Federico PACINI

Marco PETTORALI

(alias GrazianoEncryptionStandard)

Summary

Design and implementation of a secure version of the popular board game “Four-in-a-row”

secure Four-in-a-row

Foundations of Cybersecurity project, A.Y. 2019-20

Index

[Project specifications 3](#_Toc44406145)

[Design choices 3](#_Toc44406146)

[Cryptographic issues 3](#_Toc44406147)

[Server and multithreading 3](#_Toc44406148)

[Blocking and non-blocking sockets 4](#_Toc44406149)

[Errors and cheats handling 4](#_Toc44406150)

[Protocol and exchanged messages 5](#_Toc44406151)

[Client-server 5](#_Toc44406152)

[Authentication phase 5](#_Toc44406153)

[List retrieval 7](#_Toc44406154)

[Request to play 8](#_Toc44406155)

[Start game 10](#_Toc44406156)

[End game 11](#_Toc44406157)

[Close connection 12](#_Toc44406158)

[P2P 13](#_Toc44406159)

[Authentication phase 13](#_Toc44406160)

[In-game messages 15](#_Toc44406161)

[BAN logic proof of client-server authentication 16](#_Toc44406162)

[Real protocol 16](#_Toc44406163)

[Assumptions 16](#_Toc44406164)

[Objectives 16](#_Toc44406165)

[Idealized protocol 16](#_Toc44406166)

[Proof 17](#_Toc44406167)

[M2: 17](#_Toc44406168)

[M3: 17](#_Toc44406169)

[M4: 17](#_Toc44406170)

[BAN logic proof of P2P authentication 19](#_Toc44406171)

[Real protocol 19](#_Toc44406172)

[Assumptions 19](#_Toc44406173)

[Objectives 19](#_Toc44406174)

[Idealized protocol 20](#_Toc44406175)

[Proof 20](#_Toc44406176)

[M2: 20](#_Toc44406177)

[M3: 20](#_Toc44406178)

[M4: 21](#_Toc44406179)

# Project specifications

The goal of the project is to design and implement a secure online version of the popular board game “Four-in-a-row”. The resulting protocol must not be vulnerable to the most common attacks, such as replay attack or man-in-the-middle attack.

The application follows a mixed client-server and P2P model: clients (the players) have a local private key protected with a password, and are assumed to be already registered in the remote server. The server collects the players’ public key, authenticates them and establish a shared session key with each of them.

When the authentication phase is completed, a user can decide to retrieve the list of all the clients logged in at the moment, and can decide to ask another player to play together. If it accepts, a P2P connection between the twos is initialized, and a phase of mutual authentication is started and a session key is established. If this phase terminates correctly, then the two players can play the game.

When the game ends, the connection between the two players is released, the session keys are destroyed and the two players can decide to quit the service or play another match with other opponents.

# Design choices

## Cryptographic issues

Clients are assumed to choose good cryptographically secure shared keys (using the OpenSSL function RAND\_poll()); in the protocol, in fact, the creation of the session key is left to the clients in order to decentralize the load from the server.

The algorithm used for digital signatures is RSA, since is the only one supported by ‘SimpleAuthority’, the software used to manage certificates in the system.

The algorithm used for symmetric encryption is AES in GCM mode: this allow an authenticated encryption of messages.

## Server and multithreading

The server of the game is multithreaded: this obviously allows more players to be connected to the system at the same time. This means that the server will have a ‘listener’ thread that accepts incoming connections, and assign each of them to a ‘handler’ thread. The name of the handler thread is the same of the username of the client that it’s handling.

The default address of the server is ‘127.0.0.1’, since the trials that are needed for testing the operation are made in local.

After the authentication phase, clients have to tell the server on which port they will be reachable for P2P communications; this is due to the fact that all the test of the application are made in local, and it would be impossible to open sockets on the same port for each one of the client instances that we want to run.

