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](#_Toc44589268)

[Design choices 3](#_Toc44589269)

[Cryptographic issues 3](#_Toc44589270)

[Server and multithreading 3](#_Toc44589271)

[Blocking and non-blocking sockets 4](#_Toc44589272)

[Errors and cheats handling 4](#_Toc44589273)

[Protocol and exchanged messages 5](#_Toc44589274)

[Preliminary information 5](#_Toc44589275)

[Client-server 6](#_Toc44589276)

[Authentication phase 6](#_Toc44589277)

[List retrieval 8](#_Toc44589278)

[Request to play 9](#_Toc44589279)

[Start game 10](#_Toc44589280)

[End game 11](#_Toc44589281)

[Close connection 11](#_Toc44589282)

[P2P 12](#_Toc44589283)

[Authentication phase 12](#_Toc44589284)

[In-game messages 13](#_Toc44589285)

[BAN logic proof of client-server authentication 15](#_Toc44589286)

[Real protocol 15](#_Toc44589287)

[Assumptions 15](#_Toc44589288)

[Objectives 15](#_Toc44589289)

[Idealized protocol 15](#_Toc44589290)

[Proof 1](#_Toc44589291)

[M2: 1](#_Toc44589292)

[M3: 2](#_Toc44589293)

[M4: 2](#_Toc44589294)

[M5: 3](#_Toc44589295)

[BAN logic proof of P2P authentication 4](#_Toc44589296)

[Real protocol 4](#_Toc44589297)

[Assumptions 4](#_Toc44589298)

[Objectives 4](#_Toc44589299)

[Idealized protocol 1](#_Toc44589300)

[Proof 1](#_Toc44589301)

[M0’: 1](#_Toc44589302)

[M0’’: 2](#_Toc44589303)

[M2: 2](#_Toc44589304)

[M3: 4](#_Toc44589305)

[M4: 4](#_Toc44589306)

[M5: 5](#_Toc44589307)

# 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 idle 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

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.

TODO: SAY THAT THE SHARED SECRET IS HASHED

TODO: SAY WHAT’S AUTHENTICATED IN AAD

TODO: SAY HOW WE PROTECTED ALL THE MESSAGES FROM REPLAY

## 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 some drawbacks in terms of memory: in fact, too much memory would be allocated for a simple task such 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

## Preliminary information

In the following pictures, for sake of readability, are omitted all the fields that include the dimension of another field in the message (e.g. you will not find CERT\_LEN and CERT fields, but CERT one only).

Moreover, for the sake of readability again, in fields encrypted with symmetric encryption using GCM won’t be shown the IV and TAG fields.

In general the encrypted fields are structured like this:

|  |  |  |
| --- | --- | --- |
| 16 B | 16 B |  |
| IV | TAG | Encrypted payload |

Here’s a simple legend for a better understanding of the schemas of the next paragraphs:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| Header fields | Signed fields | Encrypted fields with GCM | Other not encrypted fields |

