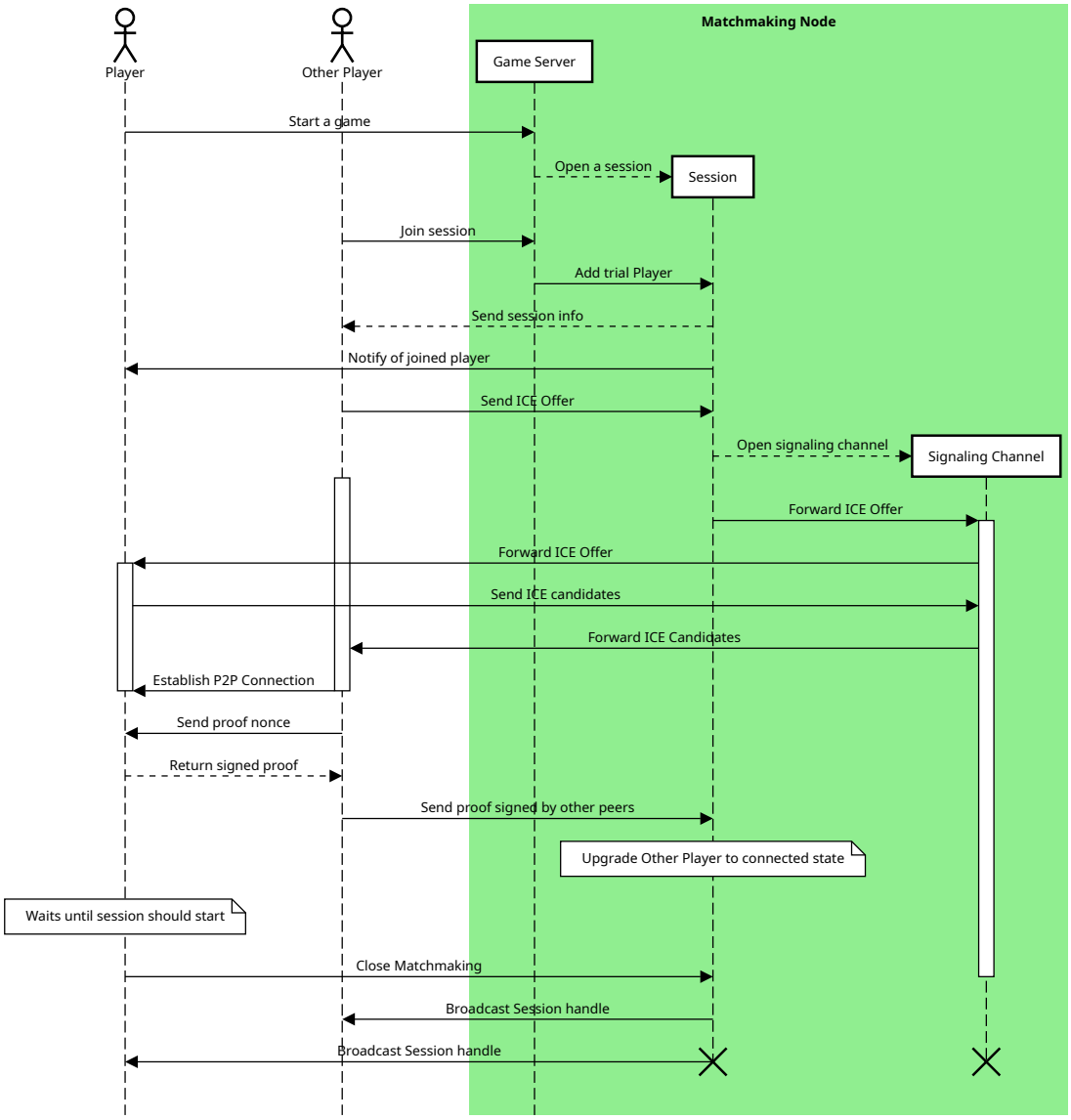

Figure 3: Sequence Diagram for the interactions when creating and joining a game session.

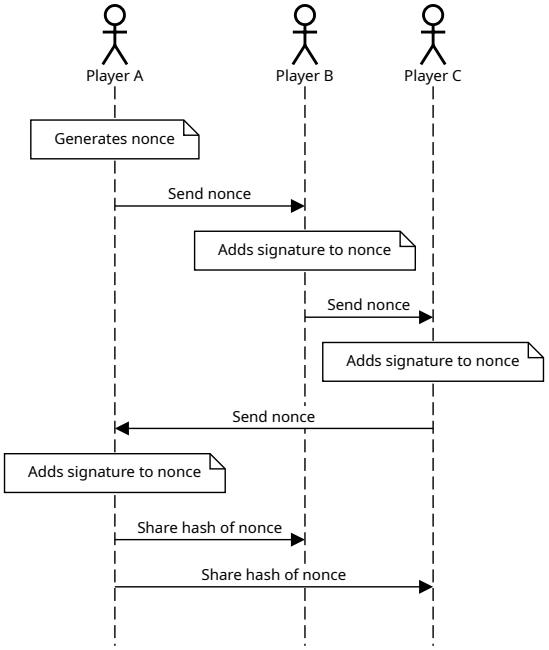

Figure 4: Sequence Diagram for the interactions between the peers when drawing a card. Here, Player A is the one who is drawing the card.

