UNIVERSITY OF PISA

School of Engineering

Master of Science in Computer Engineering

Foundations of Cybersecurity

PROJECT DOCUMENTATION

FOUR IN A ROW

WORKGROUP:

Diego Casu

ACADEMIC YEAR 2021/2022

Contents

1. Introduction

Four in a Row, also known as *Connect Four*, is a board game in which two players take turns dropping coloured discs into a vertical grid, trying to be the first to form a horizontal, vertical or diagonal line of four own discs. If the board fills up before either of these lines has been achieved, the game ends in a draw. The peculiarity of this connection game is that the pieces fall straight down from the top of a six-rows, seven-columns grid, occupying the lowest available space in the chosen column, as shown in Figure 1.

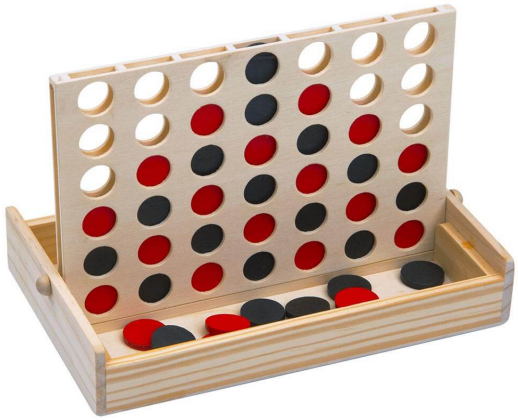

Figure 1: A *Four in a Row* game board.

The objective of this project is to design and implement a secure online version of Four in a Row, in which distant players can challenge each other by simply registering and connecting to a game server. The communications between parties are guaranteed to be confidential, authenticated and protected against replay attacks, so that even the server cannot track the progress of an on-going multiplayer match. The application is written in C++.

The document is organised as follows: Section 2 discusses the design of the application and the communication flows, with particular attention to security-related solutions; Section 3 contains the BAN logic analysis of the proposed key exchanges; Section 4 introduces the organisation of the source code and reports all the message formats.

2. Design

2.1 Overview

The application involves two main entities: a server, which represents a hub for connecting players, and a client, which represents a user willing to play.

The server is responsible for authenticating the users and enabling the matchmaking: users pre-register before using the application exploiting an external tool, and are identified by a unique username and a public key, which are securely stored at server-side. After a successful login, a user becomes effectively available for playing and can either challenge another player or receive challenges. When a matchmaking is successful, i.e. when two players both agree to compete, the server informs both clients about the identity of the opponent, sending the respective username and public key. At this point, the communication flow with the server is temporarily suspended in favour of a peer-to-peer (P2P) exchange between the involved clients.

In this additional session, the adversaries authenticate themselves exploiting the previously received information, and finally play the match. Once the latter has ended, the P2P session is dismissed and the communication flow with the server is recovered, bringing back the players to an available state. At any time, a user can decide to exit the application by informing the server about the forthcoming disconnection.

Since the flows related to the game and the matchmaking are separated, different sets of session keys are used to guarantee the confidentiality, thus neither the server is able to eavesdrop the progress or result of a match, nor a player is able to overhear what another player is asking to the server.

To ensure a correct ordering and a reliable transmission of messages during the exchanges, the TCP transport protocol is used, even in P2P sessions.

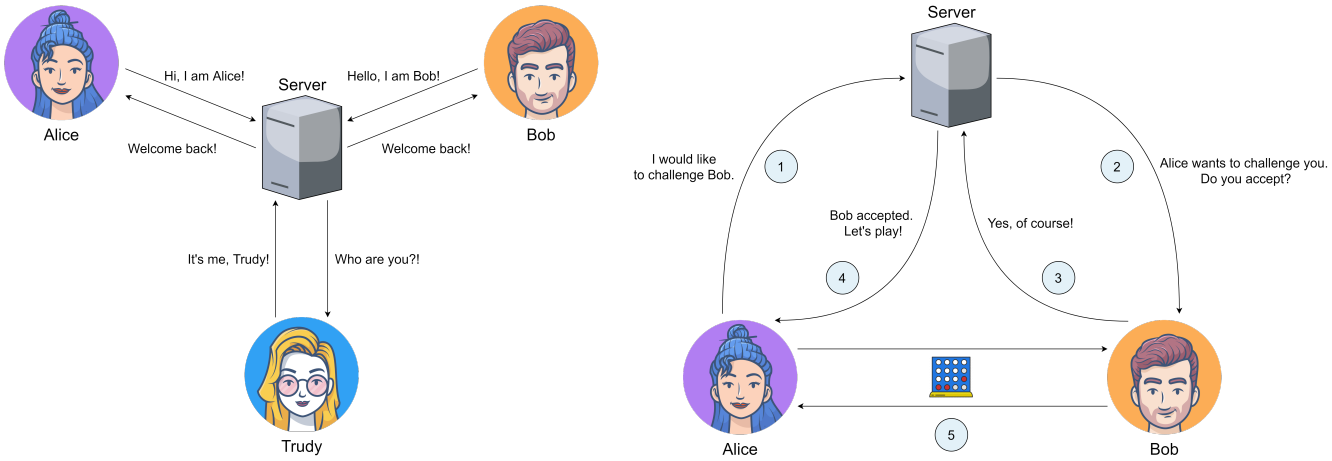

Figure 2: Example of login (left) and matchmaking (right).