## Client-server

### Authentication phase

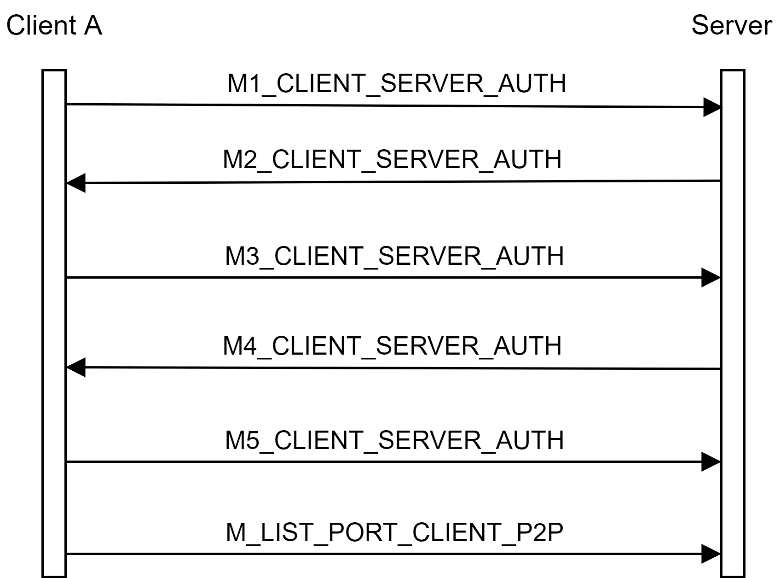
#### DESCRIPTION

The authentication phase is divided into 6 stages: client and server perform an exchange of challenges to avoid replay attacks and to assess the freshness of the protocol instance. Meanwhile, they negotiate a key using Diffie-Hellman to guarantee perfect forward secrecy. We actually use ECDHKE, since it guarantees a high level of security with shorter keys compared to the ones used in common DHKE.

After the key derivation, the two parties exchange M4 and M5, that are needed for key confirmation.

At the end of the protocol, the client informs the server on which port it will listen for incoming P2P connections.

#### SCHEME



#### M1\_CLIENT\_SERVER\_AUTH

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 B | 4 B | 16 B | 7 B | 32 B |
| OPCODE | PAYLOAD\_LEN | A | S | CH\_S |
| Opcode | Payload length | Nickname of the client | Nickname of the server | Challenge to server |

#### M2\_CLIENT\_SERVER\_AUTH

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 B | 4 B |  | 32 B | 7 B | 16 B | 32 B |  |  |
| OP | LEN | CERT | CH\_A | S | A | CH\_S | Y\_S | S(S,A,CH\_S,Y\_S) |
| Opcode | Paylen | Server cert | Ch to client | Serv id | Cli id | Ch to server | Server’s DH pubk | Signature with server’s privk |

#### M3\_CLIENT\_SERVER\_AUTH

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 B | 4 B | 16 B | 7 B | 32 B |  |  |
| OP | LEN | A | S | CH\_A | Y\_A | S(A,S,CH\_A,Y\_A) |
| Opcode | Paylen | Cli id | Serv id | Ch to client | Client’s DH pubk | Signature with client’s privk |

#### M4\_CLIENT\_SERVER\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 16 + 7 = 23 B (plaintext) |
| OP | LEN | E(Kas, S||A) |
| Opcode | Paylen | Key confirmation: the two parties’ ids are encrypted with Kas |

#### M5\_CLIENT\_SERVER\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 16 + 7 = 23 B (plaintext) |
| OP | LEN | E(Kas, A||S) |
| Opcode | Paylen | Key confirmation: the two parties’ ids are encrypted with Kas |

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

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

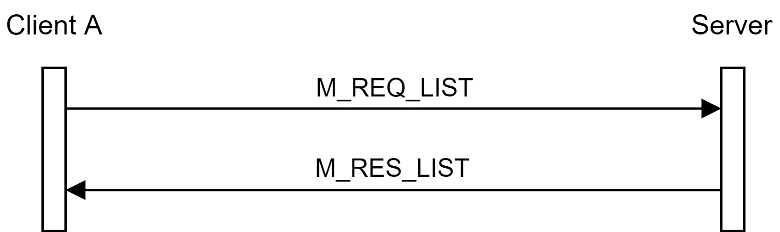
NOTE: This message is important only in case of execution of more instances of the program on the same machine, otherwise a common port could be assigned to P2P connections

### List retrieval

#### DESCRIPTION

The client asks for the list of all the logged in users at the moment, using a nonce (corresponding to the inner counter authenticated in GCM) to avoid having an empty payload. The server responds with the list of users using encryption to avoid any manipulations.

