Four-in-a-Row Online – Project Report

Mirko Laruina, Riccardo Mancini

June 18, 2020

Contents

1 Project Specification

The project consists in the implementation of a four-in-a-row application. The user is proposed with the possibility of playing offline (against the computer) or online.

When playing online, the communication is encrypted and authenticated. Both the server and the clients have a certificate. The server holds all the certificates of the users registered to it and, when a game has to start between two players, sends the certificate of the other party to the users.

The two users establish a connection between themselves and can start playing their game.

We can distinguish two phases during our application lifetime:

- Server lobby
- Peer to peer

During the first phase, the user can see all the active users and can decide who (s)he wants to challenge. The peer to peer phase is the one in which the match happens. After it is concluded, both players are brought back to the server lobby.

Notice that we have chosen to implement the peer to peer phase as a standalone gaming mode, even if it wasn't a functional requirement, since it is useful for debugging purposes and its implementation is quite easy. Between clients no certificate exchange is possible so the users can challenge only peers of which they know the certificate.

1.1 Application run

When a user runs the application, (s)he will be proposed with a welcoming screen in which all the commands are listed.

- *server host port [certificate]*: the user connects to the server listening on the provided host and port. The certificate is an optional parameter. If it is specified, it will be the one used for the communication, otherwise a `CERTIFICATE_REQUEST` message will be issued to which the server will reply with its own certificate. The user is then brought in server lobby.
- *peer host port certificate*: the user connects to the peer listening on the provided host and port. The certificate has to be provided, otherwise the communication will not be possible. If the connection is successful, the match can begin.
- *peer listen_port certificate*: the user will listen on the specified port and will wait for a connection from the peer having the specified certificate. The match will begin when the other peer connects to the host.

1.2 Server lobby

In this mode, the user connects to a server. The server is responsible of handling all the active users, forwarding incoming challenges and setting up a match if the parties agree on it.

An active user is a player actively connected to the server, he can be free or playing a match against another user.

In particular, the server accepts different commands:

- *list* : shows the list of all the active users

- *challenge username*: challenges the user with the specified username, if (s)he is available (s)he will be asked to accept or decline the challenge.
- *exit*: ends the communication with the server

When a challenge is accepted, the user receives the certificate of the challenger (so does the other player) and he can start communicating with him as if it was a peer to peer connection.

1.3 Peer to peer

The peer to peer phase, which can be standalone as previously specified or following the starting of a match decided in the server lobby, is the phase in which the users connect directly to eachother.

In order to be possible, one of the peers should listen to a port and the other should connect to it. In server mode, the information on the host and port are forwarded by the server itself, while if connecting directly to the peer, we have to have this knowledge beforehand.

The existence of NATs makes this mode difficult to use when not in the LAN.

1.4 Remarks

Since there are no main differences between the server and clients capabilities, the protocol used is the same both for the communication server-client than peer-peer.

2 Communication Protocol

2.1 Description

The communication protocol is the same for both the client-server and client-client communications. It has been adapted from TLS 1.3 (RFC 8446[1]). The choice of this adaptation is due to the fact that the requested specifications closely match the features of TLS (e.g. perfect forward secrecy, authentication, message integrity, etc.).

The communication protocol is split in two parts:

- **handshake protocol**: establishes a shared key among the two parties.
- **record protocol**: sends an encrypted message to the other party.

2.1.1 Handshake Protocol

The differences of the proposed handshake protocol from TLS 1.3 handshake protocol using ECDH are the following:

- **No cipher suite negotiation** since we are going to always use the same algorithms.
- **No certificate exchange**: certificates are exchanged before the connection (either by explicitly asking for it, e.g. client connecting to the server, or from a previous knowledge, e.g. clients receiving the peer certificate from the server at the start of the game).
- **certificate verification and finished messages are collapsed in the same message**: it would have been redundant to send both HMAC and DS. Furthermore, the DS is computed on just the key parameters (ephemeral public keys, nonces and identities) instead of the whole communication transcript.

Here are the steps that the parties A and B need to perform, shown also in figure 1:

1. **A generates** her random nonce N_A and her ECDH ephemeral public key K_A^E.
2. **A sends to B** her identifier A, B's identifier B, N_A and K_A^E.
3. **B generates** his random nonce N_B and his ECDH ephemeral public key K_B^E.
4. **B computes** the digital signature using his private key K_B over the plaintext:

$$(A, B, N_A, K_A^E, N_B, K_B^E)$$

5. **B sends to A** his identifier B, A's identifier A, N_B, K_B^E and his digital signature S_B.
6. **A verifies** B's signature.
7. **A computes** the digital signature using her private key K_A over the same plaintext as B did.
8. **A sends to B** the digital signature.
9. **B verifies** A's signature.