## Blocking and non-blocking sockets

Non-blocking sockets are used by handler threads after its corresponding client has authenticated to periodically listen for both client commands and requests for playing from other handler threads in the server. In fact, if we used a single blocking socket listening to client’s commands, the socket would remain blocked to wait for an input, and could not be waken up when an outer request for playing arrives.

A solution may be using two threads for each client connected in the system, but that would have cons of overhead in terms of memory: too much for silly task such as controlling periodically outer requests.

Instead, our solution is to use a non-blocking socket: periodically, the handler thread checks for incoming commands from the client, but remains awake for handling outer requests for playing with the handled client. This solution brings in a little overhead due to the periodical check for commands, but it’s not so relevant, since the set period is 2 secs and our application has not strict time requirements.

## Errors and cheats handling

Each error or exception in the system is handled to shut down the communication between the clients or the clients and the server. This is necessary to react to any manipulation that an adversary may be able to perform on the exchanged messages, and makes the system secure and reliable.

If a user tries to cheat by inserting a bad move in its local play, the system detects it and shows a message on screen, and then the client can try again. No bad moves are allowed in the system, and this mechanism prevents it.

# Protocol and exchanged messages

## Client-server

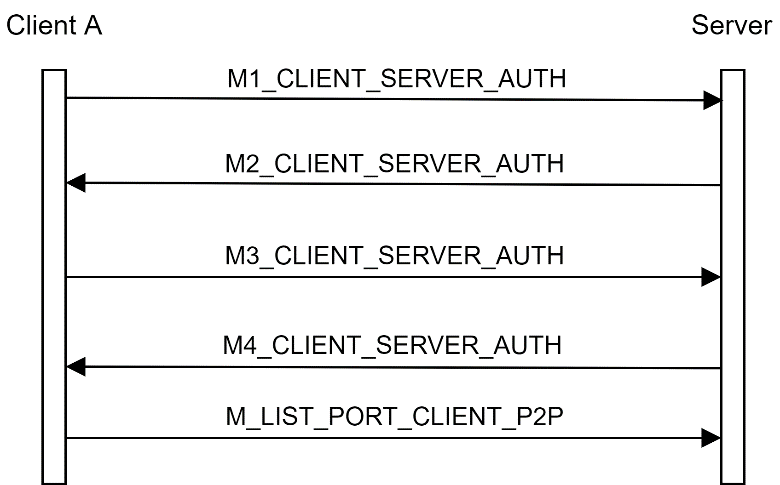
### Authentication phase

#### DESCRIPTION

The authentication phase is divided into 5 stages: with the first nonce (necessary to cover the exchange of the server’s certificate from a replay attack) and the subsequent exchange of challenges, client and server can realize they’re not experiencing a man in the middle nor a replay attack.

The client then choses a random shared key and sends it to the server, that replies using that key and encrypting the server’s challenge to prove to the client that it has correctly received the session key.

#### SCHEME



#### M1\_CLIENT\_SERVER\_AUTH

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 B | 4 B | 16 B | 7 B | 32 B |
| OPCODE | PAYLOAD\_LEN | ID\_CLIENT | ID\_SERVER | NONCE\_A |
| Opcode | Payload length | Nickname of the client | Nickname of the server | Nonce created by the client |

#### M2\_CLIENT\_SERVER\_AUTH

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 B | 4 B | 4 B | S\_CERT\_LEN B | (7 + 16 + 32 + 32 B = 87 B)plaintext |
| OPCODE | PAYLOAD\_LEN | S\_CERT\_LEN | S\_CERT | EPUBKA(ID\_SERVER, ID\_CLIENT, NONCE\_A, CHALLENGE\_A) |
| Opcode | Payload length | Length of the server’s certificate | Server’s certificate | Response to nonce, challenge created by the server |

#### M3\_CLIENT\_SERVER\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 7 + 32 + 32 + 16 B = 103 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EPUBK\_SERVER(ID\_CLIENT, ID\_SERVER, CHALLENGE\_A, CHALLENGE\_S, KAS) |
| Opcode | Payload length | Response to challenge, challenge created by the client, creation of a shared session key |