#### SCHEME



#### M\_REQ\_LIST

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 4 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKAS(counter) |
| Opcode | Payload length | Payload |

#### M\_RES\_LIST

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B |  |
| OPCODE | PAYLOAD\_LEN | EKAS(LIST) |
| Opcode | Payload length | Communication of the list of logged in clients |

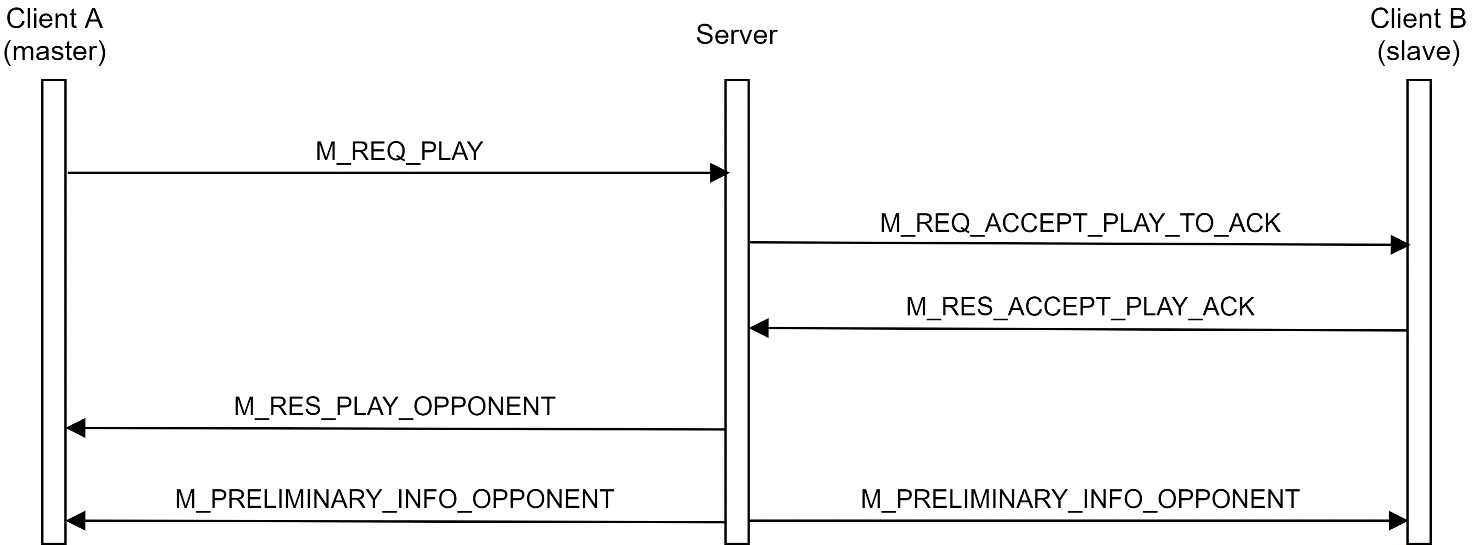
### Request to play

#### 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.

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 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_OPPONENT) |
| Opcode | Payload length | Request to play with the specified opponent |

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

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 16 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_OPPONENT) |
| Opcode | Payload length | The server asks a client if it wants to play with the specified opponent |

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

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 1 + 16 = 17 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKas(RESPONSE\_1BYTE, ID\_OPPONENT) |
| Opcode | Payload length | Response of the slave client to a certain opponent |

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

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 1 + 4 + 16 = 21 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKas(RESPONSE\_1BYTE, OPPONENT\_PORT, ID\_OPPONENT) |
| Opcode | Payload length | Response forwarded to the master client, together with the opponent’s listening port |

#### M\_PRELIMINARY\_INFO\_OPPONENT

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B |  |
| OPCODE | PAYLOAD\_LEN | EKas(ID\_LOCAL, ID\_OPPONENT, 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.

#### SCHEME



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

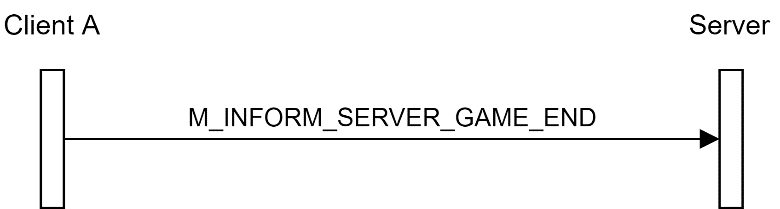
|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 4 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKas(counter) |
| Opcode | Payload length | The client informs the server that it will not be reachable anymore to play |

### End game

#### DESCRIPTION

When the game ends, the clients tell the server that they’re now reachable for playing with other opponents.

