



UNIVERSITÀ DI PISA

LM in Cybersecurity

Applied Cryptography Project Report

FALL 2021

Professors: Gianluca Dini, ing. Michele la Manna

Students: Giuseppe Crea, Laura Norato, Marco Parti

Specifications and Design Choices

Users are already registered, and for this purpose 5 test users have been generated with RSA, through command line interface. Their private keys were generated with a password, which for test purposes is the same as the username.

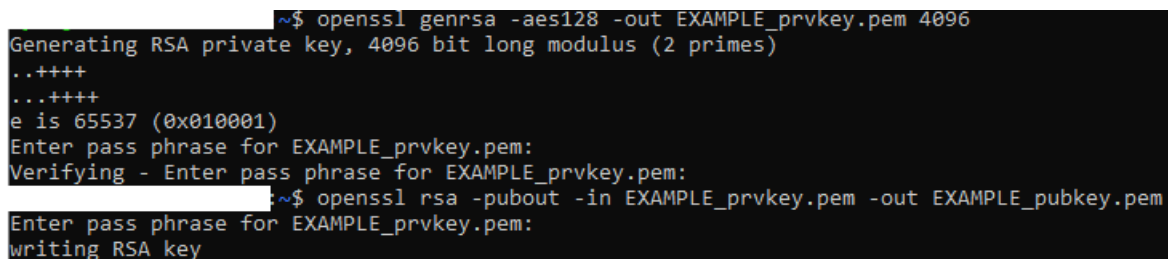
The command used for private key generation:

```
openssl genrsa -aes128 -out <clientname>_privkey.pem 4096
```

The one for public key generation:

```
openssl rsa -pubout -in <clientname>_privkey.pem -out <clientname>_pubkey.pem
```

Both commands ask for password when generating the key.



```
~$ openssl genrsa -aes128 -out EXAMPLE_privkey.pem 4096
Generating RSA private key, 4096 bit long modulus (2 primes)
...+++++
...+++++
e is 65537 (0x010001)
Enter pass phrase for EXAMPLE_privkey.pem:
Verifying - Enter pass phrase for EXAMPLE_privkey.pem:
~$ openssl rsa -pubout -in EXAMPLE_privkey.pem -out EXAMPLE_pubkey.pem
Enter pass phrase for EXAMPLE_privkey.pem:
writing RSA key
```

Server and clients must authenticate one another when connecting. The server will be authenticated by the public key included in its certificate, signed by a certification authority, while the clients will be authenticated through a copy of their own public keys, pre-installed on server. The problem of distributing the public keys of the clients and trusting the CA are considered solved, and outside the scope of the project.

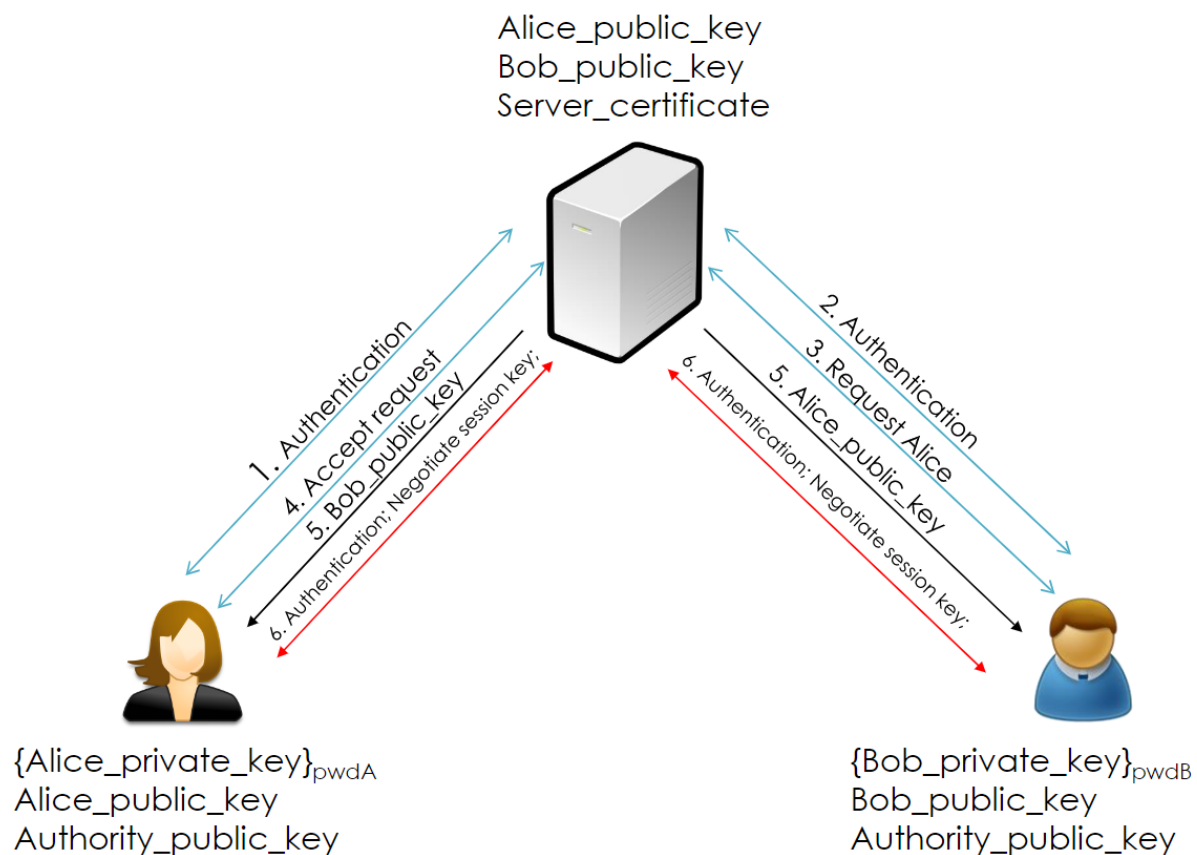
The CA and server certificates were generated through Simple Authority.

During authentication, a symmetric session key that provides perfect forward secrecy is negotiated through Elliptic Curve Diffie-Hellman. The curve was chosen beforehand and won't change between sessions.

After the login, users can obtain a list of other available (logged in and not busy in a chat session) users from the server through the "list" command.

The list only returns the names of "not busy" users due to our interpretation of the requirements. Users are expected to only have one chat session open at a time, and cannot start a second one without ending the first one. Despite the list, forwarding a chat request is checked server-side, because in-between the moment the list is received and the chat request is sent the target user might have disconnected or started another chat session.

Users can chat between each other with E2E authenticated encryption, but their messages move through the server. Clients receive their peer's public key only after the chat request has been accepted, and use that to negotiate a session key.



1: Application in the specification

Every message in the session is protected against replay attacks through the use of either a nonce (in authentication phase) or a directional counter (after a session key has been established), and all session-specific messages are encrypted with AES-GCM, both on a server-to-client and on a client-to-client level, effectively encapsulating one encrypted message between two clients (with key K_c) inside another encrypted message between client and server (with key K_s).

The directional counters begin at 0 each time, starting with the first symmetrically encrypted message, incrementing with each sent message. When the counter hits a threshold value which can be set at compile time (but realistically can be as high as $\text{INT_MAX} - 1$) the corresponding connection will be dropped, because the session key will need renegotiating. This is a way to protect against possible replay attacks within the same session. Using the same counter number across different sessions is not a problem, as the different session key will render the encryption totally different.

Message format

The following section will list the exact content of each message. A few notations before we begin: S → C means a message begins within the Server and reaches a Client.

$E_k[\text{message}]$ means a message has been symmetrically encrypted with key k “counter” refers to the directional counter between the two principals exchanging that message, with the same direction as the message. (A message from S to C will use the counter of messages sent to C from S on both principals).

The messages are listed by their message code.

The general message structure is made of an opCode which client and server will use to agree on an operation, a data field, and various control fields. These control fields change depending on whether the message is symmetrically encrypted or not. Some messages use an empty data field. All messages will have a size value, of length int32_t, as first element. This size value will contain the number of bytes in the whole message.

Non-symmetrically encrypted messages (those used during authentication) will have a negative size value, to distinguish them.

The underlined field guarantees freshness to the message. The field in italics guarantees the authenticity of the message.

The Messages

C->S: First_auth_msg_code (290)

Int32_t -size_tot, int32_t opCode, int32_t nonceC, unsigned char user_id

S->C: second_auth_msg_code (291)

Int32_t -size_tot, int32_t opCode, int32_t nonceS, long pem_size, unsigned char sv_dh_pubkey_pem, int32_t sign_size, unsigned char signature

signature = Sk_{sv_privkey}{int32_t nonceC, unsigned char sv_dh_pubkey_pem }

C->S: final_auth_msg_code (292)

Int32_t -size_tot, int32_t opCode, long pem_size, unsigned char cl_dh_pubkey_pem, unsigned int sign_size, unsigned char signature

signature = Sk_{cl_privkey}{int32_t nonceS, unsigned char cl_dh_pubkey_pem }

C->S: chat_request_code (301)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, unsigned char data:requested_user)

S->C: chat_request_received_code (301)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, unsigned char data:requesting_user)

C->S: chat_request_accept_code (303)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter)

S->C: chat_request_accept_code (303)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, long pem_size, unsigned char (pem_size) pem:public_key_partner)

S -> C: peer_public_key_msg_code (307)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, long pem_size, unsigned char (pem_size) pem:public_key_partner)

C -> S: chat_request_denied_code (304)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter)

S -> C: chat_request_denied_code (304)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter)

C1 -> S: nonce_msg_code (305)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, data:[(int32_t) nonceC1])

S -> C2: nonce_msg_code (305)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, data:[int32_t nonceC1])

C2 -> S: first_key_negotiation_code (308)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, data:[int32_t nonceC2, long pem_size, unsigned char (pem_size) c2_dh_pubkey_pem, unsigned int sign_size, unsigned char (sign_size) signature])

signature = Sk_c2_privkey(int32_t nonceC1, unsigned char c2_dh_pubkey_pem }

S -> C1: first_key_negotiation_code (308)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, data:[int32_t nonceC2, long pem_size, unsigned char (pem_size) c2_dh_pubkey_pem, unsigned int sign_size, unsigned char (sign_size) signature])

signature = Sk_c2_privkey(int32_t nonceC1, unsigned char c2_dh_pubkey_pem }

C1 -> S: second_key_negotiation_code (306)

Int32_t size_tot, unsigned char ct, unsigned char (16) tag, unsigned char (12) iv

ct = Eks(int32_t opCode, int32_t counter, data:[long pem_size, unsigned char (pem_size) c1_dh_pubkey_pem, unsigned int sign_size, unsigned char (sign_size) signature])

signature = Sk_c1_privkey(int32_t nonceC2, unsigned char c2_dh_pubkey_pem }

S -> C2: second_key_negotiation_code (306)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter, data:[long pem_size, unsigned char (pem_size) **c1_dh_pubkey_pem**, unsigned int sign_size, unsigned char (sign_size) **signature**]}

signature = Sk_{c1_privkey}{int32_t **nonceC2**, unsigned char **c2_dh_pubkey_pem** }

C -> S: end_chat_code (370)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter}

S -> C: closed_chat_code (371)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter}

C -> S: logout_code (372)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter}

S -> C: forced_logout_code (373)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter}

C -> S: list_request_code (374)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter}

S -> C: list_code (375)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter, data:[list[int32_t username_size, unsigned char (username_size) username]]}

C -> S: peer_message_code (350)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

ct = Eks{int32_t **opCode**, int32_t counter, data:[**peer_message**]}

peer_message = Int32_t size_tot, unsigned char **ct2**, unsigned char (16) *tag*, unsigned char (12) iv

ct2 = Eks{int32_t opCode, int32_t counterC, unsigned char message_data}

S -> C: peer_message_code (350)

Int32_t size_tot, unsigned char **ct**, unsigned char (16) *tag*, unsigned char (12) iv

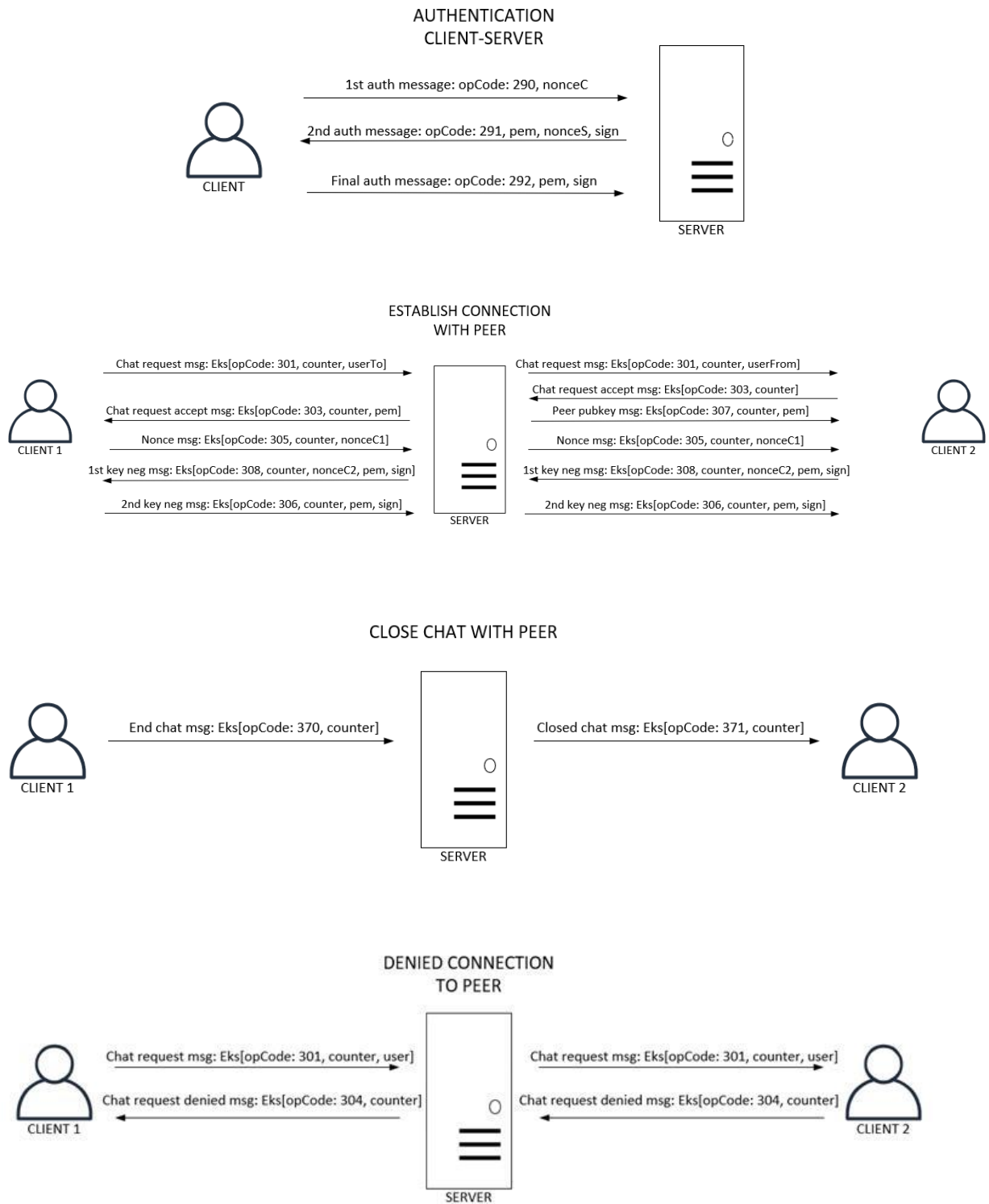
ct = Eks{int32_t **opCode**, int32_t counter, data:[**peer_message**]}

`peer_message = Int32_t size_tot, unsigned char ct2, unsigned char (16) tag, unsigned char (12) iv`

`ct2 = Ekc(int32_t opCode, int32_t counterC, unsigned char message_data)`

The protocol

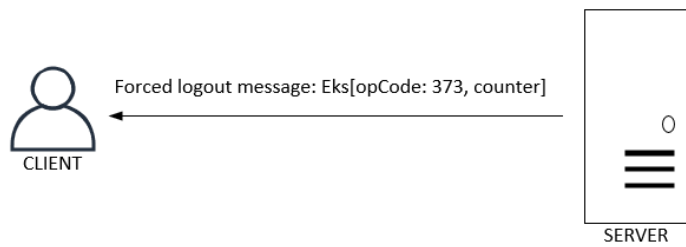
Using the opCode values reported above, we can describe the protocol in the following diagrams.



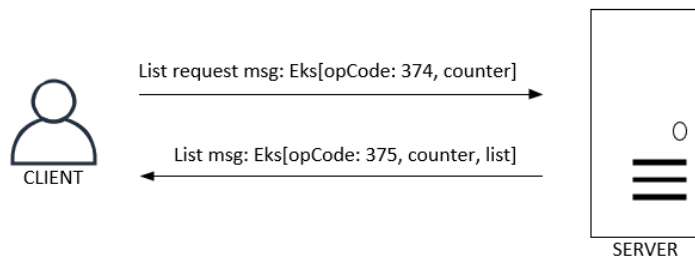
CLIENT LOGOUT



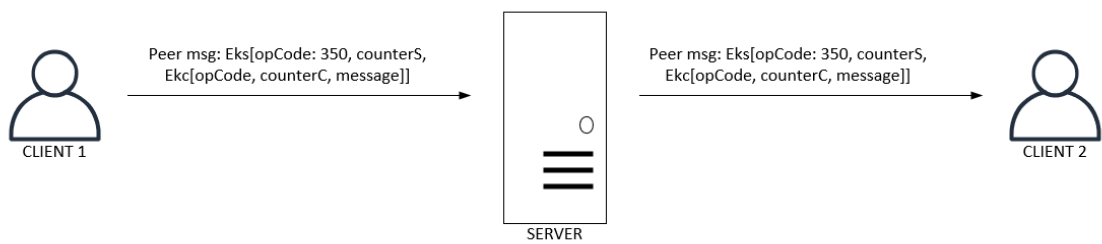
FORCED
CLIENT LOGOUT



LIST REQUEST



PEER MESSAGE
EXCHANGE

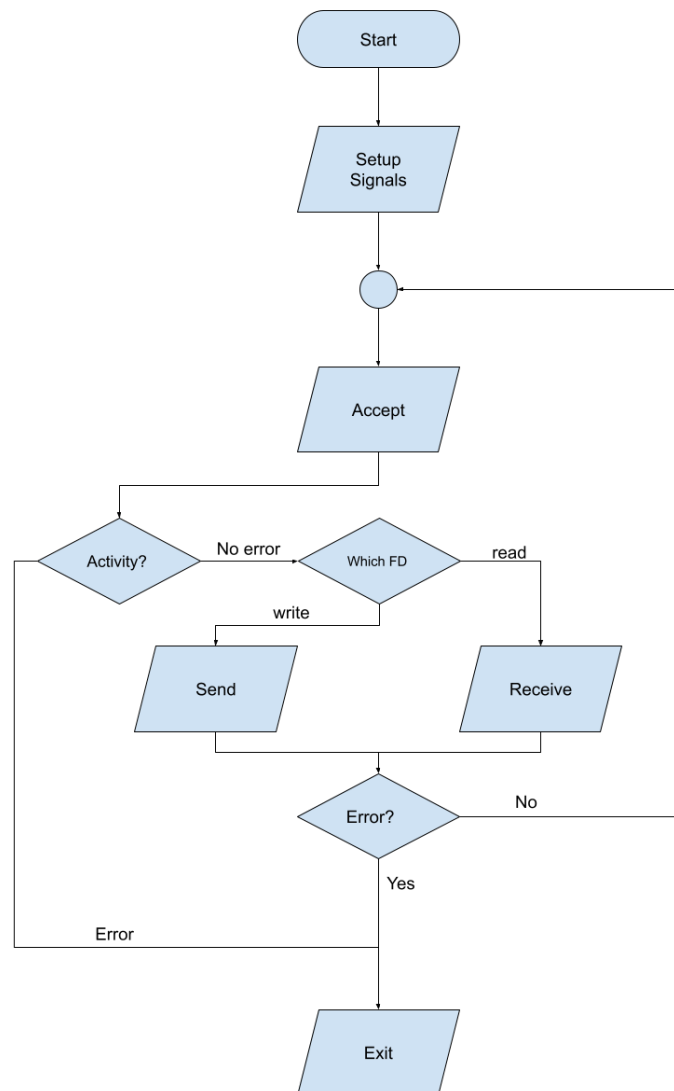


Workings of Server and Client

Both server and client use non-blocking IO to handle multiple input within a single thread. Both send asynchronous messages, from a queue.

The server

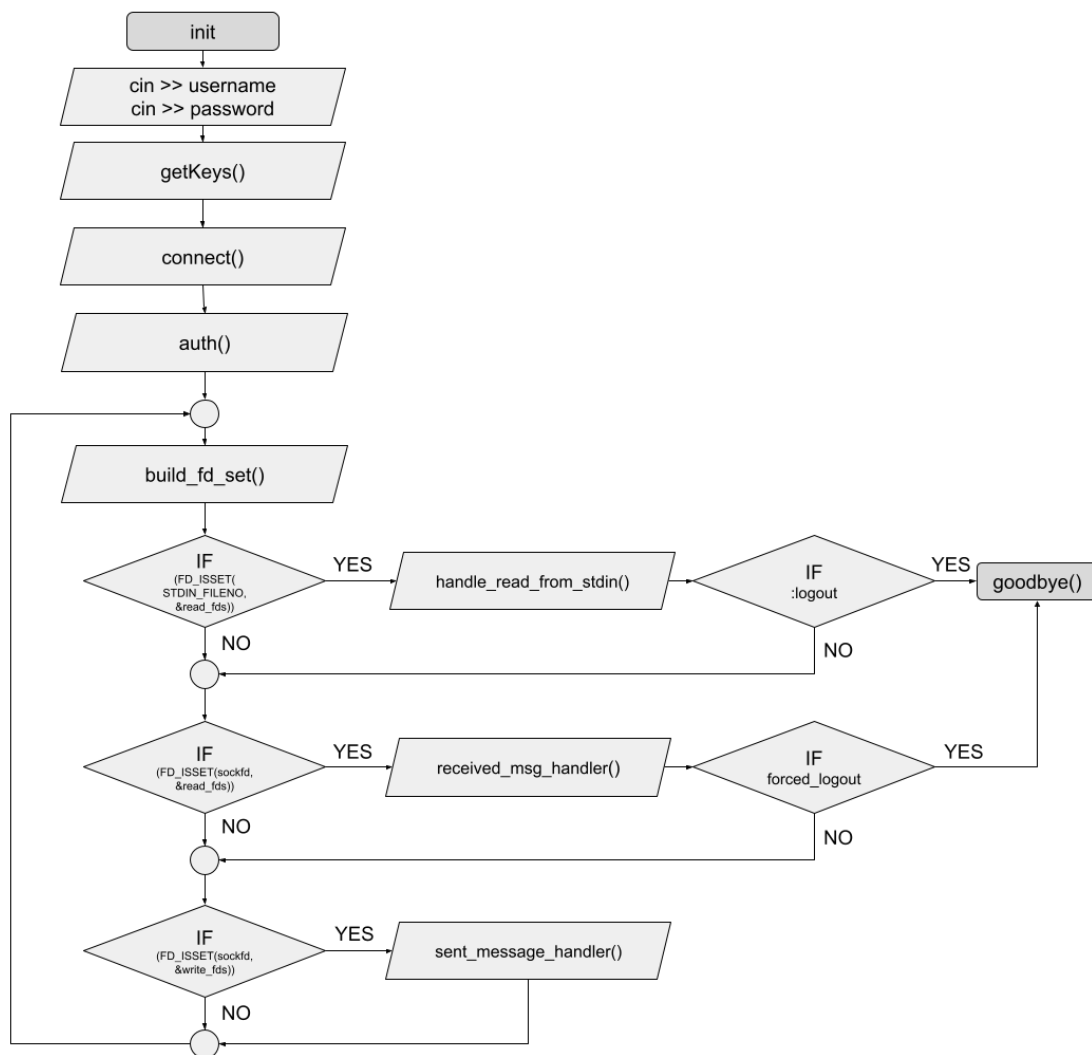
The server uses two maps of pointers to the same ClientElement Objects, one indexed by socket id and another indexed by user name (once the user name becomes available as the authentication happens). It will handle any message it receives independently from all others thanks to their opCode, using the ClientElement object to store any variables it might need from one message to the next, such as the directional counters or the session key. On any unrecoverable transmission or encryption error, the server will clean up the client and close the connection with them.



2: Server main flow diagram

The flow diagram on the side shows how the server's main function works.

The client



3:Client main flow diagram

Analogously to the Server, the Client has a cleanup function called “goodbye”, which frees all allocated memory safely.

The “getKeys” function loads both the client’s private and public keys, from PEM files stored on disk. It also prompts the user for their cert’s password.

“auth” handles all aspects of first authentication and key negotiation between client and server.

1. Computes nonce
2. Sends nonce and username to server in 290
3. Waits for a response with code 291
4. Reads the server nonce, the DH-pub-key of server, the signature over the pair (dh-pub-key of server, nonce of user) from server, and the Server CERT.
5. Extracts public key from the server CERT, checking its validity against the CA.
6. Checks the signature for validity.
7. Computes its own DH parameters and keys, both public and private.
8. Sends the public DH key back to the server, signed together with the server nonce, with opcode 292.
9. Generates the session key from the ECDHKE.

Once inside its main loop the Client will keep listening for messages from the Server, and inputs from STDIN.

Focus on encryption and signature functions

Encryption

AES in Galois-Counter-Mode encryption is used to ensure authentication of messages and confidentiality.

Gcm encryption

```
int gcm_encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *aad,
int aad_len, unsigned char *key, unsigned char *iv, int iv_len, unsigned char
*ciphertext, unsigned char *tag)
```

The encryption function returns the size of the ciphertext or a negative value in case of error. AAD and AAD_LEN fields are set to NULL and 0 respectively, as we don't use them.

Gcm decryption

```
int gcm_decrypt(unsigned char *ciphertext, int ciphertext_len, unsigned char *aad,
int aad_len, unsigned char *tag, unsigned char *key, unsigned char *iv, int iv_len,
unsigned char *plaintext)
```

As the encryption function, this function returns the size of the plaintext or a negative value in case of error.

Signature

Signature

```
bool signature(EVP_PKEY* cl_pr_key, unsigned char* pt, unsigned char** sign, int
length, unsigned int* sign_size)
```

The signature function sets its return value to false when an error occurs, otherwise returns true and places the signature inside the parameter `unsigned char** sign`.

Verify Signature

```
bool verify_sign(EVP_PKEY* pub_key, unsigned char* data, int n, long data_dim,
unsigned char* sign, int sign_dim)
```

This function verifies the validity of a signature and returns false in case of failure or errors.