3.4.3 Playing a Card

When playing a card, the player now publishes the nonce generated during the card drawing process. The other players then verify that the nonce matches the previously published hash and that all the signatures are valid. If this is not the case, the message will simply be ignored. As described in section 3.8, if the player does not manage to publish a valid card play, draw, or turn skip in time, they will be kicked from the session. Otherwise, each player advances their local game state according to the card that was played. If the game has finished, each of the players submits the result to the leaderboard service. Figure 5 shows a sequence diagram that details these interactions. See section 3.9.1 for a more detailed description of the algorithm for verifying the nonces and signatures.

3.4.4 Submitting the Results to the Leaderboard

After the game ends, each player submits the outcome of the game to the leaderboard. For the outcome of the game to be included in the leaderboard statistics, an absolute majority needs to be achieved on the outcome of the game. If no single outcome achieves this absolute majority, the game session is discarded.

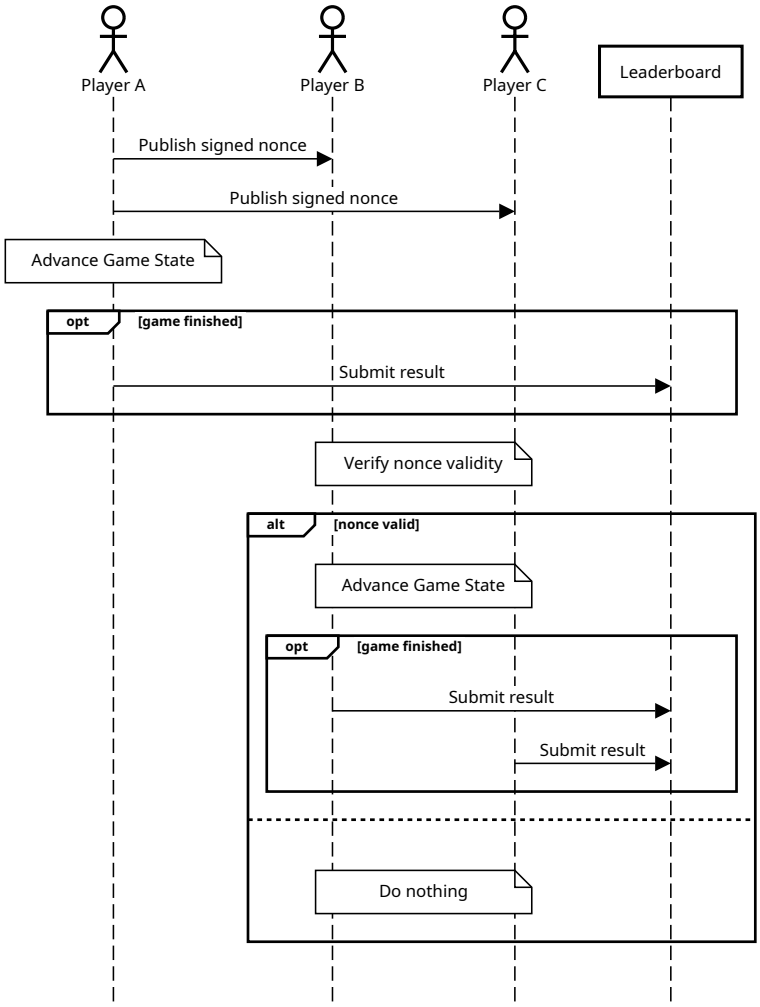

Figure 5: Sequence Diagram for the interactions between the peers and the Leaderboard server when playing a card. Here, Player A is the one who plays the card.

3.5 Behaviour

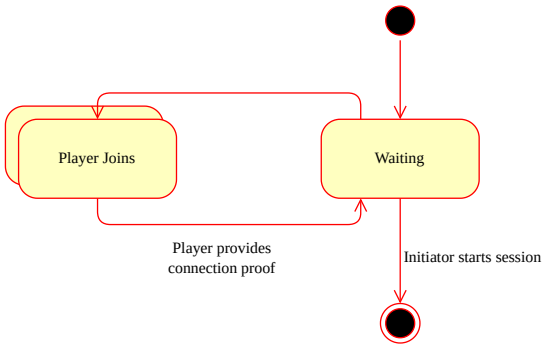

Figure 6: State diagram for a session on the matchmaking server.

The only components that need to maintain a complex state are the peer and the matchmaking server. Both the user management server and the leaderboard server only require a database whose records are updated by user interaction (logging in/submitting a record).

Figure 6 shows a state diagram for the matchmaking server in the lobby phase before the game. In total, we have only two states:

1. An idle state ('Waiting') where the matchmaking server waits for players to join.
2. A state in which the proof for existing connections to the other peers is awaited. Note that this state does not block new players from joining the session, meaning that we can have multiple parallel 'Player Joins' states.

The state diagram for the peer component, shown in fig. 7, is a bit more complex. Here, we first have a preparation phase in which each player draws their initial cards. After that, we are in the main phase of the game. Here, each player awaits a request to either play a card, draw a card, or skip the turn from the current player. When the player requests to draw a card, we switch to the 'DrawCard' state. The messages exchanged during this phase in order to draw the card are explained in section 3.4.2. If the current player plays out a card, we switch to the 'PlayCard' state, where the card hash and signatures are verified. In case the player now has 0 cards, we switch into a finished state. If this is not the case or the player requests to skip the turn, we move on to the next player and go back into the 'Waiting' state.

For both the matchmaking server and the peer component, all state updates are triggered by the peer component.