#### M4\_CLIENT\_SERVER\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (7 + 16 + 32 B = 55 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKAS(ID\_SERVER, ID\_CLIENT, CHALLENGE\_S) |
| Opcode | Payload length | Response to challenge |

#### M\_LISTEN\_PORT\_CLIENT\_P2P

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 7 + 4 B = 27 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_CLIENT, ID\_SERVER, PORT) |
| Opcode | Payload length | Communication of the port on which this client will listen for P2P connections |

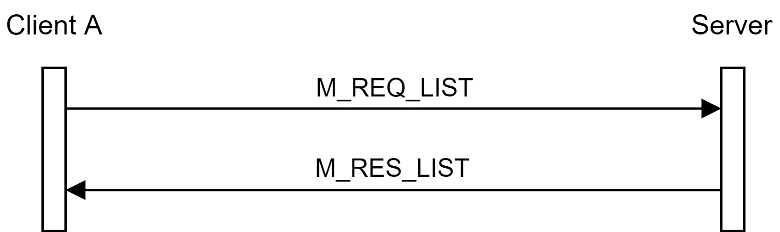
NOTE: This message is important only in case of execution of more instances of the program on the same machine, otherwise a common port could be assigned to P2P connections

### List retrieval

#### DESCRIPTION

The client asks for the list of all the logged in users at the moment, using a nonce to avoid that a replay attack (using a not updated list) is performed. The server responds with the list of users using encryption to avoid any manipulations.

#### SCHEME



#### M\_REQ\_LIST

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKAS(NONCE\_CLIENT) |
| Opcode | Payload length | Nonce created by the client |

#### M\_RES\_LIST

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 + 4 + LIST\_LEN B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKAS(NONCE\_CLIENT, LIST\_LEN, LIST) |
| Opcode | Payload length | Communication of the list of logged in clients |