When the handshake completes, both parties can generate the shared secrets using the key derivation function *HKDF*:

- *client write key*
- *client write IV*

[1] `https://tools.ietf.org/html/rfc8446`

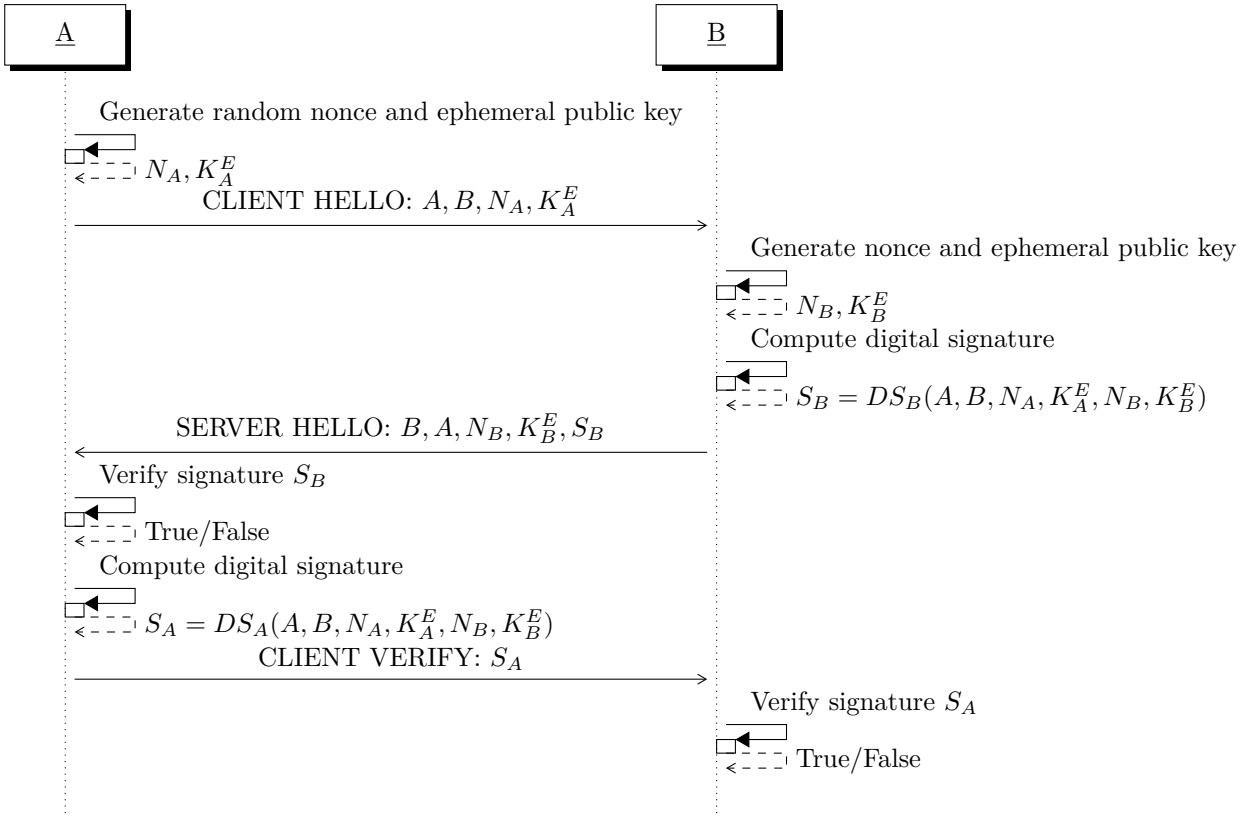

Figure 1: Sequence diagram of the handshake protocol. A and B are the identities of the two parties, N_A and N_B are random nonces, K_A^E and K_B^E are ECDH ephemeral public keys, K_A and K_B are the parties' public keys, S_A and S_B are the digital signatures.

- *server write key*
- *server write IV*

Furthermore, both parties initialize two 64 bit counters to 0, which will represent the sequence number of the client and server messages. These sequence numbers are not sent over the network but are updated at any new received message.

2.1.2 Record Protocol

If A wants to send a message to B she needs to:

1. build the message to send.
2. encrypt the message using AES-128 in GCM using the length of the message and the byte 0 (the type of the record message) as AAD. The IV is the IV generated by the *HKDF*, where the first 64 bits are XOR-ed with the 64-bit sequence number.
3. build the new message as length, 0 (message type), AES-GCM tag, ciphertext.
4. send the message.
5. increase the write sequence number.

If B wants to receive a message from A he needs to:

1. receive the full message.

2. decrypt the message following the same instructions as before.
3. check the tag
4. increase the read sequence number.