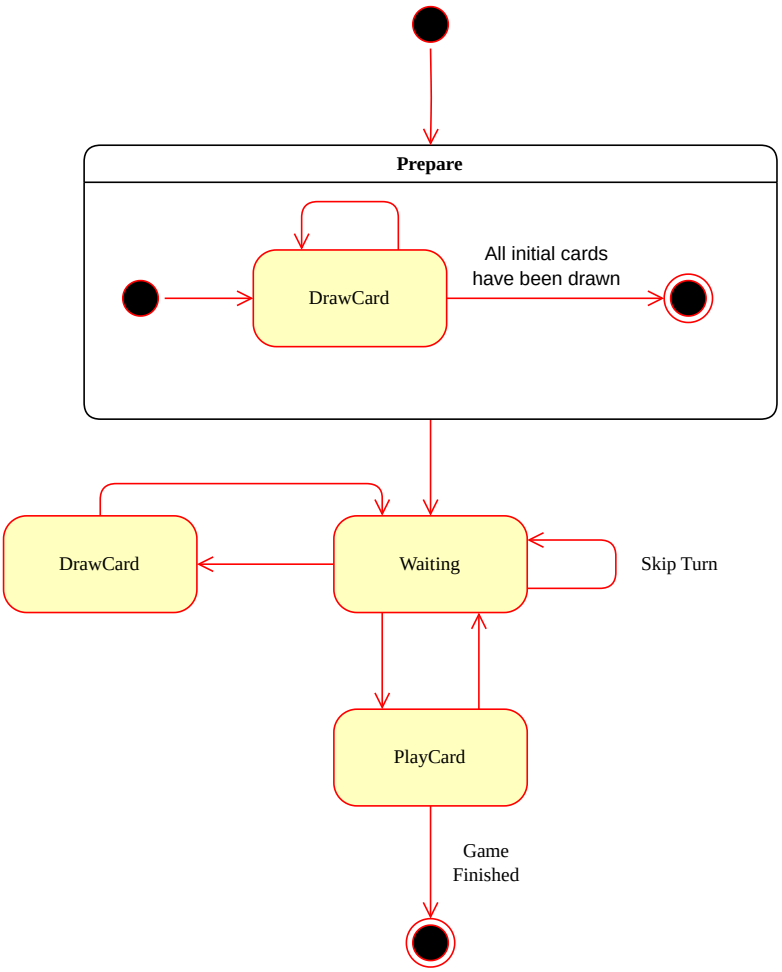

Figure 7: State diagram of an individual peer during a game.

3.6 Data and Consistency Issues

In total, we only require two databases:

1. A user database that stores usernames, hashes of passwords, and potentially refresh tokens for refreshing JWT tokens.

2. A leaderboard database that tracks the outcomes of the games and the number of games won by the individual players.

Both databases are hosted in a centralized fashion and only interact with a single server to achieve high consistency and partition tolerance. Regarding the user database, this improves account security for the users. For example, in a distributed database with eventual consistency, it might be possible to log in with an old password after a password change. This makes it easier for users to prevent bad actors from gaining access to their accounts in case the user credentials were leaked.

As for the leaderboard, simply using a centralized server for writing records enables us to prevent issues where, for example, a player would submit different game outcomes to different server nodes. For reading data from the leaderboard, we use additional caching nodes to improve response times and availability. While this means that data written to the leaderboard may not be immediately visible to the users, we do get a higher degree of availability for reading the active leaderboards. We think that in this case, it makes sense to prioritize availability over consistency, as we assume that the ranking of the players in the leaderboard will not change frequently.

Among the matchmaking service, the user management service, and the leaderboard, almost no data needs to be shared. It is only required for the matchmaking service to know the public key of the user management service and for the leaderboard to know the public keys of both the user management service and the matchmaking service.

The data in the respective databases is stored in simple relational database systems.

During the game, we use a protocol that ensures a strict ordering of the messages sent and minimizes packet loss. Due to our implemenation of the game, the state at each time can be deterministically derived from the sequence of past events. This ensures that all peers have a consistent state.

3.7 Fault-Tolerance

For the leaderboard, we utilize caching servers that can provide the last known leaderboard state while the main leaderboard server is offline. When one of the regional game server fails or is not reachable by the user for other reasons, the user can connect to one of the other regional servers. If we only provide one server per region, this might mean that the user experiences higher latencies both during the game establishment phase and also in the game session itself, as the other players are likely to be located at large distances away from the player. How many game servers exist and where they are situated is, however, up to the specific deployment.

During the game, in the peer-to-peer network, we use heart-beating between the peers. Due to our choice of technology, we do not need to implement that ourselves. See section 4 for more details.

In order to deal with unresponsive peers during the game, we have predefined maximum timeouts after which the peers vote to kick the unresponsive peer from the game. To ensure that players are not kicked from the session in cases where there are no actual faults, like e.g., the user taking too long in the waiting phase, our client implementation automatically skips the turn before the timeout ends.

3.8 Availability

- Is there any **caching** mechanism?

  We implement a caching mechanism for the leaderboard to achieve high availability when it comes to reading records.

- Is there any form of **load balancing**?

We use multiple (in production, spatially) distributed components for establishing the connections between the peers.

- In case of **network partitioning**, how does the system behave?

  Since the user management, leaderboard, and matchmaking services do not need to actively exchange messages, a partitioning among these components would have no negative effects on the availability.

  However, if one of those components is not available from the point of view of the user, certain functionality may not be available. When the Leaderboard service is not available, it will not be possible to submit new records to the leaderboards until this service is available again. In this case, however, the clients can simply store the result of the match in the local Browser storage and submit it when the service is back online. When the user management service is unavailable for the user, they will not be able to log in to their account. However, already logged-in users will still be able to use all other components until their JWT expires. If one of the matchmaking nodes or the regional leaderboard cache is offline, the user client can simply use the next best node.