2.2 Client-server handshake

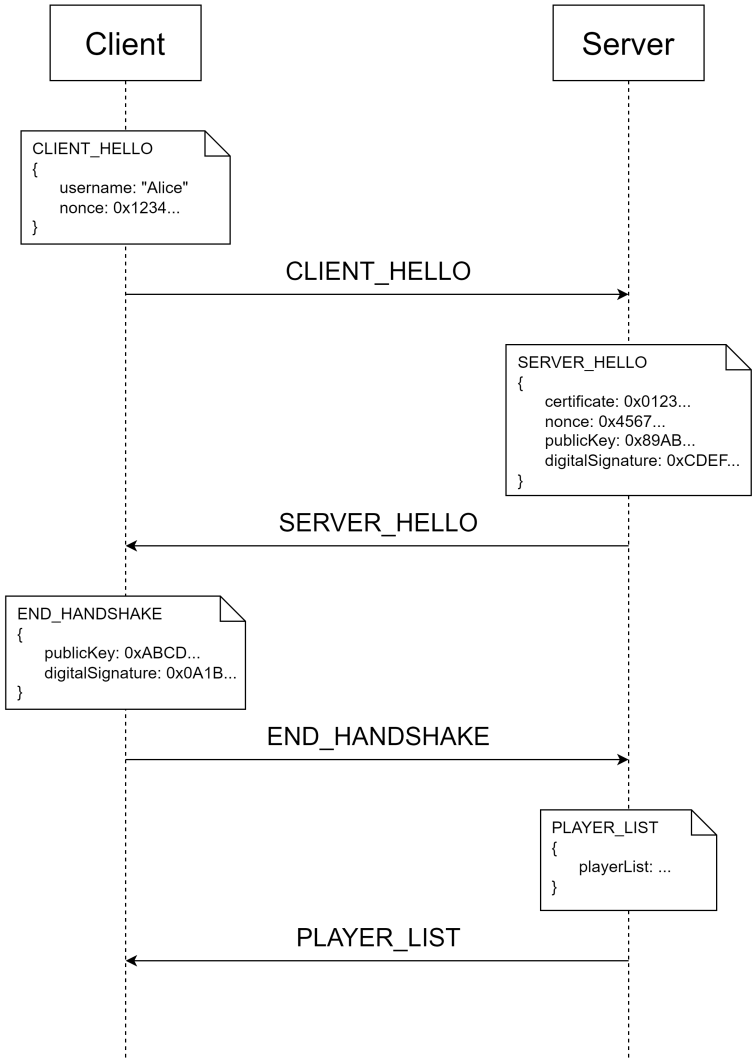

Figure 3: Sequence diagram of the client-server handshake.

The handshake between client and server aims at authenticating the user and deriving the symmetric session key employed in the forthcoming communications. The server authenticates itself using a certificate embedding an RSA-2048 public key and protected with an RSA-2048 digital signature[1]; the user is identified by a unique username and an RSA-2048 public key, both previously registered. The public keys are used to introduce digitally signed proofs of freshness in the handshake, while the actual key establishment is enabled by ephemeral elliptic-curve Diffie-Hellman (ECDHE), thus providing perfect forward secrecy. ECDHE exploits the prime256v1 curve, offering a security level of 128 bits.

[1]The choice of RSA-2048 for both the signature and the public key comes from the limitations of the free version of *SimpleAuthority*, which is the tool used to generate certificates in this project. For uniformity, the same solution is adopted for the users' public keys. A better design would employ elliptic-curve DSA to shorten the signature size without compromising security.

The initial exchange is performed entirely in cleartext and consists of the following phases:

1. the client sends a `CLIENT_HELLO` message containing the username and a 32-bit nonce;
2. the server verifies if the username is registered and a player with the same username is not already online, to avoid multiple logins. Then, it replies with a `SERVER_HELLO` containing the certificate, a 32-bit nonce, an ECDHE public key and the RSA digital signature of its proof of freshness for the session. The latter is the SHA256 hash of the concatenation of the client's nonce and the server's ECDHE public key;
3. the client verifies the certificate and the digital signature sent by the server itself. Then, it ends the handshake sending an `END_HANDSHAKE` message containing an ECDHE public key and the RSA digital signature of its proof of freshness, which is the SHA256 hash of the concatenation of the server's nonce and the client's ECDHE public key;
4. the server verifies the digital signature. If the check is passed, the handshake succeeds and both parties can derive the session key in isolation. The latter is forged by extracting the first 128 bits of the SHA256 hash of an entropy source obtained concatenating the Diffie-Hellman shared secret, the client's nonce and the server's nonce;
5. the server sends the current player list within a `PLAYER_LIST` message, which is encrypted. More details about the use of the key in encryption can be found in Section 2.3.

The verification of the proofs of freshness guarantees the integrity and authenticity of the entire handshake, avoids replay attacks and prevents identity theft. Without it, an attacker would be able to impersonate a legitimate user and perform a successful handshake. It should be noted that even an attack in which only the handshake succeeds and the attacker is not able to derive the correct session key is undesirable, because the server would still maintain an open session, consuming resources, and impede the legitimate user to go online due to the detection of multiple logins.

If the handshake fails, the server notifies the client by sending an error message whose type depends on the fault. More in details:

- a `PLAYER_ALREADY_CONNECTED` is sent if the username belongs to a player already online. Since the usernames are unique, this means that the player is attempting to login from multiple clients;
- a `PLAYER_NOT_REGISTERED` is sent if the username is not registered on the server;
- a `PROTOCOL_VIOLATION` is sent in response to a message whose type does not match the expected one (for instance, if an `END_HANDSHAKE` is received instead of a `CLIENT_HELLO`);
- a `MALFORMED_MESSAGE` is sent in response to a message whose format does not match the expected one for the type, i.e. in occurrence of serialisation errors;
- an `INTERNAL_ERROR` is sent if an internal fault occurs.