### Request to play

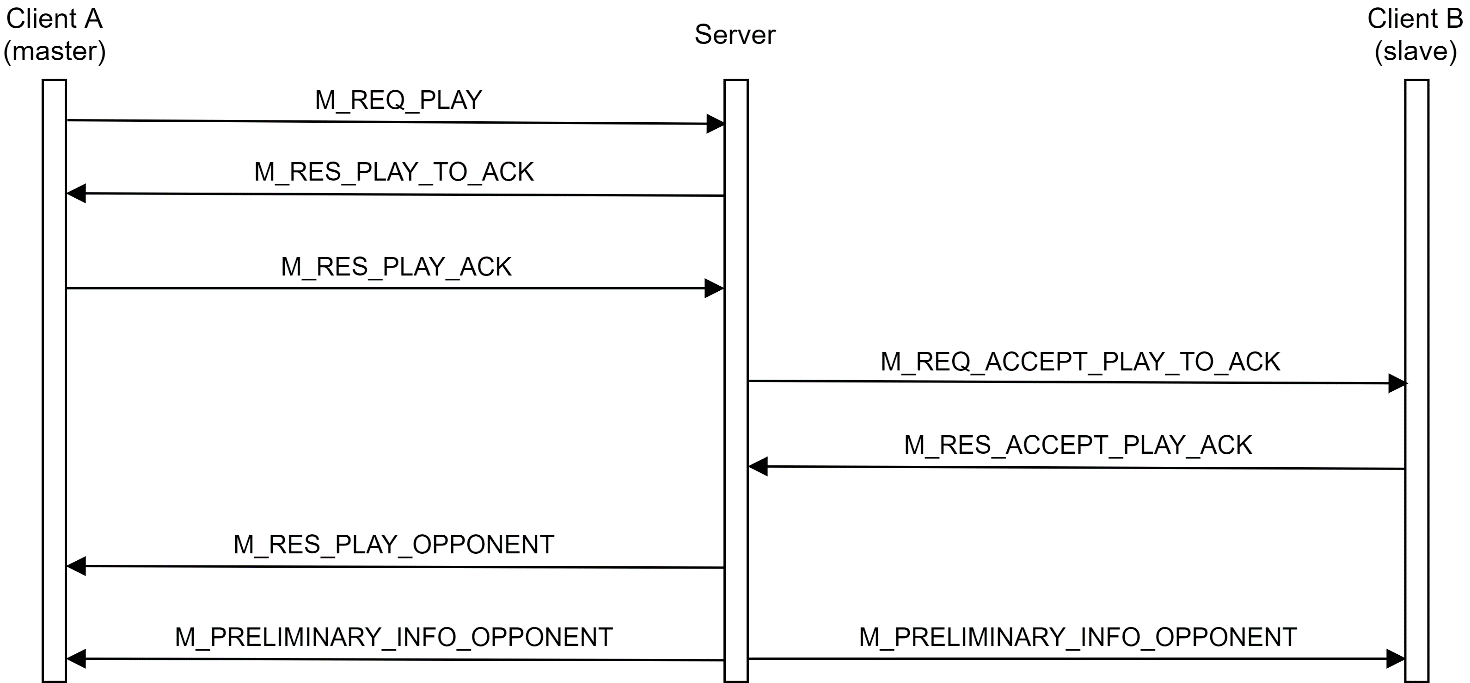
#### DESCRIPTION

After the authentication, the client can send a request to the server to play with another idle client. The request is forwarded to the latter, and its response is communicated to the client. If the response is positive, the server sends both clients the respective opponent’s public key.

Messages are encrypted and protected by replay attacks with a challenge-response mechanism that uses a client nonce and a server one.

From now on, the client that initialized the protocol will be called “master” client, while the other one will be called “slave” client.

#### SCHEME



#### M\_REQ\_PLAY

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 32 B = 48 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_OPPONENT, NONCE\_CLIENT) |
| Opcode | Payload length | Request to play with the specified opponent, nonce created by the client |

#### M\_RES\_PLAY\_TO\_ACK

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 32 + 32 B = 80 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_OPPONENT, NONCE\_CLIENT, NONCE\_SERVER) |
| Opcode | Payload length | Response to the client’s nonce, nonce created by the server |

#### M\_RES\_PLAY\_ACK

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 32 B = 48 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_OPPONENT, NONCE\_SERVER) |
| Opcode | Payload length | Response to the server’s nonce |

#### M\_REQ\_ACCEPT\_PLAY\_TO\_ACK

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 32 B = 48 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_OPPONENT, NONCE\_SERVER) |
| Opcode | Payload length | The server asks a client if it wants to play with the specified opponent, nonce created by the server |

#### M\_RES\_ACCEPT\_PLAY\_ACK

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (1 + 16 + 32 B = 49 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(RESPONSE\_1BYTE, ID\_OPPONENT, NONCE\_SERVER) |
| Opcode | Payload length | Response of the slave client, response to the server’s nonce |

#### M\_ RES\_PLAY\_OPPONENT

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (1 + 4 + 16 + 32 = 53 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(RESPONSE\_1BYTE, OPPONENT\_PORT, ID\_OPPONENT, NONCE\_CLIENT) |
| Opcode | Payload length | Response forwarded to the master client, together with its listening port, response to the master client’s nonce |

#### M\_PRELIMINARY\_INFO\_OPPONENT

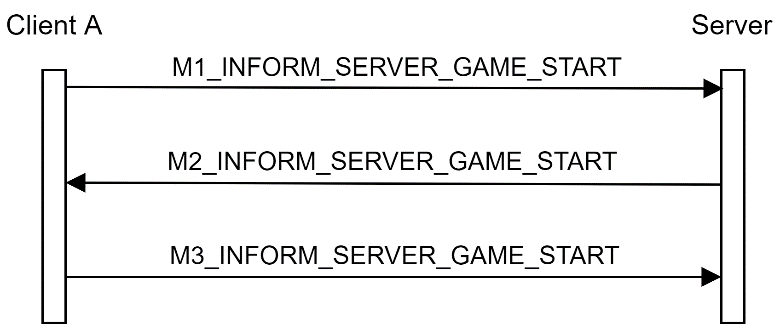
|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 16 + PUBKEY\_LEN B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_LOCAL, ID\_OPPONENT, PUBKEY\_LEN, PUBKEY\_OPPONENT) |
| Opcode | Payload length | The server informs each player of the opponent’s information, such as their nicknames and public key |