  Regarding partitions among the peer-to-peer network during the matches, things get a bit more complicated. Since each peer broadcasts all messages it receives to all other connected peers, the network will be fully functional as long as it represents a connected graph. If, however, we get disconnected components in the graph structure, two things can happen:

  – If one of the components/partitions contains the majority of the players, the game will be continued without the players from the other partitions.

  – If none of the partitions contains a majority of the players, the game will be canceled, as it will not be possible to submit this game to the leaderboard since we do not have enough players left to achieve a majority vote.

  To minimize the chances of achieving this kind of partition, we implement a self-repair mechanism in the case where single edges in the connectivity graph fail. Here, the other players then take the role that was previously handled by the matchmaking server and exchange the ICE messages in order to re-establish the failed WebRTC connection.

3.9 Security

- Is there any form of **authentication**?

  We use authentication so that we can assign users to matches when they create or join a game using the matchmaking nodes. This is required to be able to track the win statistics of the users in the leaderboard.

- Is there any form of **authorization**?

In the current blueprint, we do not require any kind of authorization. However, when deploying this system to production, one could consider adding an admin role that has access to e.g. dashboard for usage metrics.

- Are **cryptographic schemas** being used?

  We use JWT token verification in the matchmaking nodes and the leaderboard database. In the P2P communication during the game, all peers sign their messages using an asymmetric schema to prevent man-in-the middle attacks in cases where the network is no longer fully-connected. The matchmaking server also sends a handle with a cryptographic signature to the players, which they require to submit the outcome of the match to the leaderboard. Further, we use a custom protocol for drawing and playing cards, which is described in more detail in the following section.

3.9.1 Peer Secrets

During the game, we rely on a fully Peer-to-Peer architecture where no further interaction with components is required. When managing the game state among the peers, there are two major aspects that may pose a challenge. First, how we store the game state among the peers, and secondly, how to draw new cards.

Regarding the game state, three intuitive implementations could be to:

1. Select a 'master' peer that full knowledge about the game state, i.e., knows the cards of all players and is able to verify whether their moves are allowed.
2. Let every peer have full knowledge of the game state.
3. Each peer only knows their cards and has no further information about the other players.

However, all of these approaches open up the possibility of a player modifying their client software in order to achieve an advantage against the other players. In the first two approaches, players could be able to see the cards of the other players. Electing a new 'master' peer when the current one is suspected to be cheating is also not a viable solution as the old master peer has already seen the cards of the other players at this point. Furthermore, in the third approach, players would be able to play out cards that they do not own, as it would be impossible for the other peers to verify whether this peer has previously drawn that card.

For drawing new cards, we have a similar problem. When selecting a 'master' peer that draws the cards for the player, a malicious master peer could intentionally choose cards that put the other peers at a disadvantage. When, on the other hand, we let each peer draw the cards fully for themselves, users with manipulated clients could freely choose cards.

To solve both of these problems, we propose a custom cryptographic algorithm. This algorithm allows peers to keep the cards secret from other peers while still keeping the

possibility of validating that played-out cards have been previously drawn. Additionally, when drawing cards, the players are not able to freely choose cards with this algorithm.

Let $h_1 : \mathbb{N} \to \mathcal{N}$ be a hash function such that it is sufficiently hard to find $h_1^{-1}(x)$ for a given x. Let C be the set of different cards. Let $h_2 : \mathbb{N} \to C$ be another hash function that maps any natural number to a card. Further, it should be sufficiently hard to find $h_2(x)$ given $h_1(x)$ for any x. Let N be the number of players. Let $\text{sign}(p, k) : (P, \mathbb{N}) \to \mathbb{N}$ be a function that produces a signature for the payload p, given a private key k. It should be sufficiently hard to find $\text{sign}(p, k)$ given only the payload and the public key.

We assume that each player has access to a secure random number generator. Further, we assume that the players have agreed on a specific turn order. Each player i has a public and a private key $k_{\text{pub},i}$ and $k_{\text{priv},i}$. These keys are generated once at the beginning of the game and cannot be changed.

Now the process for drawing cards is as follows:

1. Player i generates a random nonce r_0 using their secure random number generator and sends it to player $i + 1 \mod N$.

2. Player $i + 1 \mod N$ signs the random nonce r_0 using their private key: $r_1 = \text{sign}(r_0, k_{\text{priv},i+1 \mod N})$ and sends it to player $i + 2 \mod N$. This process is repeated until player i is reached again.

3. Player i also signs the payload with their private key $r_N = \text{sign}(r_{N-1}, k_{\text{priv},i})$ and publishes $x = h_1(r_N)$ to all other players. Now player i can determine the drawn card by $h_2(r_N)$.

Since player i cannot predict how the signed payloads of the other players will look, they cannot know which card will be drawn when initially generating the nonce. Additionally, since the other players only have access to the hash $h_1(r_N)$, they cannot yet predict which card was drawn by the player.

One remaining problem here is that players could simply reuse the nonce of previously drawn cards. To counter this, each player stores all the payloads they received during past card drawing processes and verifies that the received payload has not been seen before.

When player i intends to play a card, they can now simply publish r_N to the other players, allowing them to verify that:

1. The previously published hash $h_1(r_N)$ matches r_N.

2. The signatures from all players are valid.

Since the game is fully deterministic, apart from drawing cards, each player can now use $h_2(r_N)$ to advance their local game state.

