Università di Pisa

Master of Science in Computer Engineering

Project Report
Four-in-a-row

Foundations of Cybersecurity

T. Billi, E. Casapieri, A. Di Donato

Academic Year 2019/2020

Contents

Chapter 1

Project Specification

The project carried out consists in the implementation of the famous game four in a row. Once users have registered with the server, they can challenge each other and once the connection is established, they can play with each other.

After the authentication phase the communication between client and server, and between two clients in play is confidential and authenticated. Specifically, the server has a public key certified by a certification authority and has the public keys of the various users registered with it; each client instead has a private key, protected by a password, which corresponds to the public key pre-installed on the server.

The game experience is mainly based on two distinct phases:

- *Main menu*: During the following phase the user is in the main menu of the game where he can challenge another user online at that time. Once the authentication phase is completed, the user is directed to this first step.
- *Game*: Here the user plays four in a row with the opponent who has accepted his challenge.

1.1 Main menu commands

When the user is here, he only communicates with the server that responds to certain requests sent by the online users.

More precisely, the commands that a user can use, here, are the following (the syntax reported is the same to be used within the application):

- *!help*: shows the available commands
- *!users*: shows users who are in a state of waiting for a challenge
- *!challenge <username>*: challenge the user with the username specified in the command
- *!wait*: the user waits to receive a challenge request from another user. This wait lasts for a limited period of time. After this time, he must re-enters the command if he wants to return to the waiting state.

- *!quit*: close the connection with the server

Note that during the sending/waiting phase of the challenge, the user who receives the challenge through the *!wait* command listens to a specific port to which the challenger connects. This has received the listening port from the server after sending the *!challenge* command; it is in fact the server that forwards the address, to which the requested user is available, to the challenger.

When a user in waiting status receives a challenge request, he or she must respond whether or not he accepts it within a set amount of time otherwise such request is automatically rejected.

1.2 Game commands

If the user is at this stage, it means that the connection to the opponent from whom (to whom) the game request was received (sent) has already been successfully established. Both are then ready to play, once it has been established who starts first, users will be able to use the following commands:

- *!move <column_number>*: places the token in the column number specified in the command
- *!concede*: gives the win to the adversary

If the game has ended successfully due to victory, draw or surrender, both players return to the main menu and the connection between the two is closed. Otherwise, in addition to closing the connection between the two clients, the connection between the server and the client(s) causing the error is closed.

Chapter 2

Implementation

The source code is organized into 3 modules:

- *Client*: implements the client side of the application
- *Server*: implements the server side of the application
- *Game*: provides the implementation of the commands used by the users to play

2.1 Server

In the implementation of the server have been made the following design choices:

- In order to accept and manage the connections with different clients at the same time, the server has been built by means of poll function included in the C++ library `<sys/poll.h>`.

  Let's describe briefly what it consists of. The main task of the poll is to control and wait for one of a set of file descriptors to become ready to perform I/O operations, for instance read or write a socket.

  In the following is reported the definition:

  ```
  int poll(struct pollfd *fds, nfds_t nfds, int timeout)
  ```

  The set and the number of file descriptors to be monitored have been passed as first and second argument respectively, whereas the timeout argument has been set to -1 to make it unlimited because the server should block until a file descriptor become ready. For each file descriptor the input parameter events specifies the event the application is interested in; it has been set to `POLLIN`, namely there is data to read. Our server is organized as an infinite loop in which poll is called at the beginning of each iteration waiting for incoming connections or for incoming data on any of the connected sockets. If one or more descriptors are readable, in order to determine which ones they are, an internal loop finds the ones that returned `POLLIN` distinguishing the listening socket and the active sockets which have already established connections. If the listening socket is readable, accept is executed and the new connection is added to the set monitored by poll. At the disconnection of a client it is necessary to squeeze together the `fds` array and decrement the number of file descriptors.