### Start game

#### DESCRIPTION

Before the two clients perform the authentication phase, they have to inform the server that they’re not reachable anymore for playing with other players until the end of the game. This is done by exchanging the INFORM\_SERVER\_GAME\_START messages, that include a challenge-response mechanism using nonces to avoid replay attacks.

#### SCHEME



#### M1\_INFORM\_SERVER\_GAME\_START

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(NONCE\_CLIENT) |
| Opcode | Payload length | The client informs the server that it will not be reachable anymore to play, nonce created by the client |

#### M2\_INFORM\_SERVER\_GAME\_START

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 + 32 B = 64 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(NONCE\_CLIENT NONCE\_SERVER) |
| Opcode | Payload length | Response to the client’s nonce, nonce created by the server |

#### M3\_INFORM\_SERVER\_GAME\_START

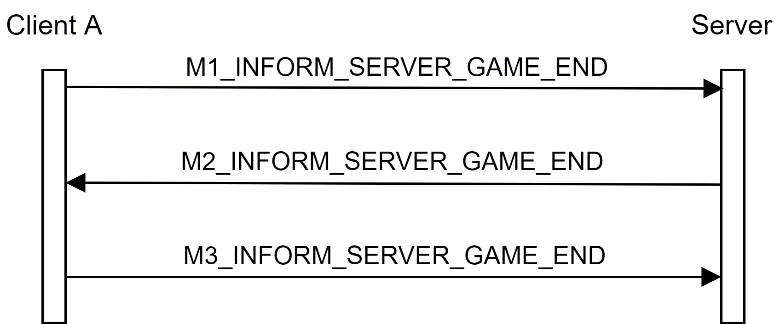
|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(NONCE\_SERVER) |
| Opcode | Payload length | Response to the server’s client |

### End game

#### DESCRIPTION

When the game ends, the clients tell the server that they’re now reachable for playing with other opponents. This is done by exchanging the INFORM\_SERVER\_GAME\_END messages, that include a challenge-response mechanism using nonces to avoid replay attacks.

#### SCHEME



#### M1\_INFORM\_SERVER\_GAME\_END

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(NONCE\_CLIENT) |
| Opcode | Payload length | The client informs the server that it’s now reachable for playing, nonce created by the client |

#### M2\_INFORM\_SERVER\_GAME\_END

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 + 32 B = 64 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(NONCE\_CLIENT NONCE\_SERVER) |
| Opcode | Payload length | Response to the client’s nonce, nonce created by the server |

#### M3\_INFORM\_SERVER\_GAME\_END

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (32 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(NONCE\_SERVER) |
| Opcode | Payload length | Response to the server’s client |

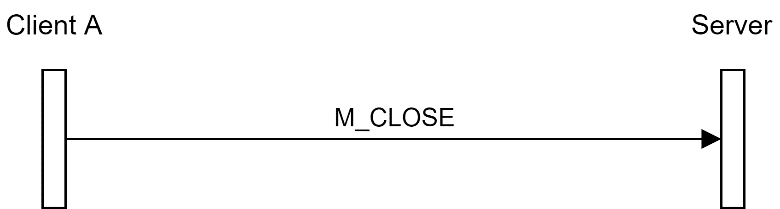
### Close connection

#### DESCRIPTION

When the user decides to disconnect from the server, it first tell the server to close the socket on the thread that communicates with it. To make this disconnection not vulnerable to replay attacks, the client ciphers the session key, so that the server can understand that it’s the very client that wants to shut down the communication, and not an adversary.

It’s not needed to protect this message from record and replay attacks with other mechanisms, because the session key is assumed to be known to the client and the server only, and since it’s different in every session the M\_CLOSE message cannot be recorded and replayed.

#### SCHEME



#### M3\_INFORM\_SERVER\_GAME\_END

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKas(Kas) |
| Opcode | Payload length | Session key encrypted with the very session key |

## P2P

### Authentication phase

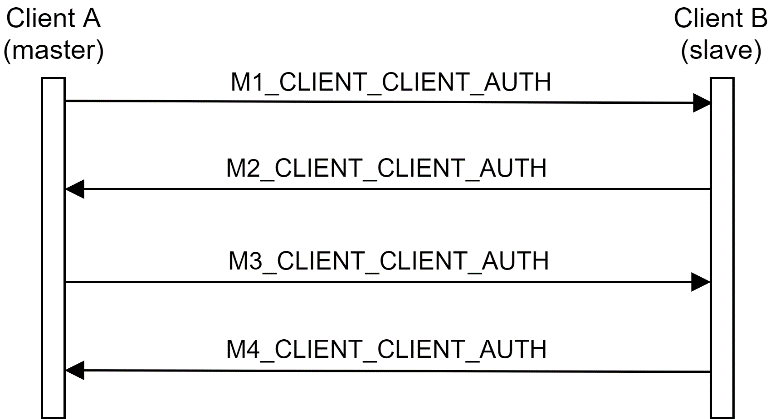
#### DESCRIPTION

When the two clients have received the opponent’s public key from the server, the authentication protocol between them can start.

This protocol is very similar to the one seen for client-server authentication, with the only difference that there’s no exchange of certificate, since both the clients trust the server and the keys that it provided to them.

The names “local” and “opponent” are referred to the part that sends the message, and can refer to both master and slave clients, according to their position in the protocol (as sender or receiver of the message).

#### SCHEME



#### M1\_CLIENT\_CLIENT\_AUTH

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 B | 4 B | 16 B | 16 B | 32 B |
| OPCODE | PAYLOAD\_LEN | ID\_LOCAL | ID\_OPPONENT | NONCE\_LOCAL |
| Opcode | Payload length | Nickname of the master client | Nickname of the slave client | Nonce created by the master client |

#### M2\_CLIENT\_CLIENT\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 16 + 32 + 32 B = 96 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EPUBK\_OPPONENT(ID\_LOCAL, ID\_OPPONENT, NONCE\_CLIENT, CHALLENGE\_OPPONENT) |
| Opcode | Payload length | Response to nonce, challenge created by the slave client for the master |

#### M3\_CLIENT\_CLIENT\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 16 + 32 + 32 = 96 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EPUBK\_OPPONENT(ID\_LOCAL, ID\_OPPONENT, CHALLENGE\_LOCAL, CHALLENGE\_OPPONENT, KAB) |
| Opcode | Payload length | Response to slave’s challenge (called CHALLENGE\_LOCAL because it’s destined to the local player), challenge created by the master client destined to the opponent slave client, creation of a shared session key |

#### M4\_CLIENT\_CLIENT\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 16 + 32 B = 64 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKAB(ID\_LOCAL, ID\_OPPONENT, CHALLENGE\_LOCAL) |
| Opcode | Payload length | Response to the master’s challenge (called CHALLENGE\_LOCAL because it’s destined to the local player) |

### In-game messages

#### DESCRIPTION

The core of the game is represented by the MOVE message, containing the column on which a checker must be inserted, and a counter on 4 bytes to prevent the replay of the moves, and encryption to prevent any manipulations. Note that the counter can count up to 232 = 4 G, while the game grid can only allow at maximum 6 x 7 = 42 checkers, and so the maximum value for the move counter is 42: there is no risk of integer overflow.