4 Implementation

This section reports the high-level implementation choices we made while implementing the design.

In this prototype, only the following components are actually implemented. We think that the core aspect of our project is our P2P protocol that prevents cheating.

- The P2P Client.
- The matchmaking server.

However, we still describe the implementation choices we would have made for the missing components in the following section.

- which **network protocols** to use?

  The communication between the game client and the other non-peer components, such as the user management server, the leaderboard server, and the matchmaking server, is done using HTTP. One exception is in the waiting lobbies, where we use WebSocket connections to establish the P2P connections.

  As we previously both required a frontend that can run as a web application and a peer-to-peer network, the options are limited. Therefore, we chose to implement the in-game communication purely using WebRTC data channels [1] to achieve true peer-to-peer connections. In order to establish the connections between the peers, we use the Interactive Connectivity Establishment (ICE) system. For this, the matchmaking server acts as a signaling server that exchanges the ICE messages. For NAT traversal, we first attempt to use STUN servers provided by Google [4]. We intentionally provide multiple servers to ensure availability in case some of these servers are unavailable. If the connection establishment using only STUN fails, we provide a collection of TURN servers through which the traffic is then routed. In order to avoid unauthorized use of those TURN servers, the matchmaking server issues user-specific, time-limited credentials following the TURN Rest API specification [3].

- how should *in-transit data* be **represented**?

  All in-transit data is serialized to JSON due to the simplicity and debugability of the format. Binary blobs are encoded as base64 strings. This can introduce a minor increase in packet sizes; however, since we are only dealing with fairly small blobs in this system, we think that this is acceptable.

- how should *databases* be **queried**?

  The databases for the leaderboard server and the user server can be queried using SQL. However, in this prototype, this functionality is not implemented.

- how should components be *authenticated*?

  The user authenticates themselves using a JWT. The user client is the only component that actively interacts with other components. In order to ensure that match result votes submitted to the leaderboard server are valid, the user needs to provide a handle signed by the matchmaking server that is acquired when the game starts.

4.1 Technological details

4.1.1 P2P-Client

For the user interface, we rely on the React framework[1] in combination with TypeScript[2]. As a build system, we use Vite. To build the interface, we mostly used components from the Ant Design component library[3]. Apart from that, we tried to stick to standard browser APIs without introducing too many dependencies.

4.2 REST

To implement the REST APIs for creating sessions, authenticating, and submitting records to the leaderboards, we use the Python library FastAPI for this prototype. Python, being among the most popular languages, according to the StackOverflow Developer Survey 2025 [2], already provides a rich ecosystem of libraries. Further, the ease of use, indicated by it being the most popular language among beginners [2], makes this language a suitable fit for developing a prototype for this system. Note that the user management server and the leaderboard server are not implemented in this prototype.

5 Validation

5.1 Automatic Testing

To test the peer component of our system, we use unit-tests to test the basic game flow, serialization, the message signing and card drawing mechanisms, aswell as helper functions that we manually implemented.

Specifically, we developed the following unit tests for the frontend and P2P client. All of these tests were implemented using the `vitest` testing framework and can be executed by running `npm run test` in the root directory of the frontend implementation. We also provide a GitHub workflow that runs these tests after every push.

- We developed a set of unit tests to test the functionality of a custom queue implementation with support for awaiting new elements, similar to the `asyncio.Queue` implementation in the Python standard library.
- We provide unit tests for testing a custom semaphore implementation. Like the previous test, this unit test is not directly linked to any requirement.
- We developed tests to verify our implementation for signing and verifying cards. This verifies that the implementation of the algorithm presented in section 3.9.1 works and that the requirement of being cheater-proof (apart from majority attacks) is fulfilled.

[1] `https://react.dev/`
[2] `https://www.typescriptlang.org/`
[3] `https://ant.design/`

- We developed a test to verify the functionality for signing and verifying messages. This functionality is relevant to avoid man-in-the-middle attacks in cases where the peer-to-peer network is not fully connected.
- We provide a set of tests for verifying our serialization logic. This is relevant for the communication between the components.
- We implement an automated test that tests the basic game flow (drawing & playing cards). This test verifies that the game runs as expected, but it does not test a specific requirement.

Further, we also provide a few unit tests for the matchmaking server component. These tests are implemented using `pytest` and can be run using `uv run pytest` in the root directory of the project. For these tests, we also provide a GitHub workflow that runs them after every push to the repository.

- A set of unit tests that test the functionality for fetching and creating sessions. This tests the requirement that the user should be able to create sessions and fetch sessions.
- A test that tests the functionality for signing and verifying signatures. This is necessary to verify the proofs sent to the matchmaking server when a connection to all other peers has been established.
- A test that verifies the serialization format for binary blobs. This is relevant for the communication between the components.

5.2 Acceptance test

To test the interactions between the peers during the game sessions in a production-like environment, we mostly relied on manual testing. This is because it is particularly difficult to write automated tests for this setting, as our protocol for drawing cards does not enable us to easily predefine the cards of the players. Furthermore, testing the interactions would require managing many processes in parallel, making the implementation of fully automatic end-to-end tests more complicated.

6 Deployment

For easy deployment, we provide a Docker container that launches a server that serves the frontend, aswell as two different matchmaking servers. Currently, everything is situated inside a single container. In production, one would of course split this into multiple containers so that it can be run on different machines.