- In order to keep track of the information (such as username, status, public key and listening port) of the users connected to the server, a struct client for each one has been defined. Each struct client is put into a vector easily accessible by means of an index that is directly linked to the index of the corresponding socket inside the set of file descriptor of the poll. This makes the management more explicative. Moreover, the choice of the vector is driven by the fact that it can shrink or expand as needed at run time making the handling of connection and disconnection of a user more comfortable. In order to avoid out-of-bound accesses the operator `at(index)` has been used.

- The following statuses have been associated to the users from the point of view of the server:

  - `OFFLINE`: The user is registered in the server (it stores his/her public key) but is not connected. Server does not have any element in the vector vec_client associated to that user.
  - `UNREGISTERED`: The user has not been registered in the server.
  - `CERT_REQ`: The user assumes this status as soon as the connection is accepted. The server serializes its certificate and sent it to the user along with the size.
  - `LOGIN`: After the verification of the server certificate, the user login with the username so that the server can associated one of the public key already pre-installed to that user
  - `ONLINE`: When the user is in this status, the server completes the negotiation of the session key with him/her and then is able to perform the commands received. The user remains in this status even during the game.
  - `MATCHMAKING`: The user is waiting for challenge until the expiration of a timeout.

2.2 Client

In the implementation of the client have been made the following design choices:

- Each client has a TCP socket to connect to the server.

- Each client has a listening socket which accepts new connections from other clients that sent the request of challenge.

- Each client has a socket to communicate with another peer during the game.

- The reception of a challenge from another client is implemented synchronously. Indeed, by means of a special command "!wait" the client enter in matchmaking mode and make himself visible to other clients, waiting for receiving requests of challenge. In order to control socket behavior the `setsockopt()` is used. The option `SO_RCVTIMEO` sets the timeout value that specifies the maximum amount of time an input function waits until it completes.

  It accepts a `timeval` structure with the number of seconds specifying the limit on how long to wait for an input operation to complete.

In this way if a receive operation has blocked for this much time without receiving additional data, it shall return a specific error to be checked. In the last case the user is not in matchmaking mode anymore until the next execution of the *!wait* command.

- In order to let a client know if the sender of the challenge has been disconnected in the meantime he/she was thinking of accepting or refusing it, an additional exchange of messages has been created.
- The client has a limited time to send the response to the server and if the time expires the response is considered as negative.

Chapter 3

The protocol

3.1 Client-Server

The exchange of messages performed in order to establish a secure connection and authentication between client and server is shown in figure 3.1.

As soon as a new client tries to connect to the server, the latter sends the certificate to the user. If the certificate is valid, and the subject identifier reported in the certificate corresponds to the server's id, the client proceeds to send the username in clear.

If the user is registered on the server (i.e. the corresponding public key exists in the directory), the server sends a nonce to the client. Similarly, the client generates a nonce, that sends to the server.

The client computes a DH public key, which (s)he sends to the server with the two nonces, along with a signature of said quantities.

At this point, the server proceeds to the verification of the signature, and, if correct, sends in turn its DH public key that it previously computed, with the aforementioned signatures.

If the signature is correctly verified on the client's side, the session key is computed by truncating the first 128 bits of the SHA-256 hash of the derived secret.

Once the session key has been generated, the subsequent messages between the two parties are encypted with AES-128 in GCM, with the previously generated session key.

These steps are repeated at every new connection.

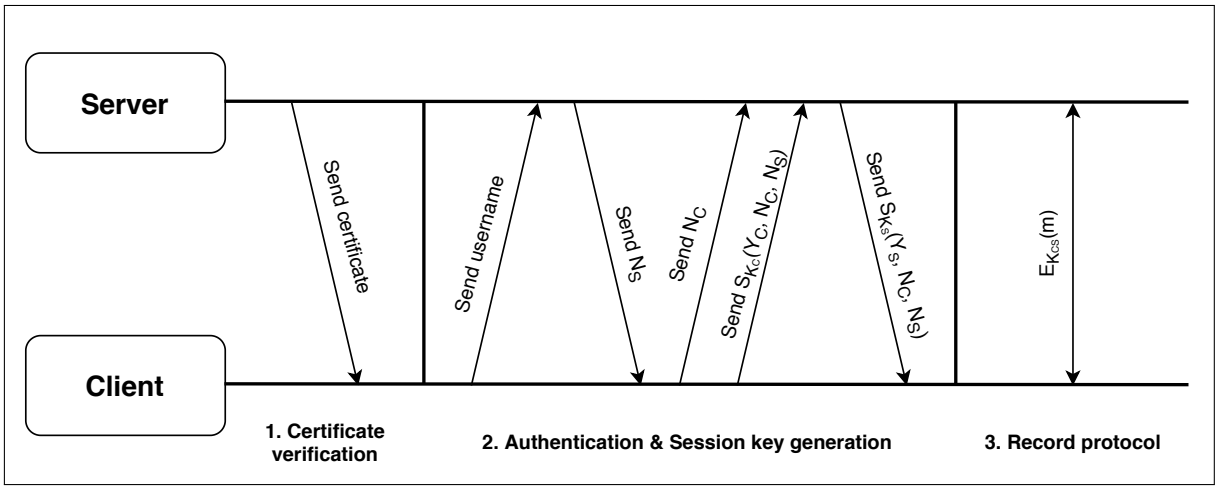

Figure 3.1: Message exchange sequence between client and server.

3.2 Peer-to-peer

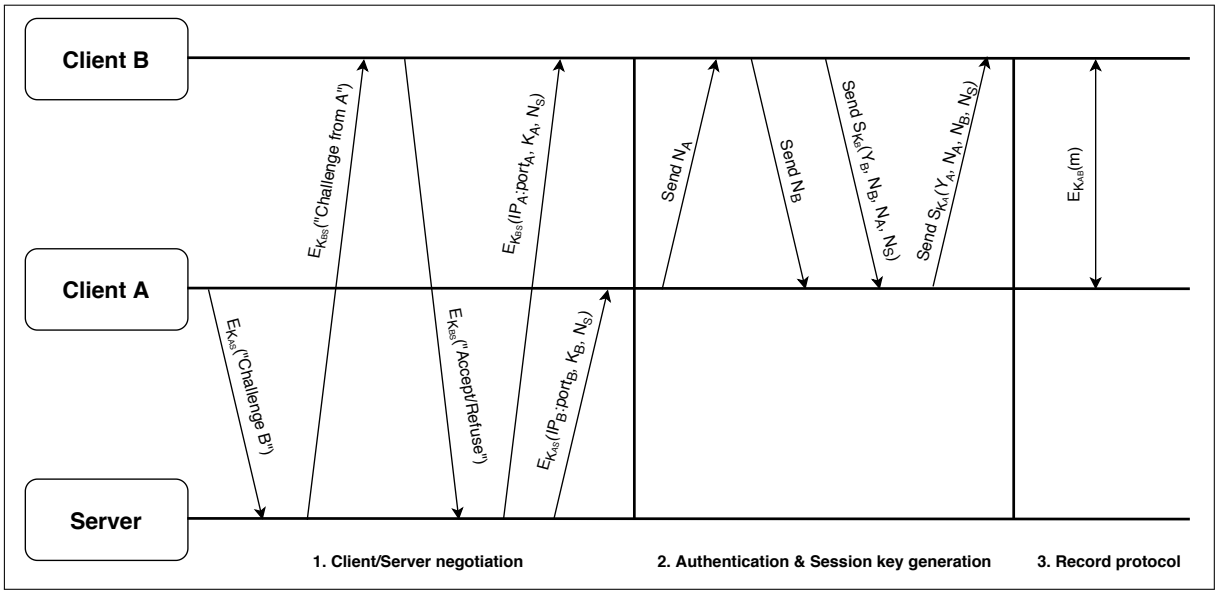

Figure 3.2: Message exchange to establish a connection between two clients.

After the handshake phase with the server, a user can decide to challenge another user that is authenticated on that server who is ready to be challenged.

By issuing the *!challenge <username>* command, the server checks the status of the username that was specified. If the state is `MATCHMAKING`, the challenge request is forwarded to the user. In any other case, the server responds with an error code specifying the user status, and the protocol is interrupted.

If the challenged user was indeed in `MATCHMAKING`, (s)he can either accept or refuse the challenge. In either case, the response is forwarded to the challenge initiator. If the response is "N", the protocol is aborted, and both users return to the main menu. If the challenge is accepted ("Y"), the server sends to both users all the information they need to establish a peer to peer connection. Specifically, this includes:

- the other user's IP address and listening port;
- the other user's public key;
- an additional nonce (the $match\_id$), that is the same for both users.

At this point, the users connect directly, and authenticate each other and negotiate a session key in a similar fashion to what's been seen in the previous section. The only difference is that in this case, also the

When a user wants to challenge another user that is ready to be challenged, (s)he sends a challenge request to the $match\_id$ generated by the server is signed and verified by both parties.

Chapter 4

Messages

4.1 Handshake Protocol

4.1.1 Certificate messages

This messages are sent by the server to the user that open a connection with him. The message are sent in clear; first is sent the length of the certificate and then the certificate itself.

Figure 4.1: Certificate message format

4.1.2 Username messages

These messages are sent when the user has verified the server's certificate. It contains the username of the user that want to log in the application. First is sent the length of the username and the the username itself.

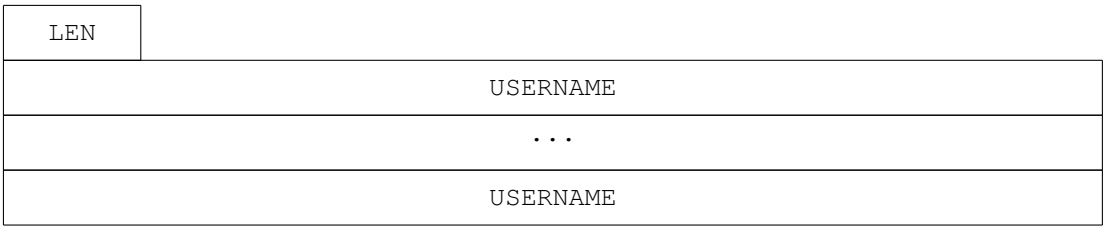

Figure 4.2: Username messages format

4.1.3 Nonce message

This message is sent by the user and the server to the other party when there is a need for a nonce to use it in the authentication of the other party.

Since the size of the nonce has been set as a project choice to, 16 byte, sending the size is not required.

Figure 4.3: Nonce messages format

4.1.4 Diffie-Hellman public key message

These messages are sent during the negotiation of the session key, both for peer-to-peer connections and client-server connections. The first part of the message contains the length of the public key, and the second part contains the key itself.

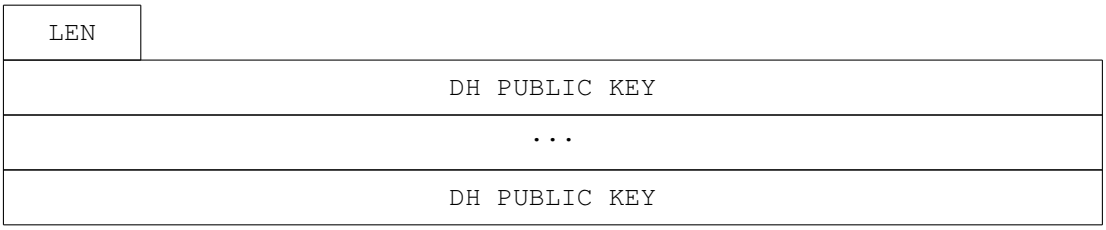

Figure 4.4: DH public key messages format

4.1.5 Signature messages

These messages are sent when one of the party has generated and sent a nonce, the nonce of the other party and has generated a Diffie-Hellman public key. The signature is in fact generated using those three parameters; first is sent the length of the signature then is sent the signature itself.

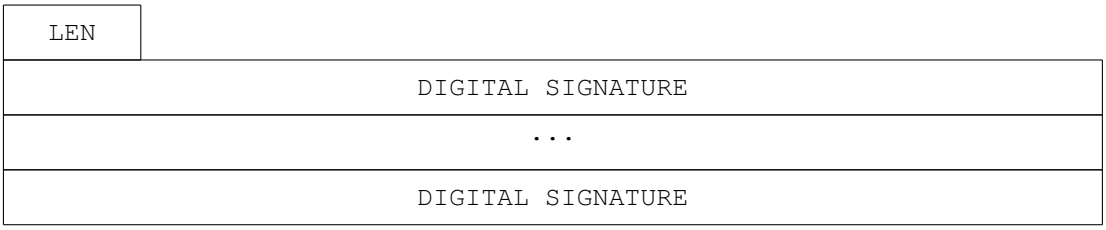

Figure 4.5: Digital signature messages format

4.2 Record Protocol

All the messages in the record protocol are encrypted using AES-128, with Galois/-Counter Mode. The following quantities are present in each message in this protocol:

- `IV`: initialization vector; nonce generated at each new message.
- `CNT`: counter. The use of this quantity, that is increased at each new message exchanged between two parties, avoids possible replay attacks.

- AAD: additional authenticated data, that includes both IV and CNT.
- TAG: authentication tag that proves the integrity of the message.

4.2.1 Opcode message

The following messages[1], encrypted in GCM mode, are sent by a client every time he runs one of these commands:

- !users: is sent only the message containing the opcode SHOW_ONLINE_USERS_OPC.
- *!challenge <username>*: after the opcode CHALLENGE_REQUEST_OPC is received by the server, the client send first a message containing the length of the username to challenge and after that is sent the message containing the username itself.
- *!wait*: the message containing the corresponding opcode (WAITING_REQ_OPC) is sent to the server, if the timeout expires because no challenges arrive the user send another message containing the opcode END_OF_MATCHMAKING. Otherwise if a challenge arrives (message containing the opcode NEW_CHALLENGE_REQ_OPC) the client send his response that is a message containing the opcode CHALLENGE_ACCEPTED or CHALLENGE_REFUSED (please note that if the response it is not given in a certain amount of time, it is sent the opcode CHALLENGE_REFUSED).
- *!concede*: is sent only the message containing the opcode GAME_OVER_CONCEDE

When the game ends between the players, an opcode of GAME_OVER or GAME_OVER_TIE.

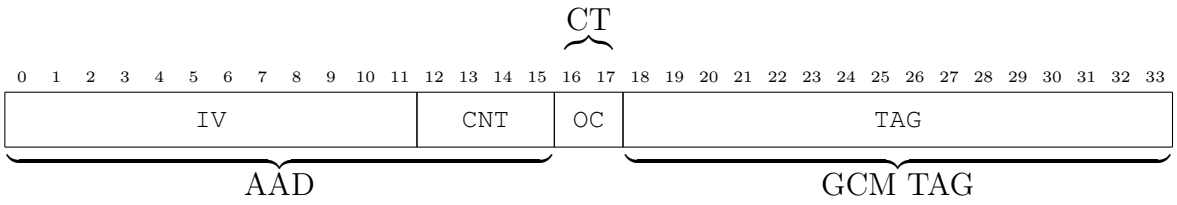

Figure 4.6: Encrypted opcode message format.

[1]In Appendix A a list of all the opcodes used by the application can be found.

4.2.2 Secure message

4.2.2.1 Length message

This message is sent each time before sending a message with content different from an opcode. Basically its purpose is to indicate to the recipient the length of the plaintext contained in the message sent immediately after this.

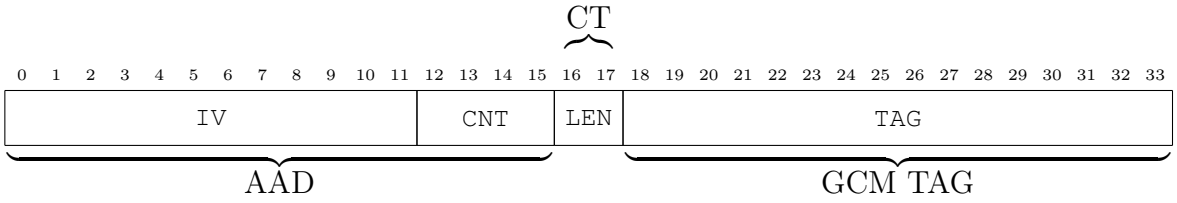

Figure 4.7: Encrypted length message format.

4.2.2.2 Encrypted message

This message is sent, as said in the previous section, every time it has to be sent a message that is different from an opcode and with variable length. In our application this is only the case when is sent:

- the list of the username corresponding to users in a `MATCHMAKING` state.
- the username of the user to be challenged.
- the IP address corresponding to the users involved in a challenge.
- the public key corresponding to the users involved in a challenge.

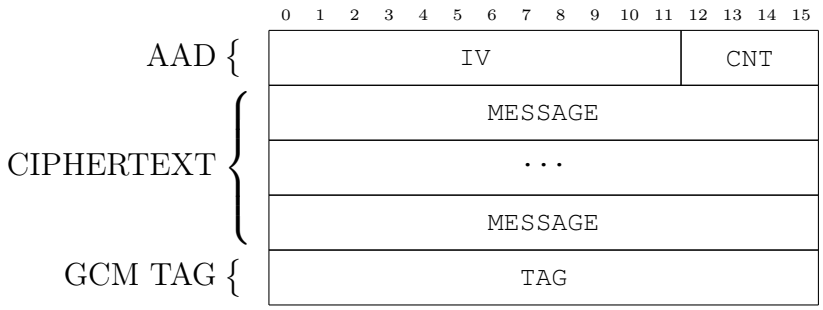

Figure 4.8: Encrypted message format.

It should also be noted that encrypted messages are sent in GCM mode that do not require an estimate of the length because it is already known. That is the case of:

- send the move to perform in the game.
- send of the listening port where receiving the challenge requests.

Chapter 5

BAN logic proof of key exchange protocol

5.1 Real protocol

The real protocol to establish a session key between a client and the server is presented below. The handshake between two peers follows the same model, and for this reason we decided to omit it.

$$\begin{array}{ll|ll} M_1 & C \to S : C & M_3 & C \to S : Y_C, N_C, N_S, \{h(Y_C, N_C, N_S)\}_{K_C^{-1}} \\ M_2 & S \to C : N_S & M_4 & S \to C : Y_S, N_S, N_C, \{h(Y_S, N_S, N_C)\}_{K_S^{-1}} \end{array}$$

5.2 Idealized protocol

In the idealized protocol, the first message (containing the client's identification) is dropped, as it doesn't add anything for the purpose of this proof, and so are the hashes. The idealized protocol is the following.

$$\begin{array}{c|c} M_2 \quad S \to C : N_S & \\ M_3 \quad C \to S : \{N_C, N_S, \overset{Y_C}{\longmapsto} C\}_{K_C^{-1}} & M_4 \quad S \to C : \{N_S, N_C, \overset{Y_S}{\longmapsto} S\}_{K_S^{-1}} \end{array}$$

5.3 Assumptions

Since there is a certificate that proves the authenticity of the server's public key, and the server itself is assumed to have and trust all the public keys of the registered clients, both parties' public keys are assumed to be known.

$$\begin{array}{c|c} C \mid\equiv \overset{K_S}{\longmapsto} S & S \mid\equiv \overset{K_C}{\longmapsto} C \\ C \mid\equiv \#(N_C) & S \mid\equiv \#(N_S) \\ C \mid\equiv \#(\overset{Y_C}{\longmapsto} C) & S \mid\equiv \#(\overset{Y_S}{\longmapsto} S) \\ C \mid\equiv S \Rightarrow N_S & S \mid\equiv C \Rightarrow N_C \\ C \mid\equiv S \Rightarrow \overset{Y_S}{\longmapsto} S & S \mid\equiv C \Rightarrow \overset{Y_C}{\longmapsto} C \end{array}$$

5.4 Goals

At the end of the proof, we want to prove that both parties know all the Diffie-Hellman parameters needed to derive a shared secret, and consequently generate the session key, along with the other quantities needed to prove freshness and authenticity. More specifically, we want to prove the following:

$$\begin{array}{c|c} C \mid\equiv (N_C, N_S, \overset{Y_C}{\longmapsto} C, \overset{Y_S}{\longmapsto} S) & S \mid\equiv (N_S, N_C, \overset{Y_S}{\longmapsto} S, \overset{Y_C}{\longmapsto} C) \\ C \mid\equiv S \mid\equiv (N_C, N_S, \overset{Y_C}{\longmapsto} C, \overset{Y_S}{\longmapsto} S) & S \mid\equiv C \mid\equiv (N_S, N_C, \overset{Y_S}{\longmapsto} S, \overset{Y_C}{\longmapsto} C) \end{array}$$

5.5 Proof

After M_2 $(S \to C : N_S)$, C only sees a nonce, but no authenticity whatsoever is provided. This value is going to be used in a subsequent message.

$$C \triangleleft N_S \tag{5.1}$$

After M_3 $(C \to S : \{N_C, N_S, \overset{Y_C}{\longmapsto} C\}_{K_C^{-1}})$, S is able to verify C's identity, and has confirmation that the nonce has been received. With this message, S also receives C's public key.

$$\text{by applying the message meaning rule,} \qquad S \mid\equiv C \mid\sim (N_C, N_S, \overset{Y_C}{\longmapsto} C) \tag{5.2a}$$

$$\text{by applying the nonce verification rule,} \qquad S \mid\equiv C \mid\equiv (N_C, N_S, \overset{Y_C}{\longmapsto} C) \tag{5.2b}$$

$$\text{by applying the jurisdiction rule,} \qquad S \mid\equiv (N_C, \overset{Y_C}{\longmapsto} C) \tag{5.2c}$$

After M_4 $(S \to C : \{N_S, N_C, \overset{Y_S}{\longmapsto} S\}_{K_S^{-1}})$, C is able to verify S's identity, and has confirmation that the nonce has been received. With this message, C also receives S's public key.

$$\text{by applying the message meaning rule,} \qquad C \mid\equiv S \mid\sim (N_S, N_C, \overset{Y_S}{\longmapsto} S) \tag{5.3a}$$

$$\text{by applying the nonce verification rule,} \qquad C \mid\equiv S \mid\equiv (N_S, N_C, \overset{Y_S}{\longmapsto} S) \tag{5.3b}$$

$$\text{by applying the jurisdiction rule,} \qquad C \mid\equiv (N_S, \overset{Y_S}{\longmapsto} S) \tag{5.3c}$$

After this step, both parties can confirm they are communicating with whom they're supposed to and have all the parameters they need to derive a session key. Perfect forward secrecy of the algorithm is guaranteed by the fact that a new public key is chosen at random by both parties at the beginning of a new handshake.

Appendices

Appendix A

List of opcodes

```
1  /*** MAIN APPLICATION CODES ***/
2
3  const unsigned int SHOW_ONLINE_USERS_OPC = 1;
4  const unsigned int CHALLENGE_REQUEST_OPC = 4;
5  const unsigned int WAITING_REQ_OPC       = 5;
6  const unsigned int NEW_CHALLENGE_REQ_OPC = 6;
7  const unsigned int END_OF_MATCHMAKING    = 7;
8  const unsigned int CHALLENGE_ACCEPTED    = 8;
9  const unsigned int CHALLENGE_REFUSED     = 9;
10 const unsigned int ERROR_CHALLENGE_SELF  = 20;
11
12
13 /*** GAME CODES ***/
14
15 const unsigned int GAME_OVER             = 101;
16 const unsigned int GAME_OVER_TIE         = 102;
17 const unsigned int GAME_OVER_CONCEDE     = 103;
```