2.2 BAN-logic proof

2.2.1 Real protocol

From the description before, the formal specification of the protocol easily follows as presented below.

$$M1\ A \rightarrow B : A, B, N_A, K_A^E$$
$$M2\ B \rightarrow A : B, A, N_B, K_B^E, \{h(A, B, N_A, N_B, K_A^E, K_B^E)\}_{k_B^{-1}}$$
$$M3\ A \rightarrow B : \{h(A, B, N_A, N_B, K_A^E, K_B^E)\}_{k_A^{-1}}$$

2.2.2 Assumptions

Note that since we are using certificates to authenticate A and B public keys, we can assume that they both know the public key of the other.

$$A \mid\equiv \overset{K_B}{\longmapsto} B \qquad B \mid\equiv \overset{K_A}{\longmapsto} A$$
$$A \mid\equiv \#(\overset{K_A^E}{\longmapsto} A) \qquad B \mid\equiv \#(\overset{K_B^E}{\longmapsto} B)$$
$$A \mid\equiv \#(N_A) \qquad B \mid\equiv \#(N_B)$$
$$A \mid\equiv B \Rightarrow \overset{K_B^E}{\longmapsto} B \qquad B \mid\equiv A \Rightarrow \overset{K_A^E}{\longmapsto} A$$
$$A \mid\equiv B \Rightarrow N_B \qquad B \mid\equiv A \Rightarrow N_A$$

2.2.3 Expected Conclusions

In the end of the protocol, we would like that they both know all parameters to subsequently generate the shared secrets and that they are certain that the other peer knows them too.

$$A \mid\equiv (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B) \qquad B \mid\equiv (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B)$$
$$A \mid\equiv B \mid\equiv (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B) \qquad B \mid\equiv B \mid\equiv (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B)$$

2.2.4 Idealized protocol

By digitally signing all nonces and ephemeral keys, they are proving the other that they know them. Therefore in the idealized protocol the hash is removed for clarity's sake.

$$M1\ A \rightarrow B : N_A, \overset{K_A^E}{\longmapsto} A$$
$$M2\ B \rightarrow A : \{N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B\}_{k_B^{-1}}$$
$$M3\ A \rightarrow B : \{N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B\}_{k_A^{-1}}$$

2.2.5 Proof

After message 1 From the first message, B just knows SOMEONE sent him a nonce and an ephemeral public key. He will use these values in later exchanges but at the moment he knows nothing about the sender.

$$M1\ A \to B : N_A, \overset{K_A^E}{\longmapsto} A \qquad \text{B does not know anything about the sender}$$
$$B \triangleleft (N_A, \overset{K_A^E}{\longmapsto} A)$$

After message 2 From this message A is able to verify B's identity and has confirmation that B actually knows the nonce and her ephemeral public key. A also receives B's nonce and ephemeral public key.

$$M2\ B \to A : \{N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B\}_{k_B^{-1}} \qquad \text{apply message meaning postulate}$$
$$A \mid\equiv B \mid\sim (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B) \qquad \text{apply nonce verification postulate}$$
$$A \mid\equiv B \mid\equiv (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B) \qquad \text{furthermore, due to juristiction postulate}$$
$$A \mid\equiv (N_B, \overset{K_B^E}{\longmapsto} B)$$

After message 3 From this message B is able to verify A's identity and has confirmation that A actually knows the nonce and his ephemeral public key.

$$M3\ A \to B : \{N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B\}_{k_A^{-1}} \qquad \text{apply message meaning postulate}$$
$$B \mid\equiv A \mid\sim (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B) \qquad \text{apply nonce verification postulate}$$
$$B \mid\equiv A \mid\equiv (N_A, N_B, \overset{K_A^E}{\longmapsto} A, \overset{K_B^E}{\longmapsto} B) \qquad \text{furthermore, due to juristiction postulate}$$
$$B \mid\equiv (N_A, \overset{K_A^E}{\longmapsto} A)$$

2.2.6 Final remarks

It is worthwhile to note that the nonces N_A, N_B play no role in the proof of the protocol. In fact, the nonce verification postulate could be applied also on the ephemeral public keys (since they are fresh too). Furthermore, the shared DH secret is guaranteed to always be different since both parties choose an ephemeral public key at random at every handshake. Therefore, we could remove the random nonces from the protocol without loss of security.

3 Implementation

3.1 Development

During all the development of the application, we posed particular attention to the modularity of the software.

In particular, our development was divided in different phases which acted as building blocks:

- *offline* : the basic game was implemented (CLI, win conditions etc.)
- *networking* : a new layer was added and allowed playing over the network in an "insecure" way (all messages in the clear). To avoid having to modify the code heavily in the next phase, an interface to utilize the socket was defined (*SocketWrapper*).
- *security*: the cryptographic algorithms were implemented using the OpenSSL library and a *SecureSocketWrapper* was defined and used in the client and server applications.

To speed up the development, a *Makefile* and some automated tests were written to automatically compile the code and check that the behavior of the application was as designed. By simply running `make test` all three possible game modes are tested by spawning dummy processes.