To run the system, follow the following steps:

1. Clone the repository at `https://github.com/IchbinLuka/p2p-uno`.
2. Run `docker-compose up --build` inside the project directory.
3. Now the Frontend should be available under `http://localhost:8080`.

7 User Guide

7.1 Session Browser

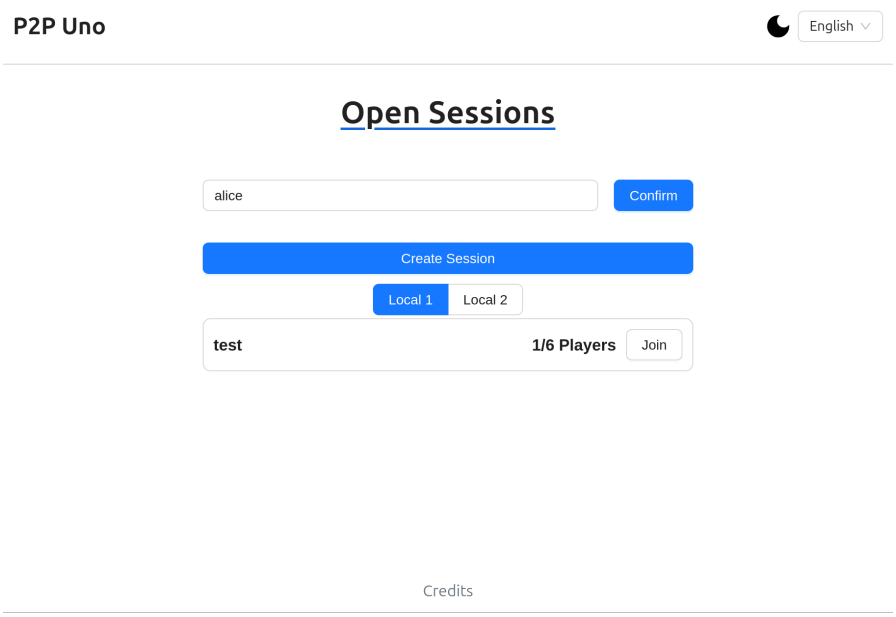

Figure 8: Screenshot of the homescreen/session browser.

After connecting to the server that hosts the frontend, the first thing the user sees is the session browser, which can be seen in fig. 8. At the top, there is a header with the name of the game, a button for switching between light and dark mode, and a drop-down menu for changing the language. Available languages are German and English. Below the site title, there is an input field for entering a desired username. In the future, one would implement a more advanced solution for creating and managing profiles when implementing the user management server. The 'Create Session' button can be used to create new sessions. That page is explained in more detail in the next section.

Below the 'Create Session' button, there is a bar of buttons that is used to select the desired matchmaking server that will be used for creating and fetching existing sessions. All the active sessions on the selected matchmaking server will be shown in the list view below. To join a session, the user can simply click the 'Join' button. Note that the buttons for joining and creating sessions are disabled if the username field is empty.

7.2 Creating Sessions

When clicking on the 'Create Session' button in the session browser, the user is taken to the screen shown in fig. 9. This screen shows a simple form where the user can specify a session name and a maximum player count. Upon clicking the 'Create Session' button,

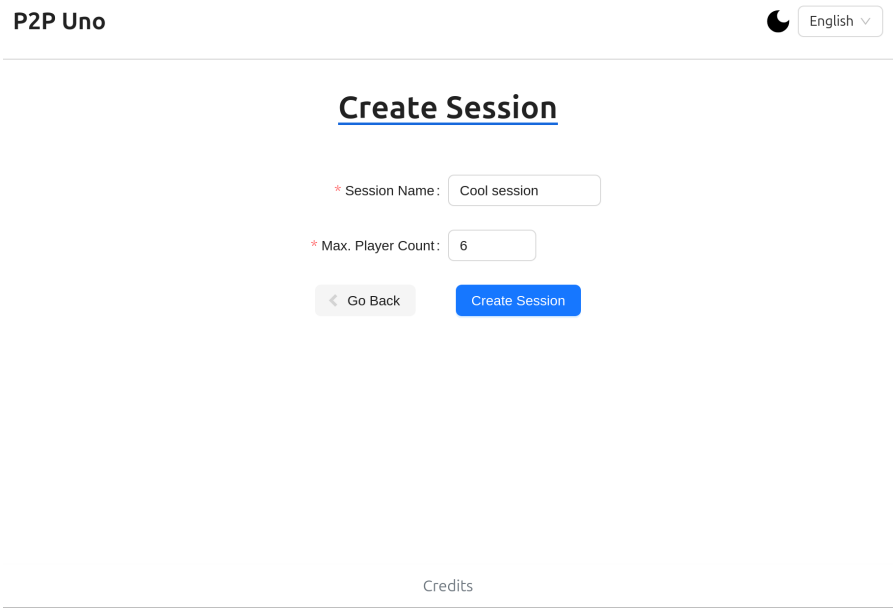

Figure 9: Screenshot of the screen for launching new sessions.

a request is sent to the matchmaking server to create a session, and the user is taken to the waiting lobby screen, shown in the next section.

7.3 Waiting Lobby

When joining a session from the session browser or creating a session, the user is shown the screen displayed in fig. 10. This screen shows all users that are currently connected or are in the process of connecting. The connection status of each player is shown next to the username.

If the user is the initiator/'host' of the session, they can start the session by clicking the 'Start Session' button. Note that this is only possible if two or more players are in the session.