2.3 Client-server session

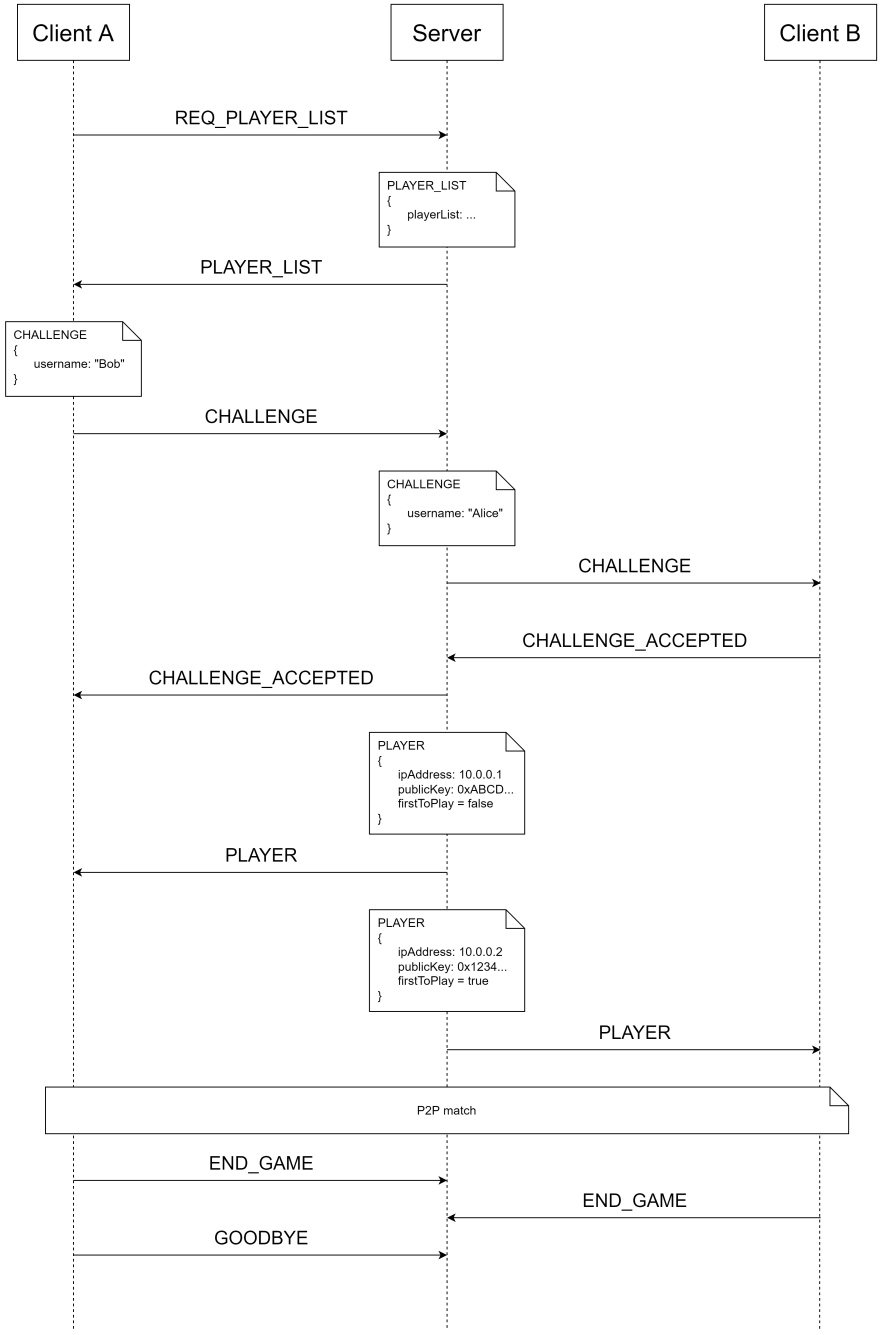

Figure 4: Sequence diagram of the client-server session.

During a client-server session, a user can retrieve the list of online players and start a matchmaking. The communications are encrypted and authenticated by means of AES-128 in Galois Counter Mode (GCM): the symmetric key is the one derived at the end of the handshake, while the initialisation vector (IV) on 96 bits is randomly generated at each encryption. The IV is sent along with the ciphertext and the tag by the encrypting party.

The exchange is protected against reordering and replay by means of sequence numbers, which are realised using synchronised 32-bit counters on both ends, and employed as additional authenticated data during the tag generation; this means that they do not travel through the network. More in details, there are two counters at each side: one dedicated to reads (decryption operations), and one dedicated to writes (encryption operations). A counter is incremented by one at each encryption or decryption, until the maximum sequence number is reached and a disconnection is enforced. The validation policy is *exact match*, i.e. the sequence numbers must precisely correspond and no window of acceptance for older values exists.

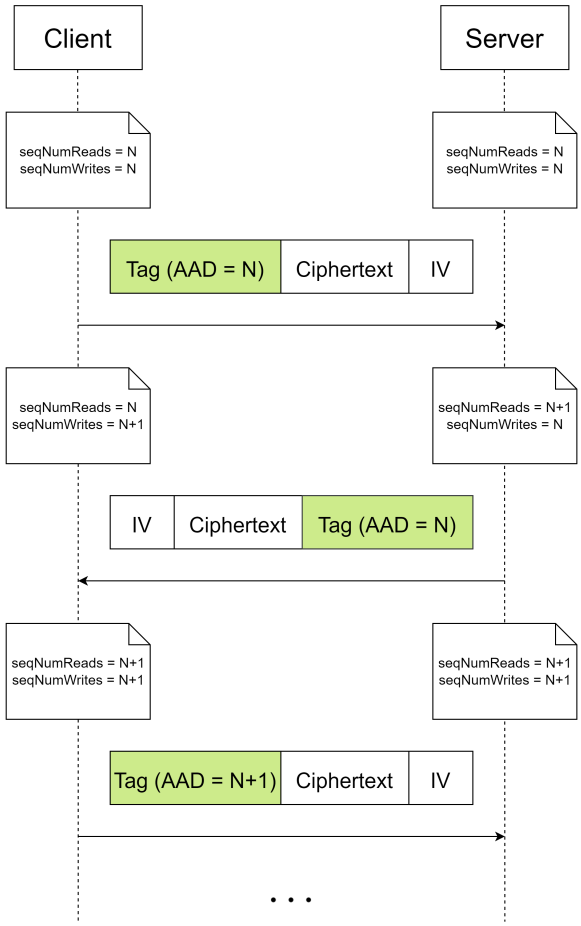

Figure 5: Example of use of the sequence numbers.

The possible interactions between client and server are the following:

- the client can retrieve the current player list by sending a `REQ_PLAYER_LIST` message, to which the server responds with a `PLAYER_LIST` message;
- the client can start a matchmaking sending a `CHALLENGE` message containing the username of the candidate opponent. A player cannot send or receive more than one challenge request at a time, and can play only one match at a time. The server propagates the request to the corresponding client, replacing the username of the challenged with the one of the challenger. The challenged can decide to refuse (`CHALLENGE_REFUSED`)

or accept (`CHALLENGE_ACCEPTED`) the proposal by responding to the server, which propagates the answer to the challenger. If the matchmaking succeeds, the server sends a `PLAYER` message containing the IP address and the RSA public key of the opponent to each client, allowing them to set up a P2P communication for playing; the server chooses which player has the first move too. Once the match has ended, a client recovers the communication with the server by sending an `END_GAME` message;

- when a client wants to disconnect, it informs the server by sending a `GOODBYE` message.

If an error occurs when exchanging messages, the server notifies the client by sending:

- a `PLAYER_NOT_AVAILABLE` if the challenged user is not available anymore at the time of reception of a `CHALLENGE` message;
- a `PROTOCOL_VIOLATION` if the received message type does not match the expected one;
- a `MALFORMED_MESSAGE` if the message has a format not matching the expected one for the type (serialisation errors), or the message does not pass the security checks (tag mismatch, invalid sequence number);
- an `INTERNAL_ERROR` if an internal fault occurs.

2.4 P2P handshake

The P2P handshake aims at authenticating both players and deriving the symmetric session key employed during the match. The design and considerations are the same of the client-server handshake, with the only difference represented by the lack of certificates, which are replaced by the trusted RSA-2048 public keys forwarded by the server at the end of the matchmaking. The public keys are used to introduce proofs of freshness, guaranteeing the integrity and authenticity of the handshake, the avoidance of replay attacks and the prevention of identity theft. The key exchange exploits ECDHE with the prime256v1 curve, offering a security level of 128 bits and providing perfect forward secrecy.