3.2 Source code overview

The source code is organized into 5 modules:

- *network*: contains common utilities for managing the network. The two most important abstractions are the *Message*, which is an abstract interface that all message classes must implement, and the *SocketWrapper*, which wraps an *inet socket* and provide functions for easily sending and receiving *messages*.
- *security*: contains the OpenSSL implementation of the cryptographic functions and the *SecureSocketWrapper* class, providing the same interface as the *SocketWrapper* but using encrypted messages.
- *utils*: miscellaneous classes providing easy-to-use utility functions for, e.g., securely reading from *stdin* (preventing buffer overflows), reading/writing to a bounded buffer (once again to prevent possible buffer overflows), dumping a buffer to stdout for debug purposes. It also provides an implementation of a thread-safe message-passing queue which is used by the server to distribute tasks among worker threads.
- *server*: implementation of the server executable, along with some private utility classes: *User*, which represent a user as a finite state machine, and *UserList*, which provides a convenient thread-safe collection of users which can be retrieved (through an hash map) by either their username or the socket file descriptor they are bound to.
- *client*: implementation of the client executable, along with some private utility classes: *Connect4*, which provides the game implementation, *Server*, which proxies the client communication to the server by providing a more simple interface without having the client implement the networking logic. Furthermore, the *multi_player* and *single_player* modules provide the main functions to handle the game in the different circumstances.

In the following, the main workflow of the client and server is first illustrated and then a brief explanation of some of the main utility components is provided. For a more in-depth description, please refer to the auto-generated Doxygen documentation or the source code itself.

3.3 Server

The server is implemented as a main thread, accepting communications from clients and reading messages from sockets, and multiple worker threads, that handle the messages (including decryption). We will first describe the main thread and then the worker threads.

3.3.1 Main thread

At start-up, the main thread loads its own certificate and key, the CA's certificate and all client certificates. If this operation succeeds, the server creates a new listening socket on the given port, spawns the worker threads and waits for new connections using the *select* system call. On a new connection request, the server accepts the connection and adds the socket to the list of sockets that checked by the *select*. When a client sends data to the server, the main thread is woken and the corresponding message will be en-queued in the *MessageQueue*, waiting for a worker thread to elaborate it.

3.3.2 Worker thread

The worker thread waits on a *condition* variable waiting for new messages to be available in the *MessageQueue*. Once a new message is available, the thread starts elaborating it. First of all, it retrieves the *User* instance[2] that message refers to and locks it with a *mutex* to prevent other threads to change the *User* state concurrently. Depending on the state of the user, messages are handled accordingly:

- *Just connected*: this is the starting state of a *User*. In this state the following messages are handled:
  - *Certificate request*: the server certificate is forwarded to the *User* by means of the *certificate* message.
  - *Client hello*: the client has begun an *handshake* with the server. The server replies with a *server hello* message.
  - *Client verify*: the client is terminating the *handshake*. If verification succeeds, the user is considered authenticated and is moved to the *Securely connected* state.
- *Securely connected*: this is the first state a user is brought upon authentication. In this state only one message is valid:
  - *Register*: the *User* communicates the server that she is ready for receiving challenges and is moved to the *registered* state without further action. Note that this message may not be necessary, however it has been kept for compatibility with the non-secure version of the game.
- *Available*: in this state the *user* is fully operational: he may be returned in the user list to other users, he may be challenged by other users and challenge other users. In this state the following messages are valid:
  - *User list request*: the *User* asks the server to tell him which users are online. The server replies with the *user list*.
  - *Challenge*: the *User* tells the server he wants to challenge an *opponent*. The server moves both *Users* to the *Challenged* state and forwards the challenge to the *opponent* with a *challenge forward* message.

[2] at this time the user may not be logged in yet or he could not even have completed the handshake. In these cases, the *User* instance just represents the socket and not yet the user.

- *Challenged*: in this state the *user* cannot receive any further challenge and is either waiting for a reply from the *opponent* or deciding whether to accept it. In this state only the following message is handled.
  - *Challenge reply*: the *User* tells the server that she accepts or refuses the challenge. In the first case, the user also indicates on which port he will listen for a connection from the opponent. If the challenge is accepted, both users are sent a *game start* message indicating the certificate of the *opponent* and, in the case of the user that sent the challenge in the first place, also the IP address and port which the opponent is listening for a connection. Finally, both users are moved to the *playing* state. In the case the challenge is refused, both users are sent a *challenge cancel* message and are moved back to the *available* state.
- *Playing*: in this state the *user* cannot receive any challenge since it's playing with another user. In this state only one message can be received:
  - *Game end*: the user signals the server that he is now available and is moved to the *available* state.
- *Disconnected*: the user connection dropped. The *User* instance is waiting to be destroyed and all pending messages from the users are ignored.