7.4 Game

Figure 11 shows the user interface during the game. At the top, the current card counts of the other players are displayed in grey 'card-shaped' boxes. In the middle of the screen, you can see the current topmost card on the discard pile. Next to the discard pile, the time left for the move is displayed. When this counter runs out, the turn is automatically skipped. Below this pile, there are buttons for skipping the turn and drawing a new card in case none of the cards in the players 'hands' match the card on the discard pile. At the bottom of the screen are the cards that the player has in their 'hands'. Here, cards that can be played are highlighted by slightly shifting them up and

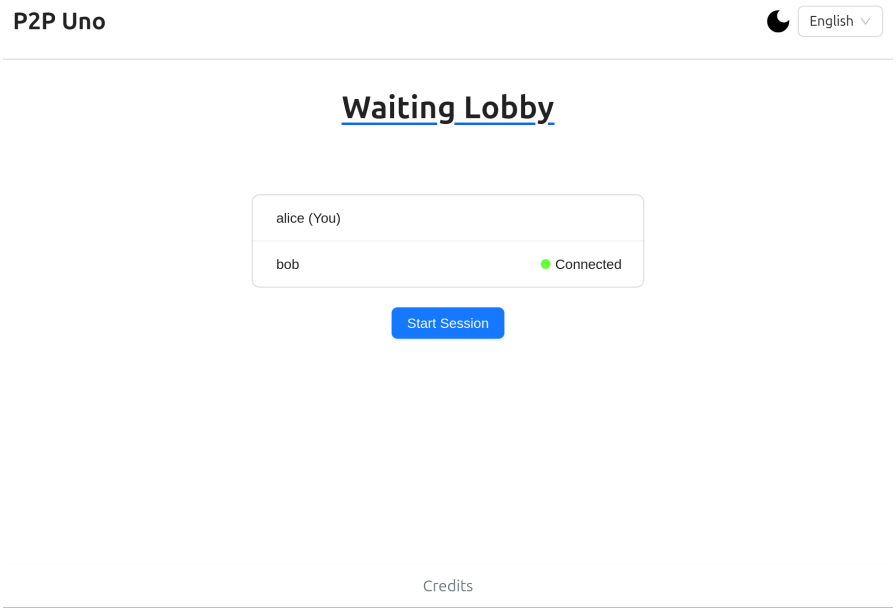

Figure 10: Screenshot of the waiting lobby.

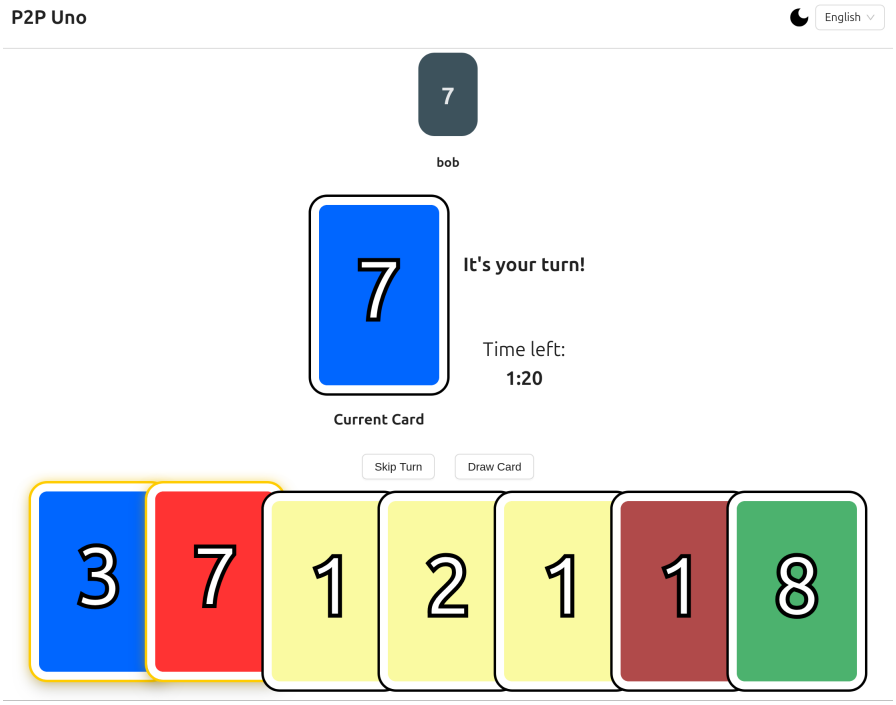

Figure 11: Screenshot of the game.

giving them a highlighted border. All other cards are slightly greyed out to indicate that they cannot be played.