Despite being P2P, the handshake protocol is built on top of TCP and requires that one of the clients acts as server, i.e. one of them must wait for the connection of the other. The server role is taken by the client that has not the first move.

The exchange is performed in cleartext and consists of the following phases:

1. the client that has the first move, called *player 1*, sends a `PLAYER1_HELLO` message containing a 32-bit nonce;
2. the client acting as server, called *player 2*, replies with a `PLAYER2_HELLO` containing a 32-bit nonce, an ECDHE public key and the RSA digital signature of its proof of freshness for the session. The latter is the SHA256 hash of the concatenation of the player 1's nonce and the player 2's ECDHE public key;
3. player 1 verifies the digital signature sent by player 2. Then, it ends the handshake sending an `END_HANDSHAKE` message containing an ECDHE public key and the RSA digital signature of its proof of freshness, which is the SHA256 hash of the concatenation of the player 2's nonce and the player 1's ECDHE public key;

4. player 2 verifies the digital signature. If the check is passed, the handshake succeeds and both parties can derive the session key in isolation. The latter is forged by extracting the first 128 bits of the SHA256 hash of an entropy source obtained concatenating the Diffie-Hellman shared secret, the player 1's nonce and the player 2's nonce.

If the handshake fails, the client acting as server notifies the other peer by sending:

- a `PROTOCOL_VIOLATION` if the received message type does not match the expected one;
- a `MALFORMED_MESSAGE` if the message format does not match the expected one for the type, i.e. in occurrence of serialisation errors;
- an `INTERNAL_ERROR` if an internal fault occurs.

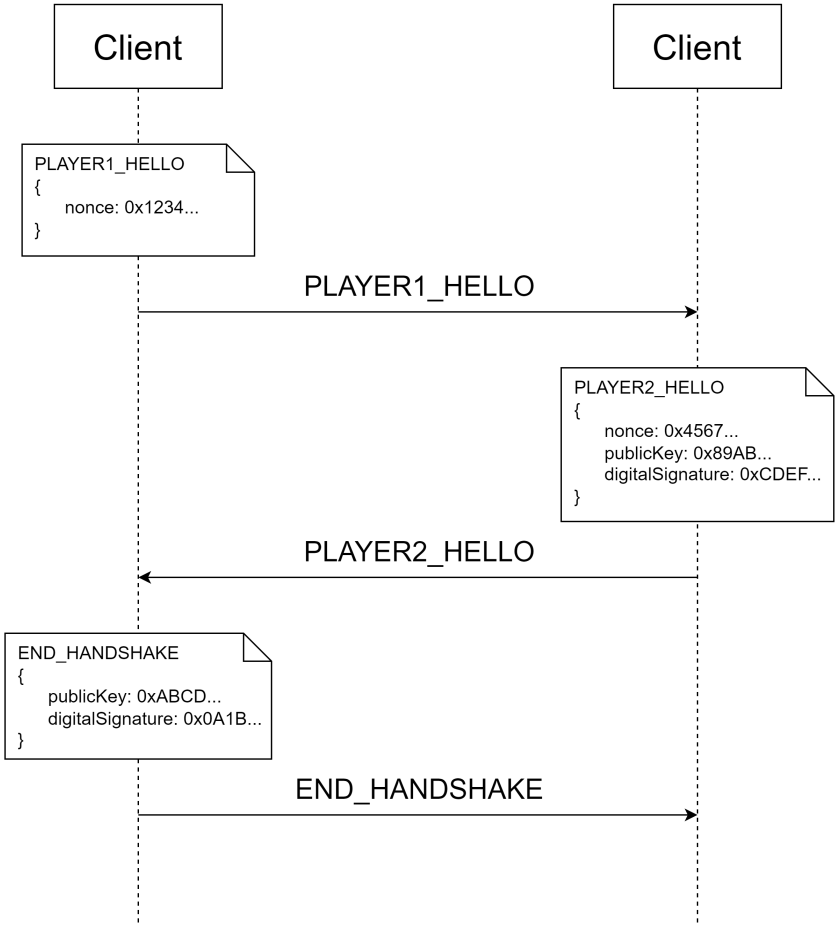

Figure 6: Sequence diagram of the P2P handshake.

2.5 P2P match

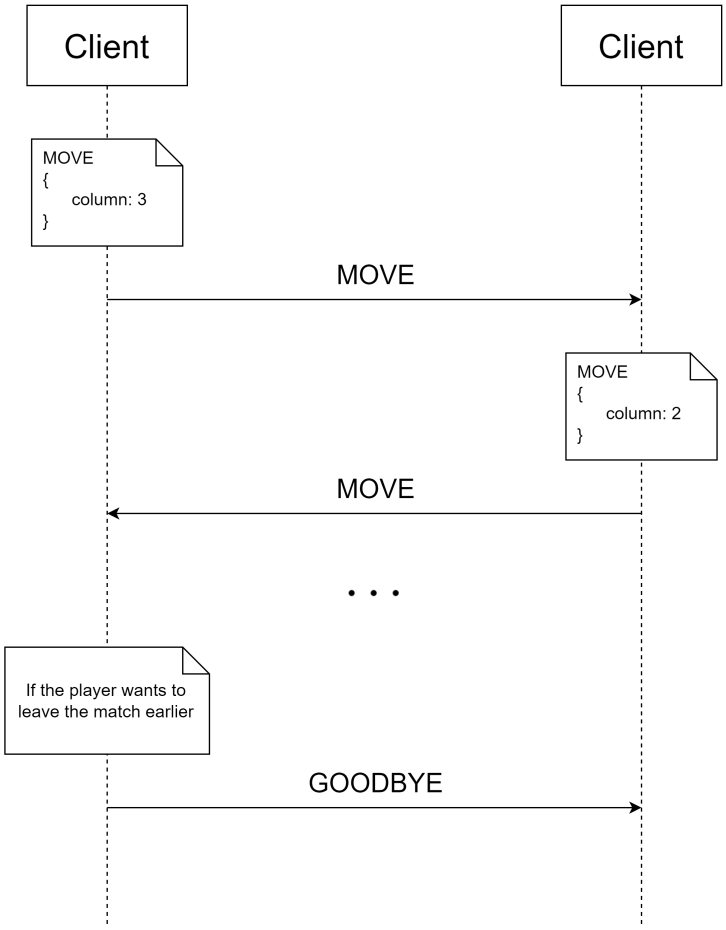

Figure 7: Sequence diagram of the P2P match.

During a P2P match, the players take turns making moves on a shared board. The status of the board is kept updated at each side, thus it does not travel through the network, and each player is able to detect independently when a match ends. A received move is always validated against the known status of the board: if a player tries to cheat submitting an invalid one, the victim client tears down the connection.