Furthermore, note that:

- In case an unexpected message is received, the message is simply ignored.
- In case a challenge is not possible, either because the user or the opponent is in a state where he cannot receive challenges or because the *user*/*opponent* disconnected in the meanwhile, a *game cancel* message is sent to both users (or just one).

Once the message handling is completed, the message is deleted, the *User mutex* unlocked and the *MessageQueue* is polled again. However, if the message handling completed with problems that indicate that the user is no longer connected, the *User* is also moved to the *disconnected* state before releasing the lock. It is important to note that the worker thread must not delete the *User* instance because another thread may hold a reference to it. For this reason, the *UserList* keeps track of outstanding references to every user and is in charge of deleting disconnected *Users* that are no longer referenced.

3.4 Client

At start-up, the client loads his certificate and key (inserting a password if necessary) and then waits for the user to decide a playing mode: server, peer (waiting for a connection), peer (connecting to the other peer) and offline (used for debug purposes, the opponent has no logic).

3.4.1 Server mode

Once the user selects this mode, he also indicates the address and port of the server. The client then sends a *certificate request* to the server and waits for the corresponding *certificate* reply. It then initiates the *handshake* with the server, sending the *client hello* message. Once the corresponding *server hello* message is received, the signature is checked against the certificate and it sends back the *client verify* message to complete the authentication. Now that the *handshake* is completed, the client sends the *register* message and waits for either user input from stdin or server messages from the network using the *select* system call. Let's first describe the possible user commands that can be given from stdin:

- *list*: the client sends a *user list request* message, waits for the corresponding *user list reply* message from the server containing the list of users and then displays the list to the user.

- *challenge*: the client sends a *challenge* message to the server indicating that he wants to challenge an *opponent* and waits for either a *game start* or a *game cancel* message. In the first case, the client enters the *connect to peer* peer-to-peer mode. In the second case, the client restarts polling with the *select* system call.

In addition, the following messages may be received from the server:

- *challenge forward*: the server is forwarding a challenge from another user that the user can either accept or refuse. If the challenge is accepted, the client sends the server the port he will wait for the peer connection in the *challenge response* message and enters the *wait for peer* peer-to-peer mode. Otherwise, it will just indicate that the challenge is to be refused.

Note that when the client leaves the *server mode* to enter one of the two *peer-to-peer modes* the connection with the server is kept alive and will be reused when the game finishes to send the *game end* message. The client is then brought back to the *server mode* where the user can list users, challenge other users and receive challenges.

This mode is implemented in the *client/server_lobby.cpp* file.

3.4.2 Peer-to-Peer mode

The peer-to-peer mode is divided into two roles: server, which waits for a peer connection, and client, which connects to a peer. Once a secure connection is established between the peers, the client enters the common *playing* mode.

This mode is implemented in the *client/multi_player.cpp* file.

Server: wait for peer Once entering this mode, the client opens a new listening socket waiting for a connection from the peer. Once the peer connects, the *handshake* is performed as described before and the client enters the *playing* mode as the first player to place a move.

Client: connect to peer Once entering this mode, the client tries to connect to the peer (it will retry every second for up to 10 trials). Once the peer connects, the *handshake* is performed as described before and the client sends a *start game peer* message to start the game. The client then enters the *playing* mode as the second player to place a move.

Playing For each turn until the end of the game, the client performs the following operations depending on whose turn it is:

- if it is its turn, it will ask the user to choose a column. If the column is valid, the choice is sent to the peer with a *move* message. Once the message is sent, the move is placed on the board and the turn is switched.
- if it is the turn of the opponent, it will wait for a *move* message. Once the message is received, the move is placed on the board and the turn is switched.

3.5 Main common utility component

Args The *Args* class is used to safely parse user input from the standard input. By securely, we mean that the user input is read in a *C++ string* thus preventing possible buffer overflows. The class then returns an "argc/argv"-like format.

buffer_io The *buffer_io* module provides functions to read/write from a bounded buffer. Before reading/writing a possible buffer overflow is checked and an error return value is returned.

crypto and crypto_utils The *crypto* module provides the implementation of the cryptographic functions using OpenSSL, while the latter provides some utility functions related to OpenSSL objects (e.g. serializing a public key or a certificate).

messages The *messages* module provides the implementation of all messages. All messages are a class that implements the *Message* interface which provides a common method to read or write from/to a buffer. By exploiting polymorphism, it is then easy to read a message from a buffer and to cast it to the correct class, through the *readMessage* function.

SocketWrapper The *SocketWrapper* provides convenient methods for sending and receiving *messages*. Every message is encoded in the same way: 2 bytes for the total length of the message, one byte for the message type (used to select the class to use to read from it) and the body of the message. When sending a message, the message is written to the output buffer using the *write* method of the message, then the length is added in the heading and the message is sent through the *send* system call. When reading a message, first the first two bytes are read to detect the message length using the *receive* system call with the *WAIT_ALL* flag, then the remainder of the message is read to the input buffer using the same system call. The buffer is then passed to the *readMessage* function that will return a pointer to *Message*, which has been cast to the correct type.

SecureSocketWrapper The *SocketWrapper* provides the same abstraction of *SocketWrapper* but with the additional layer of security. Only a selected range of messages is allowed to be sent or received in plain text (i.e. *certificate request*, *certificate*, *client hello*, *server hello*, *client verify*). All other messages require peer authentication through the handshake protocol, which is managed by the *SecureSocketWrapper* itself which also manages all secure connection state variables (keys, IVs, sequence numbers). To send any other message, encryption is required. In order to perform encryption, the message is first written to a plain-text buffer, as is done in the *SocketWrapper* sending method, then the message is encrypted with AES-256 in GCM mode using the total message length and the message type as AAD. The resulting cipher-text and verification tag are then used to build a *secure message* which is finally sent through the socket using a *SocketWrapper*. Once a message is successfully sent, the write sequence number is increased. To receive messages, the converse process is done: the *secure message* is received from the *SocketWrapper*, cipher-text and tag are extracted, cipher-text is decrypted in a decryption buffer which is passed to the *readMessage* function that will return a pointer to *Message*, which has been cast to the correct type. Tag is verified in the decryption function of AES-256-GCM and, in case of verification failure, an error value is returned. Once a message is successfully read, the write sequence number is increased.

4 Messages

In this section, the format of all exchanged messages is reported. Each message is represented as a table where one cell corresponds to one byte. Fields are aligned in order to improve readability, thus empty spaces do not represent real empty parts of the message. Real messages are always "contiguous".

4.1 SECURE MESSAGE

Carries an encrypted message.

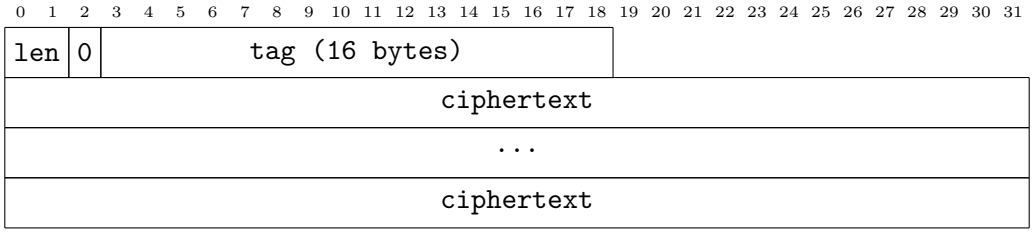

Figure 2: Secure message format

- len: length of the message
- 0: type of the message (SECURE_MESSAGE)
- ciphertext: encrypted message
- tag: aes_gcm verification tag

4.2 CLIENT HELLO

Sent by a client when a new connection has to be started.

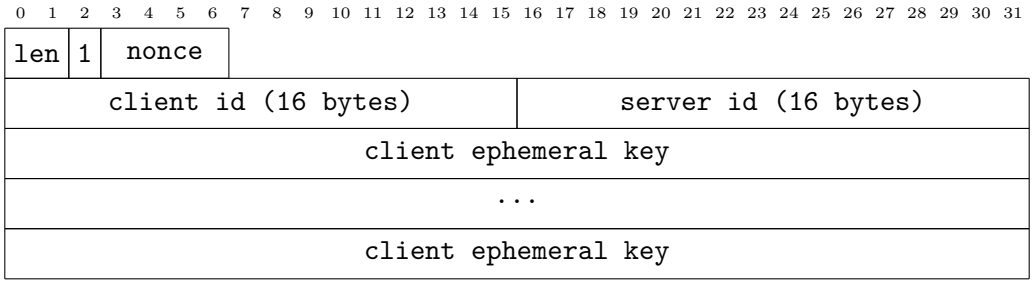

Figure 3: Client hello message format

- len: length of the message
- 1: type of the message (CLIENT_HELLO)
- nonce: random integer generated by the client
- client id: identifier of the client, i.e. his username as a string padded with 0s (padding is not required here but is kept for coherence with other messages).
- server id: identifier of the server. The identifier of the game server is the string `server`.
- eph key: DH ephemeral key of the client

4.3 SERVER HELLO

Server reply to a SERVER HELLO message.

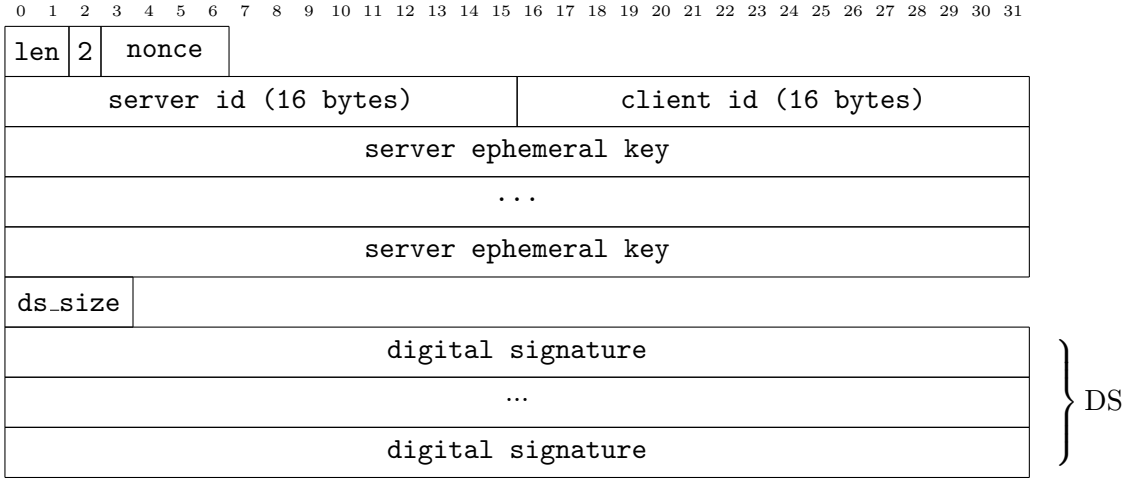

Figure 4: Server hello message format

- len: length of the message
- 2: type of the message (SERVER_HELLO)
- nonce: random integer generated by the server
- server id: identifier of the server
- client id: identifier of the client
- digital signature: digital signature of the server

4.4 CLIENT VERIFY

Client sends its digital signature.

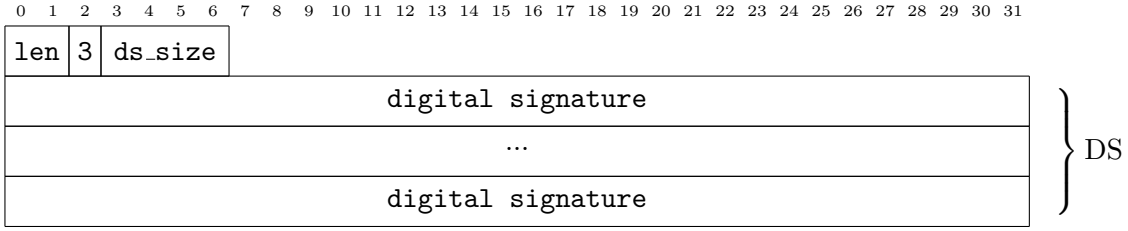

Figure 5: Client verify message format

- len: length of the message
- 3: type of the message (CLIENT_VERIFY)
- digital signature: digital signature of the client

4.5 START GAME PEER

Message sent when an user wants to start a game with another peer.

- len: length of the message
- 4: type of the message (START_GAME_PEER)

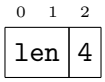

Figure 6: Start game peer message format

4.6 MOVE

Message containing the next move of the user

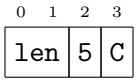

Figure 7: Move message format

- len: length of the message
- 4: type of the message (MOVE)
- C: chosen column where to insert the token

4.7 REGISTER

Registers user to the server.

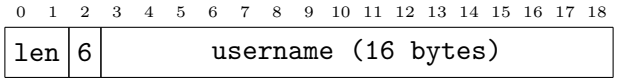

Figure 8: Register message format

- len: length of the message
- 6: type of the message (REGISTER)
- username: username of the user. It is not necessary to specify the username again since user connections are already authenticated but it's kept for compatibility with the non-secure implementation.

4.8 CHALLENGE

Sends a challenge to a user.

- len: length of the message
- 7: type of the message (CHALLENGE)
- username: adversary to challenge. Here padding is useful to prevent leaking any kind of information about the challenged user.

4.9 GAME END

Signals end of the match.

- len: length of the message
- 8: type of the message (GAME_END)

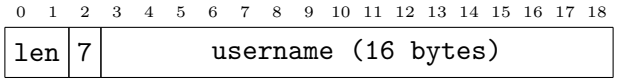

Figure 9: Challenge message format

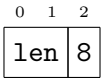

Figure 10: Game end message format

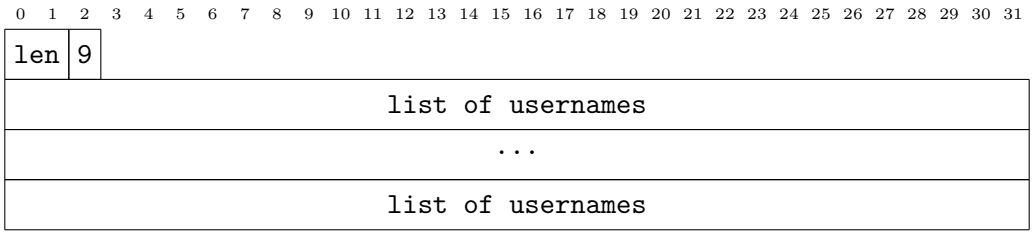

Figure 11: Users list message format

4.10 USERS LIST

Transports the list of all the active users.