#### SCHEME



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

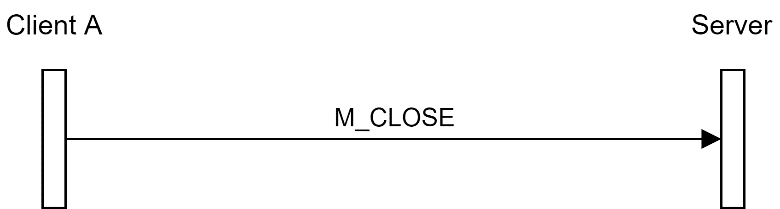
|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 4 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKas(counter) |
| Opcode | Payload length | The client informs the server that it’s now reachable for playing |

### Close connection

#### DESCRIPTION

When the user decides to disconnect from the server, it at first has to tell the server to close the socket on the thread that communicates with it.

#### SCHEME



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

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 4 B (plaintext) |
| OPCODE | PAYLOAD\_LEN | EKas(counter) |
| Opcode | Payload length | Payload |

## P2P

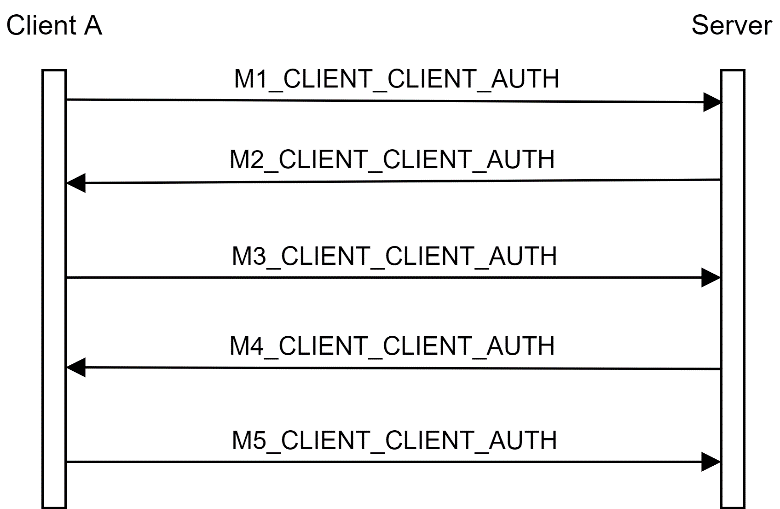
### Authentication phase

#### 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.

#### SCHEME



#### M1\_CLIENT\_CLIENT\_AUTH

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 B | 4 B | 16 B | 16 B | 32 B |
| OPCODE | PAYLOAD\_LEN | M | S | CH\_S |
| Opcode | Payload length | Nickname of the master | Nickname of the slave | Challenge to slave |

#### M2\_CLIENT\_CLIENT\_AUTH

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 B | 4 B | 16 B | 16 B | 32 B |  |  |
| OP | LEN | S | M | CH\_S | Y\_S | S(S,M,CH\_S,Y\_S) |
| Opcode | Paylen | Sla id | Mast id | Ch to slave | Slave’s DH pubk | Signature with slave’s privk |

#### M3\_CLIENT\_CLIENT\_AUTH

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 B | 4 B | 16 B | 16 B | 32 B |  |  |
| OP | LEN | M | S | CH\_M | Y\_M | S(M,S,CH\_M,Y\_M) |
| Opcode | Paylen | Mast id | Sla id | Ch to master | Master’s DH pubk | Signature with master’s privk |

#### M4\_CLIENT\_CLIENT\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 16 + 7 = 23 B (plaintext) |
| OP | LEN | E(Kms, S||M) |
| Opcode | Paylen | Key confirmation: the two parties’ ids are encrypted with Kas |

#### M5\_CLIENT\_CLIENT\_AUTH

|  |  |  |
| --- | --- | --- |
| 1 B | 4 B | 16 + 7 = 23 B (plaintext) |
| OP | LEN | E(Kms, M||S) |
| Opcode | Paylen | Key confirmation: the two parties’ ids are encrypted with Kas |