7.5 Game end

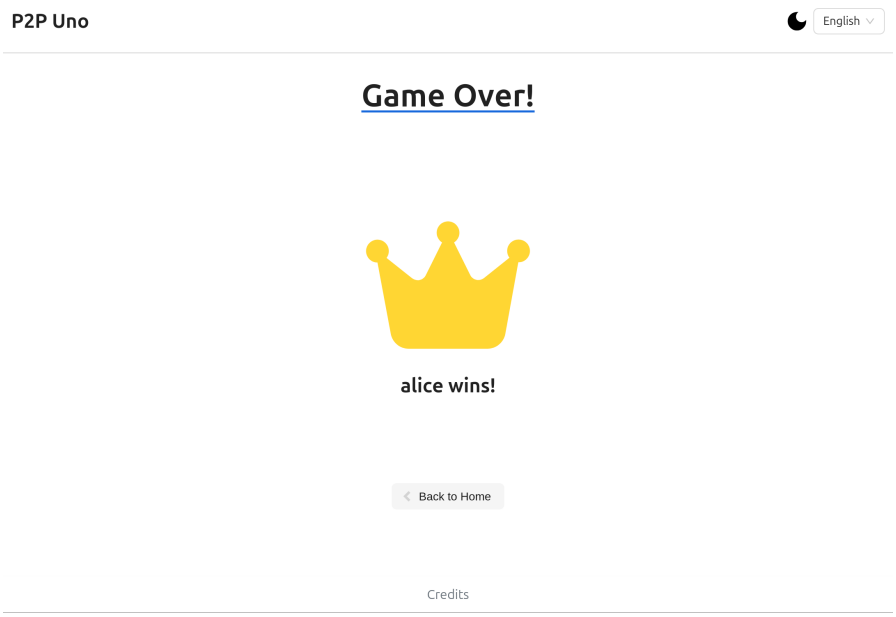

Figure 12: Screenshot of the game-end screen.

When the game is finished, the user ends up at the screen shown in fig. 12. Here, the winner of the game is displayed in the middle of the screen. At the bottom, there is a button for returning to the server browser. If the game got canceled because too many players lost their connection/were kicked, this screen simply shows a cross icon with a message indicating that the game got canceled.

8 Self-evaluation

The system in this project was completely designed and implemented by Luka Finn Born. In this project, we have presented an approach for an almost tamper-proof system that allows each peer to keep their cards secret while ensuring that card draws are actually random. We think that this method represents the core contribution of this project and could also be used for applications beyond card games, where other peers cannot be trusted. During the game sessions, the communication takes place in a peer-to-peer network, reducing the required network traffic for the entity that deploys the system. Additionally, our web-based implementation provides a high degree of cross-platform compatibility and portability. Apart from the peer-to-peer network during game sessions, the components of the system are fairly disconnected and do not require much

communication, which ensures stability in cases where individual components are not available. In total, we think that we have designed a system that fulfills the requirements outlined in section 2.

One weakness of this implementation could be the small number of automated tests, and the fact that the design of the mechanisms for drawing and playing cards makes it harder to design tests. Currently, the user also needs to choose a matchmaking server themselves. Another weakness is the limited ability to re-establish failed connections in the peer-to-peer network. We do implement a self-repair mechanism for cases where single edges fail; however, in the case of partitions where the peer-to-peer network results in multiple disconnected components, the network cannot be fully repaired. Further, the current prototype does not yet implement all modules of the system. Additionally, the peers in the P2P network are currently not fully aware of the connectivity state of the network. This means that when a peer loses the connection to all other peers at once, they will not be removed immediately from the session, but rather only after a timeout. Another weakness of the system presented in this project is that the peer-to-peer networks during game sessions are still vulnerable to majority attacks, where a malicious entity controls a majority of the peers.

9 Future works

Currently, the public keys for the individual (non-player) components are assumed to be constant. In the future, one could implement a secret-rotation mechanism to increase the security of the system. This would, of course, require a mechanism for communicating the updated public keys.

Further, one could use a more advanced CDN solution for serving the frontend, involving caching mechanisms to reduce website load times and increase availability. When a new version of the software is released, this also requires a cache invalidating system. In production, one could consider using commercial CDNs such as the ones provided by Cloudflare or AWS.

Additionally, future work could implement a more intelligent system for automatically selecting the best matchmaking server depending on the region. This could be baked into the CDN network, where each caching instance is aware of the most suitable matchmaking server.

Currently, we only implement regular cards with a number and a color. Similar to the original UNO game, one could also implement special cards, such as cards for switching the turn order or forcing the next player to draw a specific number of cards.

Regarding the components that are not yet implemented in this prototype, the leaderboard server could be implementd e.g., using a PostgreSQL database. The REST-API for this component could then be implemented using FastAPI. To implement the leaderboard cache, one could use an out-of-the-box solution such as Redis. To implement the user management server, one could also, e.g., use a PostgreSQL Database. This server would then provide endpoints for registering, logging in/retrieving JWT and updating the profile. To not require the user to always log in when their JWT expires, one could

also add refresh tokens that the client software can automatically use to retrieve new JWTs before the old one expires.

As outlined in section 8, the current design is not able to fully repair the peer-to-peer network when it is separated into disconnected components. Instead of canceling the session or continuing it with the remaining players, one could reopen the signaling channel on the matchmaking server to fully repair the network. To do this, the players would then be required to authenticate themselves with the session handle that they previously received when the session started.

References

[1] Ian Hickson, Cullen Jennings, and Taylor Brandstetter. Webrtc: Real-time communication in browsers, 2025.

[2] Stack Overflow. Stack Overflow Developer Survey 2025. `https://survey.stackoverflow.co/2025/technology#most-popular-technologies-language`, 2025. Accessed: 2026-02-21.

[3] Justin Uberti. A REST API For Access To TURN Services. Internet-Draft draft-uberti-behave-turn-rest-00, Internet Engineering Task Force, July 2013. Work in Progress.

[4] videosdk. Google stun server: The ultimate 2025 guide for developers & network engineers. `https://www.videosdk.live/developer-hub/stun-turn-server/google-stun-server`. Accessed: 2026-02-21.

[5] Steven Daniel Webb and Sieteng Soh. Cheating in networked computer games: a review. In *Proceedings of the 2nd International Conference on Digital Interactive Media in Entertainment and Arts*, DIMEA '07, page 105–112, New York, NY, USA, 2007. Association for Computing Machinery.