- len: length of the message
- 9: type of the message (USERS_LIST)
- list of usernames: comma separated list of usernames padded at multiple of 16 bytes to prevent an eavesdropper to guess online users by measuring the message length.

4.11 USERS LIST REQUEST

Requests the list of all the active users

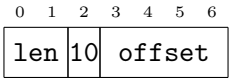

Figure 12: Users list request message format

- len: length of the message
- 10: type of the message (USERS_LIST)
- offset: offset for the list of users we want to receive (where to start from)

4.12 CHALLENGE FORWARD

Message forwarded from the server to the peer to signal an incoming challenge.

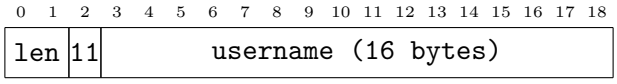

Figure 13: Challenge forward message format

- len: length of the message
- 11: type of the message (CHALLENGE_FORWARD)
- username: username of the challenger

4.13 CHALLENGE RESPONSE

Message carrying the response to a challenge.

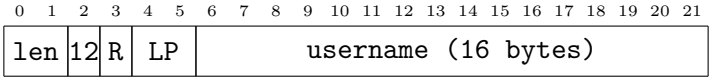

Figure 14: Challenge response message format

- len: length of the message
- 12: type of the message (CHALLENGE_RESPONSE)
- R (response): response accepted or refused
- LP (listen port): if accepted, the port on which the user is listening
- username: username of the challenger

4.14 GAME CANCEL

Message through which the server forwards a challenge rejectal or another event that caused the game to be canceled.

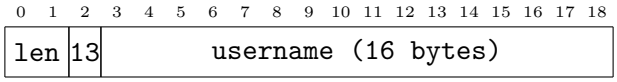

Figure 15: Game cancel message format

- len: length of the message
- 13: type of the message (GAME_CANCEL)
- username: username of the challenger

4.15 GAME START

Message through which the server makes a new game start between two peers. Sent from the server to the peers.

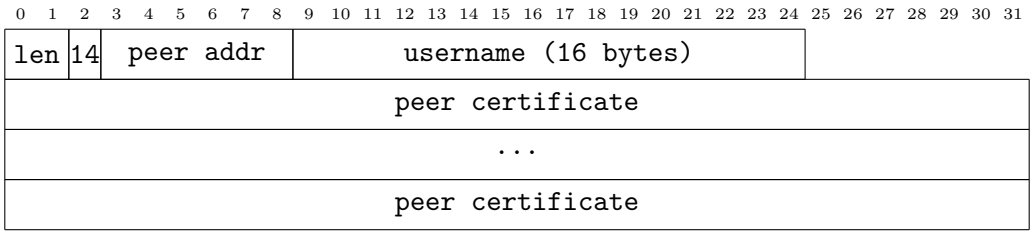

Figure 16: Game start message format

- len: length of the message
- 14: type of the message (GAME_START)
- peer addr: listening address and port (sockaddr_in) of the peer
- username: username of the peer
- peer certificate: certificate of the peer

4.16 CERTIFICATE REQUEST

Used by a client to request the server certificate.

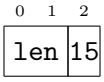

Figure 17: Certificate request message format

- len: length of the message
- 15: type of the message (CERTIFICATE_REQUEST)

4.17 CERTIFICATE

Server response to a `CERTIFICATE REQUEST` message.

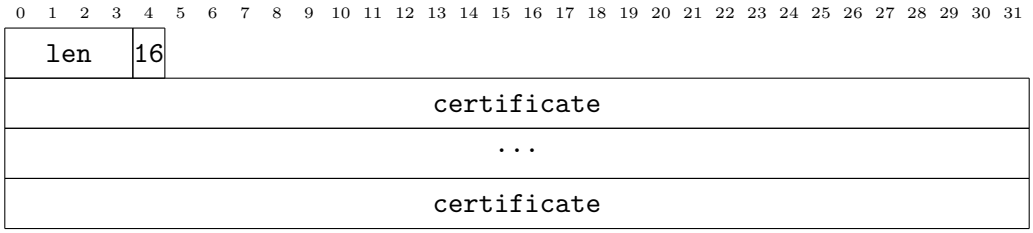

Figure 18: Certificate response message format

- len: length of the message
- 16: type of the message (CERTIFICATE)
- certificate: server certificate