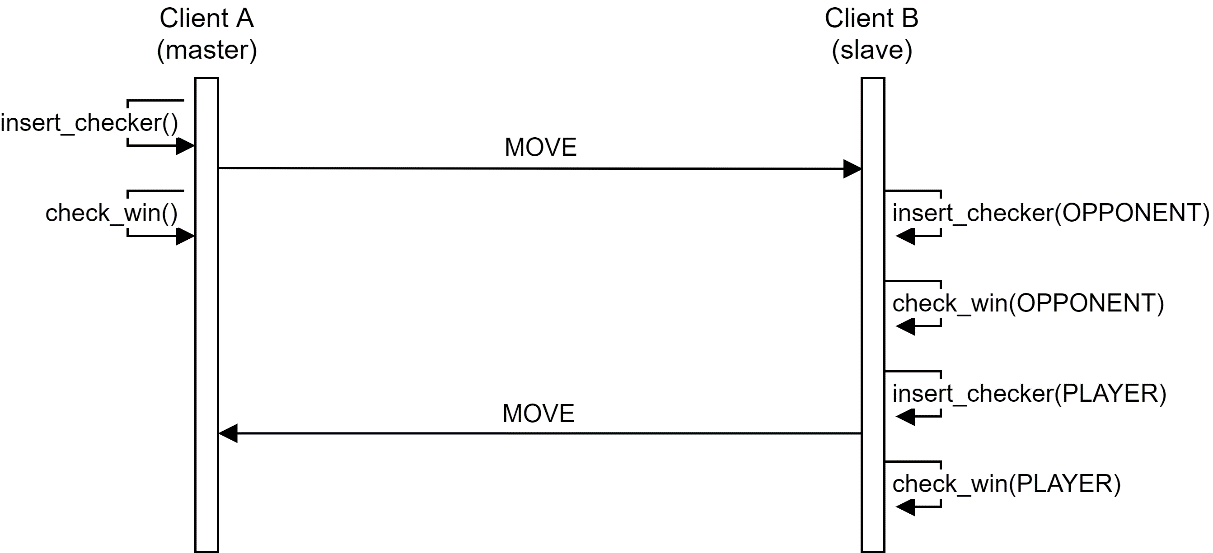
The following scheme represents not only the exchange of messages in the network in the game phase, but also the operations done by the players.

Both the players start with an empty grid; the master is the first one to start the game.

A client inserts a checker in a valid column. The move is sent to the opponent, the game grid is updated, and then the client checks if it has won the match; if it wins, a winning message appears on screen, and the player exits from the current game.

The opponent receives the message, inserts the pawn of the adversary in the corresponding column of its own game grid, and then checks for an opponent’s win. If the opponent wins, a “you lose” message appears on screen, and the player exits from the current game. Otherwise, the parts are changed, and the opponent is allowed to insert another checker, check for its win and so on.

#### SCHEME



#### MOVE

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | (16 + 16 + 4 + 1 B)plaintext |
| OPCODE | PAYLOAD\_LEN | EKAB(ID\_LOCAL, ID\_OPPONENT, COUNT, COLUMN) |
| Opcode | Payload length | Counter of the current move and column in which the checker was inserted |

# BAN logic proof of client-server authentication

## Real protocol

## Assumptions

|  |  |  |
| --- | --- | --- |
| Keys | Freshness | Trust |
|  |  |  |

## Objectives

## Idealized protocol

## Proof

### M2:

Applying the 2nd postulate:

Applying the 3rd postulate:

### M3:

Applying the 2nd postulate:

We’ve proven objective 3:

Applying the 3rd postulate:

We’ve proven objectives 1 and 2:

### M4:

Applying the 1st postulate:

Applying the 2nd postulate:

Applying the 3rd postulate:

We’ve proven objective 4:

# BAN logic proof of P2P authentication

## Real protocol

## Assumptions

|  |  |  |
| --- | --- | --- |
| Keys | Freshness | Trust |
|  |  |  |

## Objectives

## Idealized protocol

## Proof

### M2:

Applying the 2nd postulate:

Applying the 3rd postulate:

### M3:

Applying the 2nd postulate:

We’ve proven objective 3:

Applying the 3rd postulate:

We’ve proven objectives 1 and 2:

### M4:

Applying the 1st postulate:

Applying the 2nd postulate:

Applying the 3rd postulate:

We’ve proven objective 4: