Università di Pisa

Computer Engineering

Foundations of Cybersecurity

secureCom

Group Project Report

TEAM MEMBERS:
Francesco Iemma
Yuri Mazzuoli
Olgerti Xhanej

Academic Year: 2020/2021

Contents

Chapter 1

Specifications

The project consist on an application for secure communication between 2 clients through an intermediate server.
The server have to:

- authenticate clients on connecting to it (with pre-shared public key)
- authenticate ifself with a certificate
- provide the list of online clients
- relay messsages from one client to another, together with chat requests and response
- provide to a client the public key of another client, in order to permit a secure communication between them

A client have to:

- authenticate the server on connecting to it (with the certificate)
- authenticate himself with its public key

A client can:

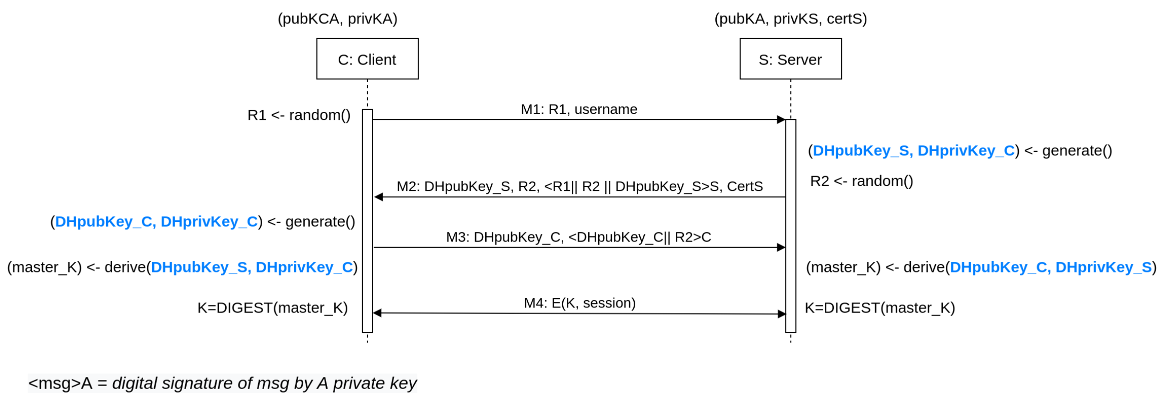
- print the list of online clients
- authenticate another client via its public key obtained from the server
- request to chat with another client
- answer to a chat request (if not already involved in another chat)
- when in a chat, exchange text messages with another client or close the chat

Chapter 2

Design choises

2.1 Client Server Handshake

In order to authenticate themselvse and enstablish a session key to securely communicate, a client and the server have to exchange handshake messages. We implement this protocol to provide perfect forward secrecy, starting from the pre-shared cryptographic quantities (public keys and certificates):



(pubKCA, privKA)
(pubKA, privKS, certS)
C: Client
S: Server
R1 <- random()
M1: R1, username
(DHpubKey_S, DHprivKey_C) <- generate()
R2 <- random()
M2: DHpubKey_S, R2, <R1|| R2 || DHpubKey_S>S, CertS
(DHpubKey_C, DHprivKey_C) <- generate()
M3: DHpubKey_C, <DHpubKey_C|| R2>C
(master_K) <- derive(DHpubKey_S, DHprivKey_C)
(master_K) <- derive(DHpubKey_C, DHprivKey_S)
M4: E(K, session)
K=DIGEST(master_K)
K=DIGEST(master_K)
<msg>A = digital signature of msg by A private key

Figure 2.1: Client Server Handshake Protocol Schema

This handshake is a custom implementation of an ephimeral Diffie-Hellman Key Exchange, in which we ensure protection against the man in the middle attack with random nuances (R1 and R2). The client is able to authenticate the server via it's certificate, signed by a trusted certification authority (the client is distributed along with CA's self-signed certificate); the server have a built-in list of all client's public keys. DH's private keys are deleted after the handshake and the key is generated by a digest of the shared seceret: in this way we provide security against a future disclousure of one of the long term private keys.

2.2 Chat Request

With the client-server handshake we build a secure tunnel between each client and the server. Using this tunnel every client can execute command on the server in a secure way; the most important (and complex) command is a chat request, of which we provide a scheme:

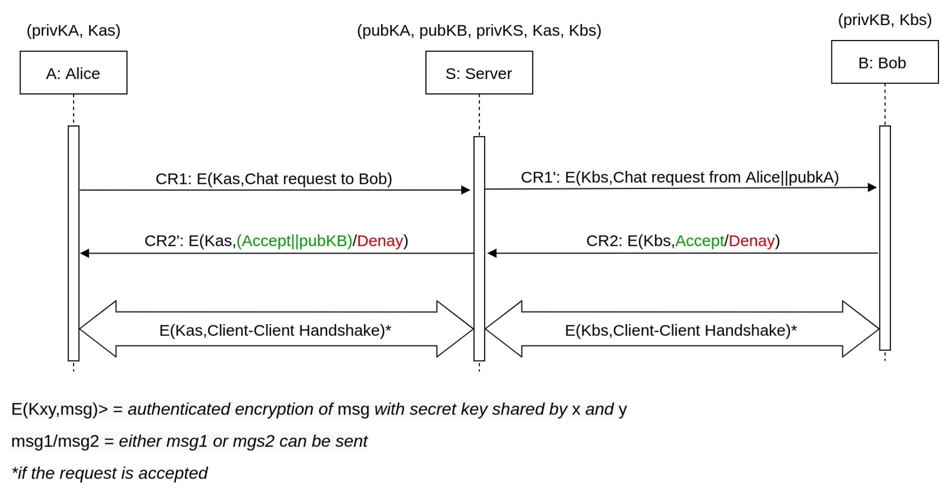

(privKA, Kas)
A: Alice
(pubKA, pubKB, privKS, Kas, Kbs)
S: Server
(privKB, Kbs)
B: Bob
CR1: E(Kas,Chat request to Bob)
CR1': E(Kbs,Chat request from Alice||pubkA)
CR2': E(Kas,(Accept||pubKB)/Denay)
CR2: E(Kbs,Accept/Denay)
E(Kas,Client-Client Handshake)*
E(Kbs,Client-Client Handshake)*
E(Kxy,msg)> = authenticated encryption of msg with secret key shared by x and y
msg1/msg2 = either msg1 or mgs2 can be sent
*if the request is accepted

Figure 2.2: Chat Request Protocol Schema

The server is obliged to communicate correct public keys.

2.3 Client to Client handshake

In order to guarantee a secure communication of the clients agaist the server, we perform an ephimeral Diffie-Hellman Key Exchange befor starting the chat. In this case the two parties already know each other public keys, because the server provided them.

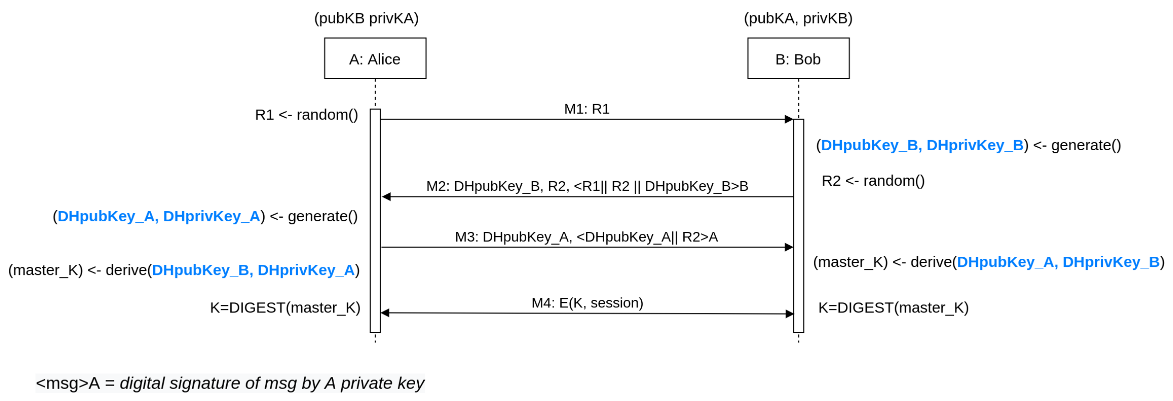

(pubKB privKA)
A: Alice
(pubKA, privKB)
B: Bob
R1 <- random()
M1: R1
(DHpubKey_B, DHprivKey_B) <- generate()
R2 <- random()
M2: DHpubKey_B, R2, <R1|| R2 || DHpubKey_B>B
(DHpubKey_A, DHprivKey_A) <- generate()
M3: DHpubKey_A, <DHpubKey_A|| R2>A
(master_K) <- derive(DHpubKey_B, DHprivKey_A)
(master_K) <- derive(DHpubKey_A, DHprivKey_B)
K=DIGEST(master_K)
M4: E(K, session)
K=DIGEST(master_K)
<msg>A = digital signature of msg by A private key

Figure 2.3: Client Client Handshake Protocol Schema

The server in not represented because it only retransmit messages from a client to the other without changing anything; if the server try to implement a ”man in the middle” attack he will only obtain a denial of service because the protocol is protected by private key signatures. Also in this case, DH keys are discared after the handshake, and future messages are numbered againts reply attacks.

2.4 Alghoritms and Protocols

We briefly describe the cypher suite we choose to use in order to guarantee security specifications.

2.4.1 Public Key Authentication

Long Term Keys The Cerification Authority (CA) have a `Publick Key`, included in a self-signed certificate; the corresponding private key is embedded in the program for certificate generation, SimpleAuthority [1]. CA's cerficiate and Revocation List (CRL) are exported and distributed with client executable. The server cerfificate, which is signed by CA, contains the `Publick Key` of the server; it is stored server side, and provided to the client during the handshake. Clients Public Keys are also stored in the server, while only the client hold its Private Key. All keys are `RSA Key Pairs with 2048-bits Public Keys` and are stored in `.pem` format, as certificates and CRLs are.

Short Term Keys Handshakes protocols are performed with the Ephimeral Diffie-Hellman Key Exchange. We choose to use the Elliptic Curve implenetation because is very efficent (in term of performance vs security strenght); we use `NID_X9_62_prime256v1` standard parameters that ensure 128-bits secuiry strenght with a 256-bits curve. We use the `SHA-256` digest of the `shared secret` as the session key. To ensure freshness in the challenge-response scheme we use `16 bytes nonces`.

2.4.2 Authenticated Encryption

From the handshake we obtain a 256-bits simmetric key which is used in `AES-256 GCM` authenticated encryprion protocol. We use a `16 bytes` tag for authentication of cyphertext and clear fields in the header. GCM scheme ensure a cyphertext size equal to the plaintext size; this makes programming easier, mantaining an optimal resistance against all known attacks. Messages excanged in sessions are numbered, starting from 1, to a maximum of $2^{32}-1$ (`0xFFFFFFFF`), which is the maximum integer representable on 32 bits; this will make possible to identify a message reply, done my an attacker. The session is automatically closed when a message with number `0xFFFFFFFF` is recived. Clients mantains two counter for the communication with the server: one for the next message to send and another for the next message to receive; the same thing is done for the communication with another client (to prevent the server to reply a message); also the server has to implement this behaveour in sessions with clients.

[1]https://simpleauthority.com

Chapter 3

Messages Format

3.1 Handshake

Client Server Handshake

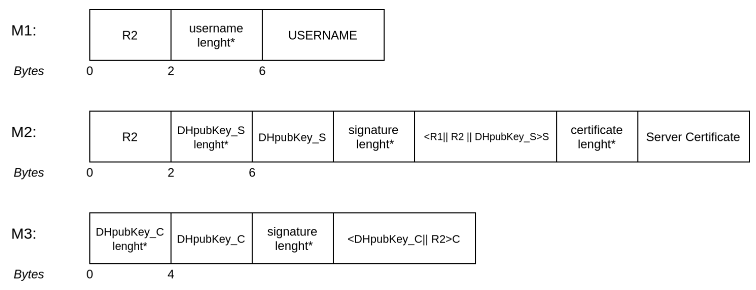


Figure 3.1: Client Server Handshake Message Format

Client Client Handshake

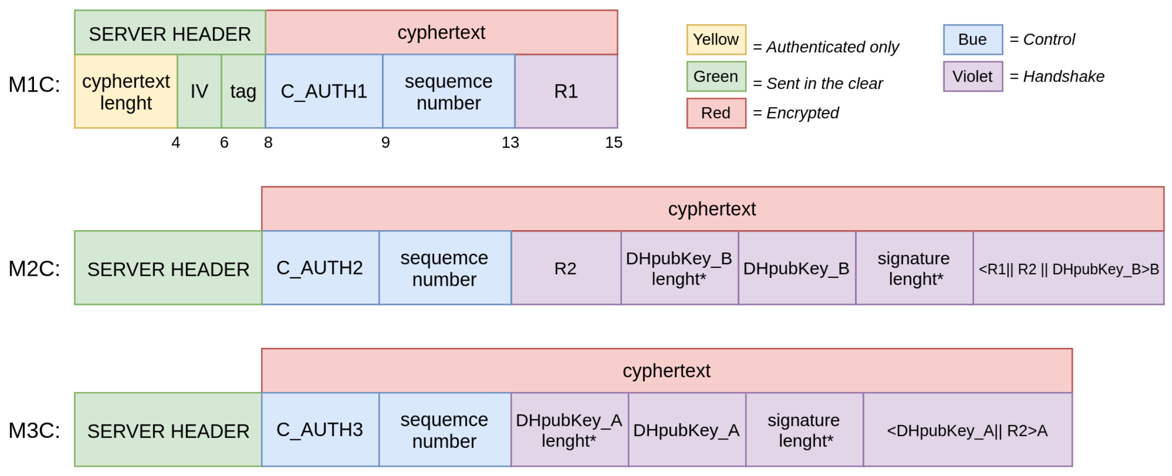


Figure 3.2: Client Client Handshake Message Format

3.2 Commands

Client Online List Request and Answer

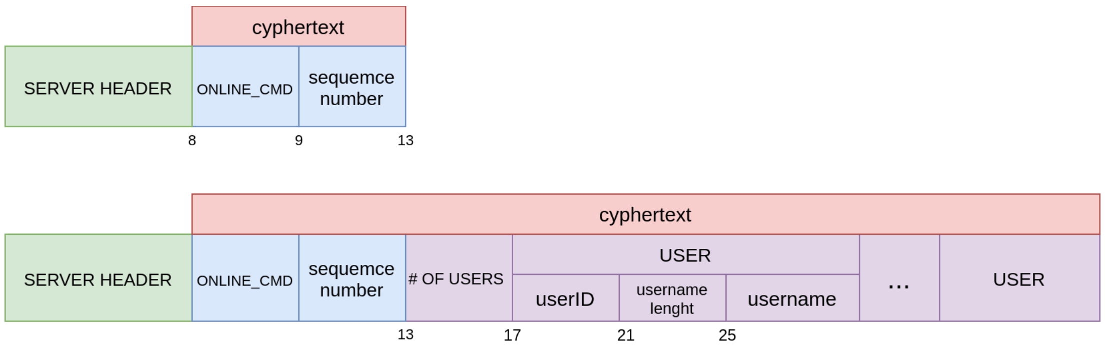


Figure 3.3: Client Online Message Format

Chat Request and Answers

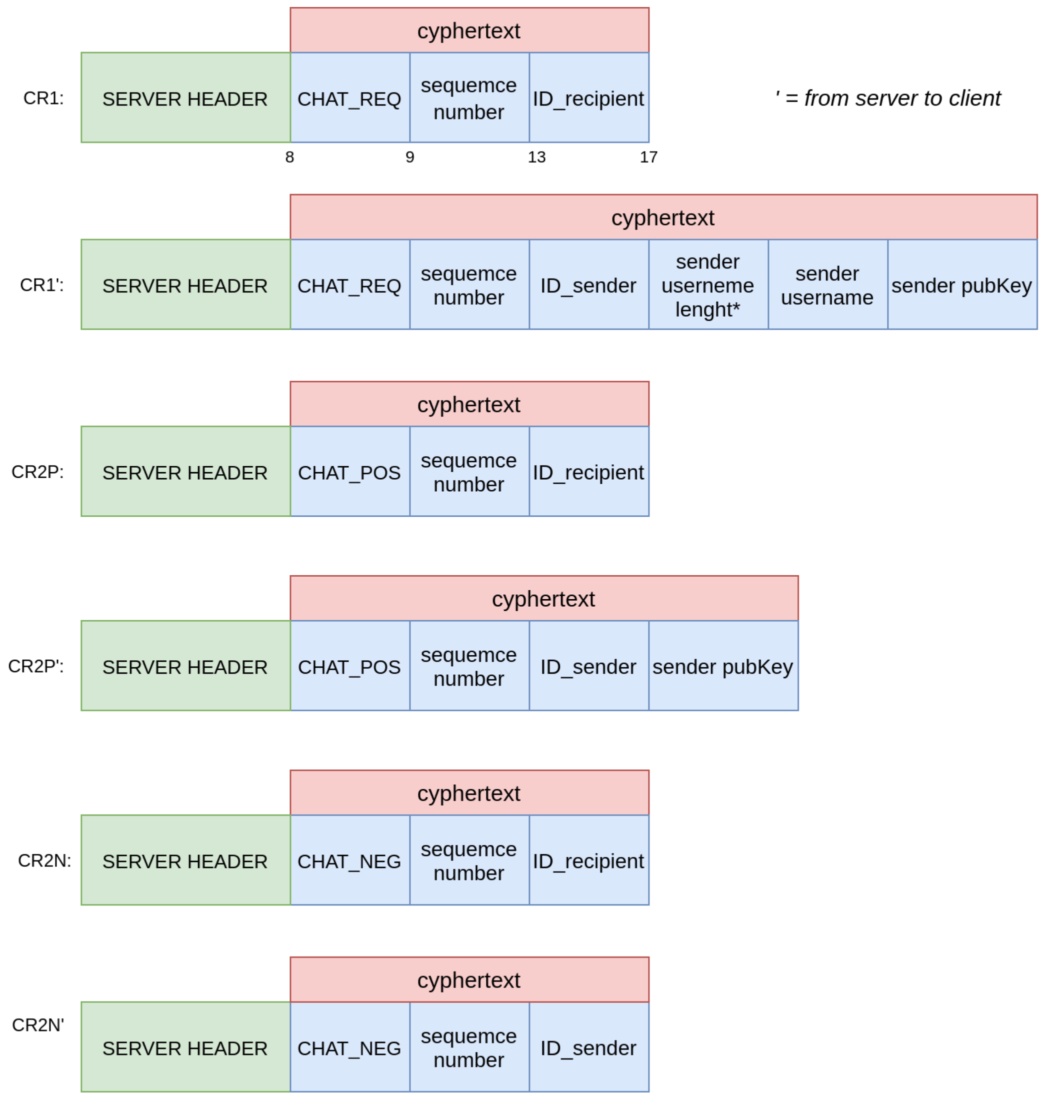


Figure 3.4: Chat Request Message Format

3.3 Chat

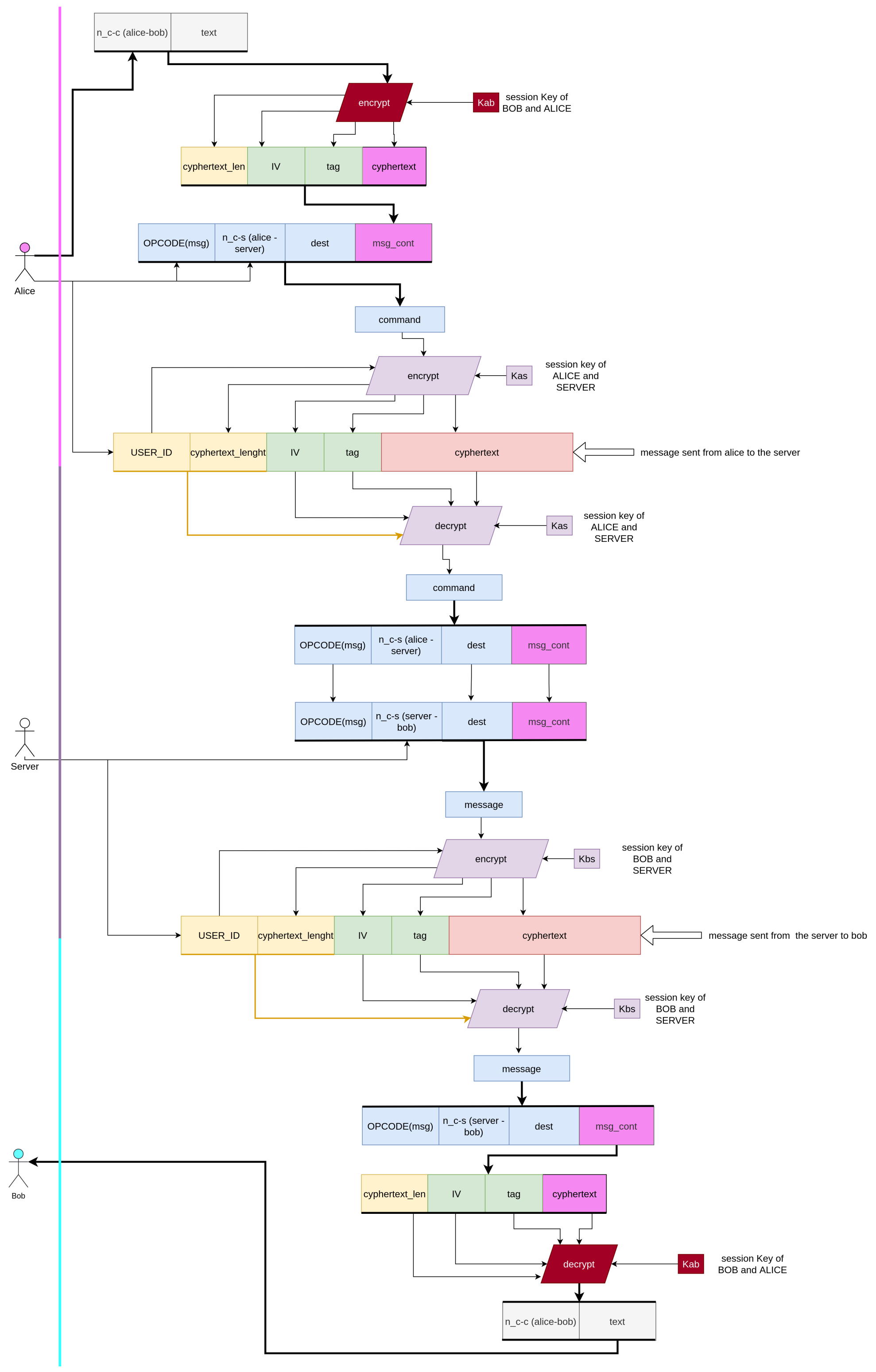


Figure 3.5: Client Client Handshake Protocol Schema

3.4 How To Handle The Chat Request

Let's start from the simple case in which a client wants to chat with another client and makes a request to the server:

- The standard channel is used to send either the request for chat and for the server's answers.
- If the target client accepts then the main process receive the confirmation from the server, then the main process set isChatting to true and it starts to chat.

When the server receive the command to chat with someone he has to send the chat request to the given client. Let suppose that Alice wants to chat with Bob. In this case the following steps must be performed:

- The request must be done through the request channel
- The client daemon process (Alice's daemon process) is listening on its socket (see after) and receive the request, then it reads the variable isChatting: if it is true the daemon process refuses automatically the server request, otherwise it ask (HANDSHAKE) to the main process if he wants to speak with Bob.
- The main process answer to the server by means of the daemon tools, if the answer is positive then it sets isChatting to true and so it waits for the message from Bob.

It's important to underline some concepts, first of all on the server side we have one process that handles two socket with the client, one for each channel. The protocol starts with the client that contacts the server, then the server and the client main process establish a connection and starts the protocol to establish a secure communication (Key Exchange). For standard messages this standard channel is used.

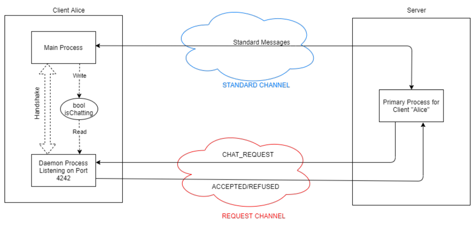


Figure 3.6: Protocol Schema

When the server has to sent a chat request to this client the request channel must be used. Thus it establishes a connection with the daemon process of the client that works as a server process and it is listening on port 4242 (see figure 3.6). The security of this secondary channel is ensured by the fact that the messages sent by the server to the client on the request channel are encrypted with the shared key established during the handshake in the standard channel, hence we can say that the authentication problem is not present. In any case to implement a greater security is possible to generate client side a one time password that is sent to the server in the encrypted session through the standard channel, then the server will sent this otp to the client daemon process to identify itself.
Other things to take in mind are:

- We assume that who send the chat request is the first to send messages.
- On the server side for each client two socket must be established, one for the standard channel and another one for the request channel but the process for each client is only one.