The communications are encrypted, authenticated and secure against replay attacks thanks to the same mechanisms used in the client-server session, i.e. AES-128 GCM with a 96-bit random IV generated at each encryption, and 32-bit sequence numbers implemented using synchronised counters.

3. BAN logic analysis

3.1 Client-server key exchange

3.1.1 Notation and idealised protocol

During the analysis, the entities in the application domain are denoted as follows:

- C and S are the client and the server, respectively;
- CA is the certification authority trusted by the client and the server;
- U_C is the username of the client;
- N_C and N_S are the nonces generated by the client and the server, respectively;
- Y_C and Y_S are the ECDHE public keys generated by the client and the server, respectively;
- E_C and E_S are the long-term RSA public keys of the client and the server, respectively;
- E_{CA} is the RSA public key of the certification authority;
- $\{E_S\}_{E_{CA}^{-1}}$ is the certificate of the server, released by the certification authority;
- K_{CS} is the symmetric session key derived at the end of the handshake;
- h is a hash function.

Since the BAN analysis aims at drawing conclusions about the correctness of the key exchange, only the `CLIENT_HELLO`, `SERVER_HELLO` and `END_HANDSHAKE` messages are investigated. Therefore, the idealised protocol is a direct transcription of the first three messages described in Section 2.2:

$$C \to S : U_C, N_C \tag{M1}$$

$$S \to C : \{E_S\}_{E_{CA}^{-1}}, N_S, Y_S, \{h(N_C, Y_S)\}_{E_S^{-1}} \tag{M2}$$

$$C \to S : Y_C, \{h(N_S, Y_C)\}_{E_C^{-1}} \tag{M3}$$

3.1.2 Postulates

- **Message meaning**: if P believes K is Q's public key and P sees a message signed by K^{-1} containing X, then P believes that X was sent by Q.

$$\frac{P \mid\equiv \overset{K}{\mapsto} Q,\ P \triangleleft \{X\}_{K^{-1}}}{P \mid\equiv Q \mid\sim X} \tag{P1}$$

- **Nonce verification**: if P believes Q sent X and P believes X is fresh, then P believes Q believes X, namely it believes Q has sent X in this protocol execution instance.

$$\frac{P \mid\equiv \#(X),\ P \mid\equiv Q \mid\sim X}{P \mid\equiv Q \mid\equiv X} \tag{P2}$$

- **Jurisdiction**: if P believes Q believes X and P trusts Q on X, then P believes X too.

$$\frac{P \mid\equiv Q \mid\equiv X,\ P \mid\equiv Q \Rightarrow X}{P \mid\equiv X} \tag{P3}$$

- **Composite see**: if P sees the message $(X_1, \ldots, X_N)$, then P sees any element X_i composing the message.

$$\frac{P \triangleleft (X_1, \ldots, X_N)}{P \triangleleft X_i} \quad i = 1, \ldots, N \tag{P4}$$

- **Composite believe**: if P believes X_1,..., P believes X_N, then P believes $(X_1, \ldots, X_N)$, and vice versa. The same holds for predicates of the form $P \mid\equiv Q \mid\equiv X_1$,...,$P \mid\equiv Q \mid\equiv X_N$.

$$\frac{P \mid\equiv X_1, \ldots, P \mid\equiv X_N}{P \mid\equiv (X_1, \ldots, X_N)} \quad \frac{P \mid\equiv (X_1, \ldots, X_N)}{P \mid\equiv X_1, \ldots, P \mid\equiv X_N} \tag{P5}$$

- **Freshness**: if P believes one or more quantities $X_1, \ldots, X_N$ are fresh, then P believes the message $(X_1, \ldots, X_N, Y_1, \ldots, Y_M)$ is fresh.

$$\frac{P \mid\equiv \#(X_1, \ldots, X_N)}{P \mid\equiv \#(X_1, \ldots, X_N, Y_1, \ldots, Y_M)} \tag{P6}$$

- **Hash function**: if P believes Q has sent the hash of $X_1, \ldots, X_N$ and P sees $X_1, \ldots, X_N$, then P believes Q has sent $X_1, \ldots, X_N$, i.e. believes Q has seen $X_1, \ldots, X_N$.

$$\frac{P \mid\equiv Q \mid\sim h(X_1, \ldots, X_N),\ P \triangleleft X_1, \ldots, P \triangleleft X_N}{P \mid\equiv Q \mid\sim (X_1, \ldots, X_N)}; \tag{P7}$$

- **Username verification**: if P believes the username U_Q, i.e. knows that the username is registered and not used by an already online player, and P sees the username, then P believes key E_Q is the registered

public key of Q. The postulate does not imply that P believes E_Q^{-1} is owned by Q in this protocol execution instance.

$$\frac{P \mid\equiv U_Q,\ P \triangleleft U_Q}{S \mid\equiv \overset{E_Q}{\longmapsto} Q} \tag{P8}$$

- **Certificate verification**: if P sees a certificate signed by T, P believes T has public key K_T and P trusts T about certifying Q's public key, then P believes K_Q is the public key of Q.

$$\frac{P \mid\equiv \overset{K_T}{\longmapsto} T,\ P \triangleleft \{K_Q\}_{K_T^{-1}},\ P \mid\equiv T \Rightarrow (\overset{K_Q}{\longmapsto} Q)}{P \mid\equiv \overset{K_Q}{\longmapsto} Q} \tag{P9}$$

3.1.3 Assumptions

- $C \mid\equiv \#(N_C)$
- $C \mid\equiv \#(Y_C)$
- $S \mid\equiv \#(N_S)$
- $S \mid\equiv \#(Y_S)$
- $C \mid\equiv \overset{E_{CA}}{\longmapsto} CA$
- $S \mid\equiv \overset{E_{CA}}{\longmapsto} CA$
- $C \mid\equiv CA \Rightarrow (\overset{E_S}{\longmapsto} S)$
- $C \mid\equiv S \Rightarrow (N_S, Y_S)$
- $S \mid\equiv C \Rightarrow (N_C, Y_C)$

3.1.4 Proof

The proof aims at achieving:

- **Key authentication**: $C \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S,\ S \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S$;
- **Key confirmation**: $C \mid\equiv S \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S,\ S \mid\equiv C \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S$;
- **Key freshness**: $C \mid\equiv \#(C \overset{K_{CS}}{\longleftrightarrow} S),\ S \mid\equiv \#(C \overset{K_{CS}}{\longleftrightarrow} S)$.

In (M1), the server S receives the username and the nonce of the client. From (P4), it holds that $S \triangleleft U_C,\ S \triangleleft N_C$; moreover, after the validity check of the username, also $S \mid\equiv U_C$ holds. Therefore, by applying the postulate (P8):

$$\frac{S \mid\equiv U_C,\ S \triangleleft U_C}{S \mid\equiv \overset{E_C}{\longmapsto} C}$$

It should be noted that, at this point, neither S believes the freshness of N_C, nor it believes that C owns the private key E_C^{-1}.

In (M2), the client receives the certificate, the nonce and the ECDHE public key of the server, together with the digital signature of its proof of freshness. Again, from (P4), the client sees ($\triangleleft$) all the quantities composing the message. The first check involves the certificate, by means of (P9):

$$\frac{C \mid\equiv \overset{E_{CA}}{\longmapsto} CA,\ C \triangleleft \{E_S\}_{E_{CA}^{-1}},\ C \mid\equiv CA \Rightarrow (\overset{E_S}{\longmapsto} S)}{C \mid\equiv \overset{E_S}{\longmapsto} S}$$

The second check involves the proof of freshness and results in the application of postulate (P1):

$$\frac{C \mid\equiv \overset{E_S}{\longmapsto} S,\ C \triangleleft \{h(N_C, Y_S)\}_{E_S^{-1}}}{C \mid\equiv S \mid\sim h(N_C, Y_S)}$$

The last belief can be combined with (P7) to obtain:

$$\frac{C \mid\equiv S \mid\sim h(N_C, Y_S),\ C \triangleleft N_C,\ C \triangleleft Y_S}{C \mid\equiv S \mid\sim (N_C, Y_S)}$$

From (P6) and the assumptions, it holds $C \mid\equiv \#(N_C, Y_C)$ and $C \mid\equiv \#(N_C, N_S, Y_S)$. These results allow to apply the postulate (P2):

$$\frac{C \mid\equiv \#(N_C, N_S, Y_S),\ C \mid\equiv S \mid\sim (N_C, N_S, Y_S)}{C \mid\equiv S \mid\equiv (N_C, N_S, Y_S)}$$

Moreover, it is possible to apply (P5) and (P3) to obtain:

$$\frac{C \mid\equiv S \mid\equiv (N_S, Y_S),\ C \mid\equiv S \Rightarrow (N_S, Y_S)}{C \mid\equiv (N_S, Y_S)}$$

At this point, the client generates its ECDHE public key and its proof of freshness, and sends (M3). Then, the conclusions for the client can be drawn. Indeed, it holds:

- **Key authentication**: $C \mid\equiv (N_C, Y_C, N_S, Y_S)$ because of C believing its generated quantities by definition, and because of the last step result and postulate (P5). This means that C can derive K_{CS}, and so $C \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S$;
- **Key confirmation**: $C \mid\equiv S \mid\equiv (N_C, Y_C, N_S, Y_S)$ because of the second-last step result and postulate (P5). Therefore, C believes that S is able to derive K_{CS}, and so $C \mid\equiv S \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S$;
- **Key freshness**: $C \mid\equiv \#(N_C, Y_C, N_S, Y_S)$ because of the assumptions and postulate (P6). Then, due to the key derivation procedure, it holds $C \mid\equiv \#(C \overset{K_{CS}}{\longleftrightarrow} S)$.

For what concerns the server, in (M3) it receives the digital signature of the proof of freshness produced by the client. From this point on, the reasoning is the same one discussed for (M2), apart the certificate verification.

By applying (P1), it holds:

$$\frac{S \mid\equiv \overset{E_C}{\longmapsto} C,\ S \triangleleft \{h(N_S, Y_C)\}_{E_C^{-1}}}{S \mid\equiv C \mid\sim h(N_S, Y_C)}$$

Therefore, from (P7):

$$\frac{S \mid\equiv C \mid\sim h(N_S, Y_C),\ S \triangleleft N_S,\ S \triangleleft Y_C}{S \mid\equiv C \mid\sim (N_S, Y_C)}$$

Applying (P6) and the assumptions, it holds $S \mid\equiv \#(N_S, Y_S)$ and $S \mid\equiv \#(N_C, N_S, Y_C)$. Therefore, using postulate (P2) results in:

$$\frac{S \mid\equiv \#(N_C, N_S, Y_C),\ S \mid\equiv C \mid\sim (N_C, N_S, Y_C)}{S \mid\equiv C \mid\equiv (N_C, N_S, Y_C)}$$

From (P5) and (P3):

$$\frac{S \mid\equiv C \mid\equiv (N_C, Y_C),\ S \mid\equiv C \Rightarrow (N_C, Y_C)}{S \mid\equiv (N_C, Y_C)}$$

Finally, the conclusions for the server can be drawn. Indeed, it holds:

- **Key authentication**: $S \mid\equiv (N_C, Y_C, N_S, Y_S)$ because of S believing its generated quantities by definition, and because of the last step result and postulate (P5). This means that S can derive K_{CS}, and so $S \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S$;
- **Key confirmation**: $S \mid\equiv C \mid\equiv (N_C, Y_C, N_S, Y_S)$ because of the second-last step result and postulate (P5). Therefore, S believes that C is able to derive K_{CS}, and so $S \mid\equiv C \mid\equiv C \overset{K_{CS}}{\longleftrightarrow} S$;
- **Key freshness**: $S \mid\equiv \#(N_C, Y_C, N_S, Y_S)$ because of the assumptions and postulate (P6). Then, due to the key derivation procedure, it holds $S \mid\equiv \#(C \overset{K_{CS}}{\longleftrightarrow} S)$.

3.2 P2P key exchange

The proof of the P2P key exchange is basically the same one introduced in the previous section, being the two protocols nearly identical. Thus, the proof is not entirely reported: hereafter, only the minor differences between the two versions are described.

3.2.1 Notation and idealised protocol

The notation stays intact w.r.t. the client-server proof, but now C denotes the player acting as client (player 1) and S the player acting as server (player 2). There are no certification authorities, certificates and usernames involved in the exchange, with the idealised protocol that becomes:

$$C \to S : N_C \tag{M1}$$

$$S \to C : N_S, Y_S, \{h(N_C, Y_S)\}_{E_S^{-1}} \tag{M2}$$

$$C \to S : \{Y_C, h(N_S, Y_C)\}_{E_C^{-1}} \tag{M3}$$

3.2.2 Postulates

The postulates can be reused as they are, except for the username verification (P8) and certificate verification (P9) ones, that have no meaning in this context.

3.2.3 Assumptions

The assumptions $C \mid\equiv \overset{E_{CA}}{\longmapsto} CA$, $S \mid\equiv \overset{E_{CA}}{\longmapsto} CA$, $C \mid\equiv CA \Rightarrow (\overset{E_S}{\longmapsto} S)$ are replaced by:

- $C \mid\equiv \overset{E_S}{\longmapsto} S$
- $S \mid\equiv \overset{E_C}{\longmapsto} C$

The beliefs about the long-term public keys are achieved when the server sends a `PLAYER` message at the end of the matchmaking, which includes the information about the opponent. Since the session with the server is secure, the beliefs can be treated as assumptions when the handshake starts.

3.2.4 Proof

The steps and reasoning are the same, with the following minor differences:

1. in (M1), the server only sees ($\triangleleft$) the received quantities, but does not acquire any belief, since no verification of the username is done. Here, the belief regarding the public key of the client is an assumption;
2. in (M2), the client does not verify the server's certificate. The belief regarding the public key of the server is an assumption.

4. Implementation

4.1 Overview

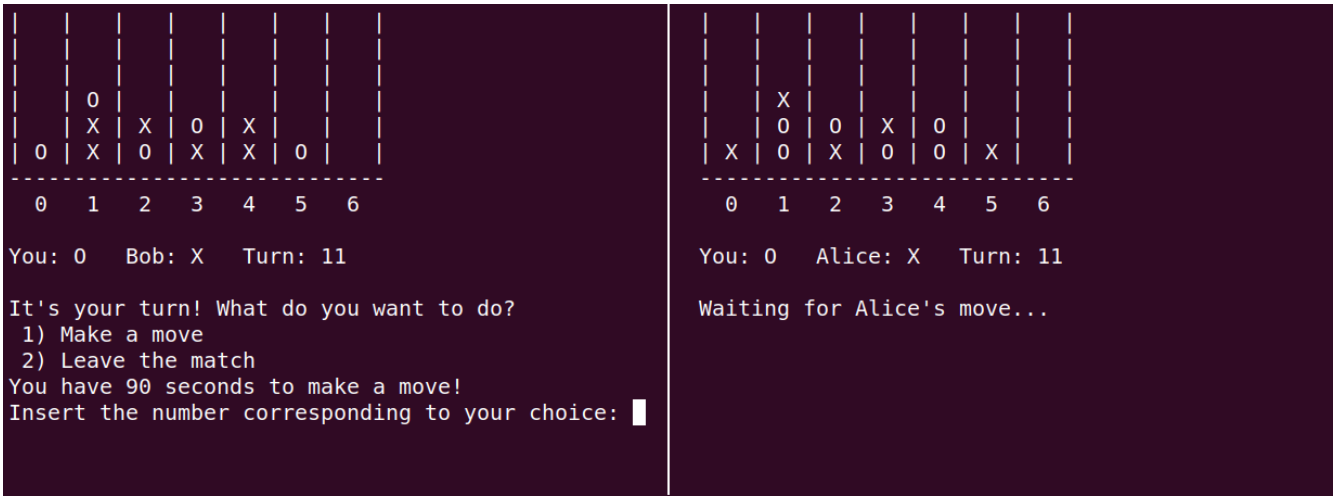

Figure 8: Example of a P2P match between Alice (left) and Bob (right).

The application is developed in C++ and exploits the OpenSSL library for cryptographic operations. The code has been tested on Ubuntu 20.04.3 LTS and is organised in the following folders:

- `crypto`, containing the cryptographic library. It is composed of a set of C++ wrappers for the OpenSSL API, offering utilities for random number generation, authenticated encryption, digital signatures, certificate verification, key exchange, hashing and serialisation of public keys and certificates;
- `socket`, containing the networking library. It offers C++ wrappers for C sockets and I/O multiplexing;
- `message`, containing the messages exchanged between parties. Each message has a dedicated class inheriting from `Message`, with custom serialisation to binary format. The messages consisting only of a type, for instance `PROTOCOL_VIOLATION`, `GOODBYE`, etc., are grouped into the same class called `InfoMessage`;
- `game`, containing the manager of the game board and a class representing a player together with the quantities derived in the handshake;
- `exception`, containing the custom exceptions used in the project. The exceptions are basically clones of `std::runtime_error` and are used to distinguish the different errors that can be generated at runtime;
- `utils`, containing utility methods and constants used by all the libraries;

- `client`, containing the source code of the client application, the certification authority's certificate and the user's keys. It exploits a set of handlers to manage the handshake, the interaction with the user and the P2P game;
- `server`, containing the source code of the server application, the server's certificate and the registered users' public keys. It exploits a set of handlers to manage the lifetime of connected clients. Each handler is named as the client state concatenated with the word *Handler*. The server is single threaded and manages multiple clients concurrently by adopting the I/O multiplexing technique.

More details about how the code works can be found in the documentation inside the source files.

4.2 Message formats

In the previous sections, the term *message* was used to refer to the unit of communication at application level. To be more precise, a message is not encapsulated directly inside a TCP segment, but has additional information attached, whose format depends on the exchange being in cleartext or ciphertext. Henceforth, a more precise jargon is used, in particular:

- the term *message* is used to refer to the meaningful data exchanged between entities, i.e. the data that effectively determine the behaviour of the application, like the information travelling inside the `CLIENT_HELLO` or `MOVE` units. A message can be encrypted (and authenticated) or not to be sent through the network;
- the term *packet* is used to refer to the actual unit of communication at application level, i.e. the one sent through a socket and embedded inside a TCP segment. A packet wraps an encrypted message, together with the information needed for its decryption and authentication, or an unencrypted message. It is composed of a 16-bit length field and a payload of variable size.

Given this distinction, a packet can have only two formats, shown in Figure 9. An exchange performed in cleartext causes the payload to be simply a message, while a confidential and authenticated exchange leads to the payload being the concatenation of the IV, the ciphertext and the tag; the length field is neither encrypted nor authenticated.

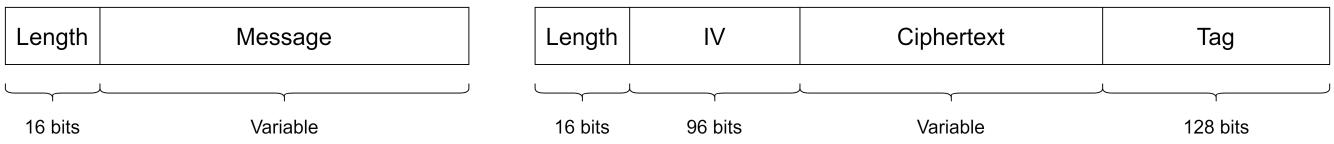

Figure 9: The packet format for exchanges in cleartext (left) and the one for secure exchanges (right).

For what concerns messages, the formats are similar to the ones sketched in Section 2, where each message is identified by a type on 8 bits. The sizes of ECDHE and RSA public keys do not match the theoretical ones, due to the serialisation format adopted by OpenSSL. All the message formats are presented below:

- a `CLIENT_HELLO` message introduces the client during the client-server handshake;

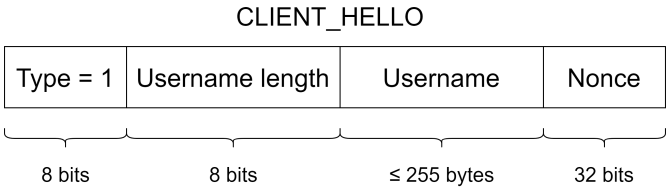

Figure 10: The format of a `CLIENT_HELLO` message.

- a `SERVER_HELLO` message introduces the server during the client-server handshake;

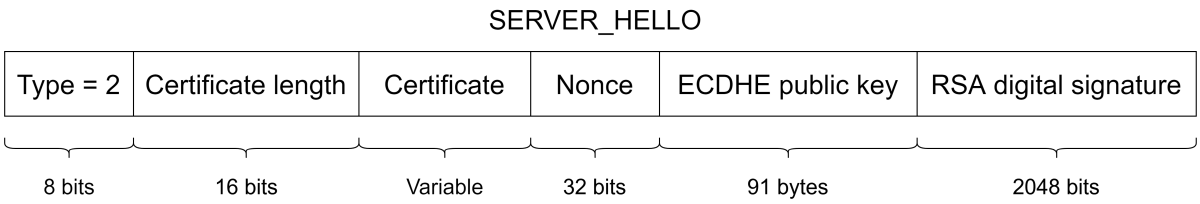

Figure 11: The format of a `SERVER_HELLO` message.

- a `PLAYER1_HELLO` message introduces the player acting as client during the P2P handshake;

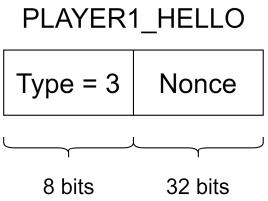

Figure 12: The format of a `PLAYER1_HELLO` message.

- a `PLAYER2_HELLO` message introduces the player acting as server during the P2P handshake;

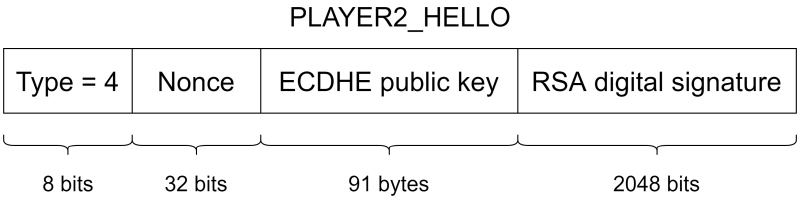

Figure 13: The format of a `PLAYER2_HELLO` message.

- an `END_HANDSHAKE` message terminates a client-server or P2P handshake;

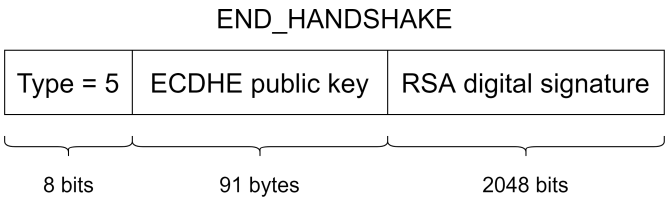

Figure 14: The format of an `END_HANDSHAKE` message.

- a `PLAYER_LIST` message carries the current player list, where each username is separated by a ";" character;

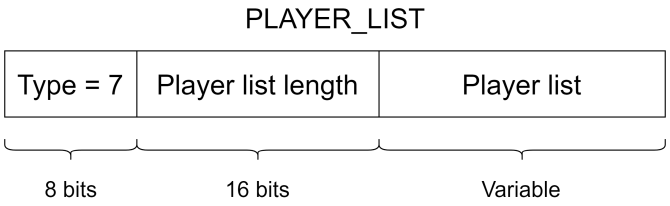

Figure 15: The format of a `PLAYER_LIST` message.

- a `CHALLENGE` message carries a request for playing. The username identifies the challenged if the message is sent to the server, the challenger if the message is sent to another user;

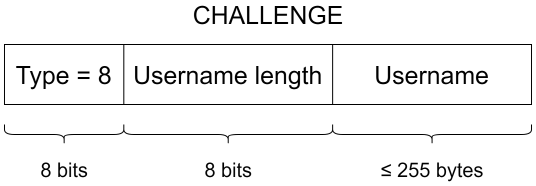

Figure 16: The format of a `CHALLENGE` message.

- a `PLAYER` message carries the information about the opponent. The `First to play` field is a boolean equal to 1 if the player has the first move, 0 otherwise;

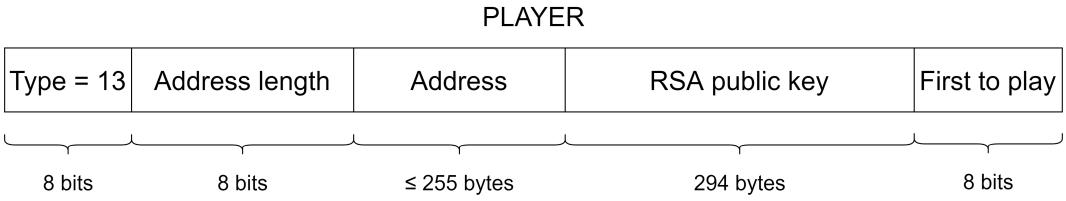

Figure 17: The format of a `PLAYER` message.

- a `MOVE` message carries a game move.

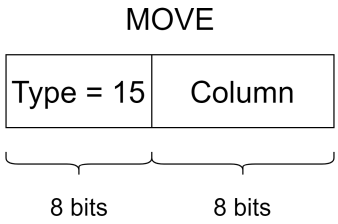

Figure 18: The format of a `MOVE` message.

- the messages that carry no information other than their types are `REQ_PLAYER_LIST` (type 6), `PLAYER_NOT_AVAILABLE` (type 9), `PLAYER_ALREADY_CONNECTED` (type 10), `CHALLENGE_REFUSED` (type 11), `CHALLENGE_ACCEPTED` (type 12), `GOODBYE` (type 14), `END_GAME` (type 16), `PLAYER_NOT_REGISTERED` (type 17), `PROTOCOL_VIOLATION` (type 18), `MALFORMED_MESSAGE` (type 19) and `INTERNAL_ERROR` (type 20). They are used to notify errors, send requests and responses not requiring additional data.

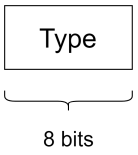

Figure 19: The format of a message carrying only its type.