### 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.

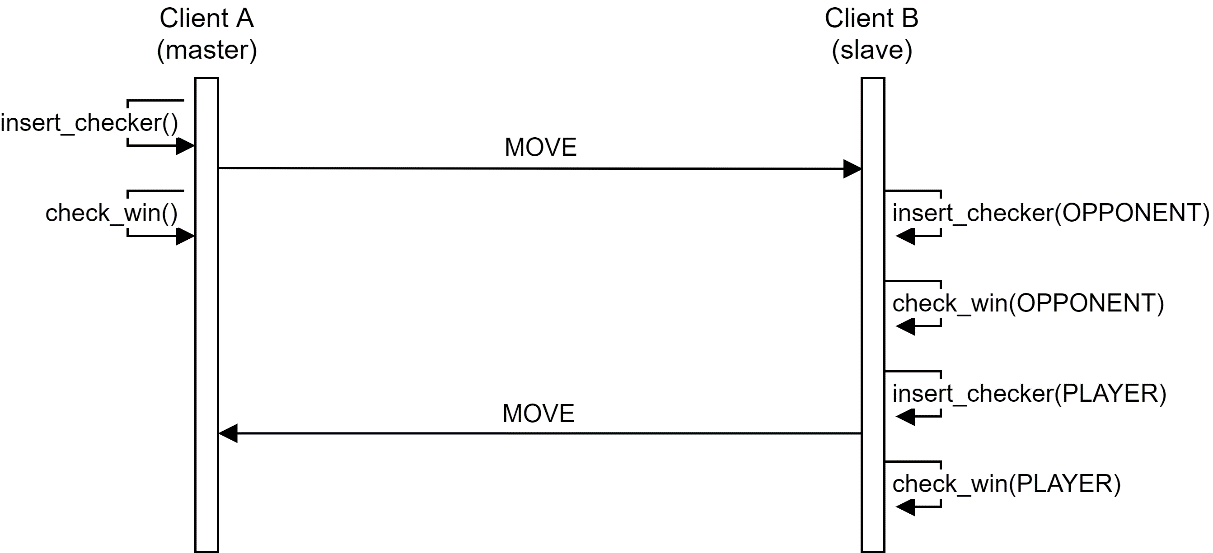
The following scheme represents not only the exchange of messages in the network in the game phase, but also the operations done by the players.

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) || IV || TAG (IV + OPCODE + PAYLOAD\_LEN) |
| 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 1st postulate:

If the certificate is valid and not revoked, we can assume it’s fresh and we can apply the 3nd postulate:

Now the client trusts the server’s public key. Applying the 1st postulate:

Applying the 2nd postulate:

Applying the 3rd postulate we prove objective 1:

We can also prove objective 5 by stating that:

### M3:

Applying 1st postulate:

Applying 2nd postulate:

Applying 3rd postulate we prove objective 2:

We can also prove objective 6 by stating that:

### M4:

Applying 1st postulate:

Applying 2nd postulate:

Applying 3rd postulate we prove objective 3:

### M5:

Applying 1st postulate:

Applying 2nd postulate:

Applying 3rd postulate we prove objective 3:

# BAN logic proof of P2P authentication

## Real protocol

## Assumptions

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

## Objectives

## Idealized protocol

## Proof

### M0’:

Applying the 1st postulate:

Applying the 2nd postulate:

Applying the 3rd postulate:

### M0’’:

Applying the 1st postulate:

Applying the 2nd postulate:

Applying the 3rd postulate:

### M2:

Applying the 1bt postulate:

Applying the 2nd postulate:

Applying the 3rd postulate we prove objective 1:

We can also prove objective 5 by stating that:

### M3:

Applying 1bt postulate:

Applying 2nd postulate:

Applying 3rd postulate we prove objective 2:

We can also prove objective 6 by stating that:

### M4:

Applying 1bt postulate:

Applying 2nd postulate:

Applying 3rd postulate we prove objective 3:

### M5:

Applying 1bt postulate:

Applying 2nd postulate:

Applying 3rd postulate we prove objective 3: