**Architecture:**

The software architecture that we are developing is based on a client-server. That is, the messages are first sent to the server, and then the server forwards the appropriate message to each client.

The client software will "pull" messages from the server cyclically (pull request.)

This feature allows messages to be sent to clients in both online and offline modes.

In addition, we will support end-to-end encryption, meaning that the messages are encrypted on the client side and sent in an encrypted form, so that no one can decrypt the information except the target client (not even the server).

**Server:**

The server's role is to manage the list of users registered in the service and allow them to exchange messages of various types with each other.

A. The server will be written in the Python language.

B. The server will support a stateless protocol, meaning it will not store information between requests (each request stands on its own).

c. The server will support multiple users through threads or selectors.

d. The server version will be 1.

**Port**

The server will read the port number from a text file in the following way:

- File name: myport.info

- File location: In the same folder as the server's code files

- File content: Port number

For example: 1234

- If the file does not exist, a warning message should be displayed

and the default port 1357 should be used

**Data**

The data will be saved using SQL tables in a file called defensive.sb.

**Note:** At the beginning of the server's run, if the file does not exist, it must be created.

Similarly, if a table does not exist, it must be created.

Information about customers will be saved in a table called clients

The table structure:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Notes** |
| ID | 16 bytes (128 bits) | A unique identifier for each customer. Index |
| UserName | String (255 bytes) | An ASCII string representing a username. Including a terminating character! |
| LastSeen | Date and hour | The time when the last request was received from a client. |

Information about messages will be saved in a table called messages

The table structure:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Notes** |
| ID | 4 bytes | Index |
| ToClient | 16 bytes | Unique recipient identifier |
| FromClient | 16 bytes | Unique sender identifier |
| Type | byte | Message type |
| Content | Blob | Message content |

**How the server works**

1. Reads the port from the file myport.info

2. Waits for requests from clients in an infinite loop

3. When receiving a request, the request is decoded according to the protocol:

A. Registration request: If the requested username already exists, the server will return an error.

Otherwise, the server will generate a new UUID for the user, save the data in the database, and

return a success response.

B. Request for a list of customers: The server will return the list of customers according to the

protocol.

C. A request to send a message will be handled as follows:

The server will extract the message type and the content of the message (from the payload) and

save it in memory (or in the database).

D. A request to retrieve pending messages will be handled as follows:

The server will retrieve messages from the memory (or database) and send a response to the

client.

Note! Messages that have been successfully sent to the client will be deleted.

**Client:**

The client software will allow the user to send a message to another user who is registered on the server.

Note! A message can be sent to any registered user in the system (there are no "contacts").

A. The client software will be written in C++

B. The client will run in console mode and receive

input from the user (stdin or cin) to perform

the various operations.

C. The client will support end-to-end encryption

D. The client version will be 1

**Server address and port**

The client will read the server address and port from a text file in the following way:

- File name: server.info

- File location: In the folder of the executable file (exe.)

- File content: IP address + colon + port number. For example: 127.0.0.1:1234

**Name and Unique Identifier (UUID)**

The client will save and read its name and unique identifier from a text file in the following format:

- File name: my.info

- File location: In the folder of the executable file (.exe)

- File content:

First line: Name

Second line: Unique identifier in ASCII representation where

Every two characters represent an 8-bit hex value.

Third line: Private key generated in the first run of the program in base 64 format

For example:

|  |
| --- |
| Michael Jackson  64f3f63985f04beb81a0e43321880182  MIGdMA0GCSqGSIb3DQEBA... |

**Data**

The client will store the client data (unique identifier, name, public key and symmetric key) in memory (RAM)

**How the client works**

The client will present the user with the following menu and wait for input from the user in terminal mode (stdin) in an infinite loop.

|  |
| --- |
| MessageU client at your service.  110) Register  120) Request for clients list  130) Request for public key  140) Request for waiting messages  150) Send a text message  151) Send a request for symmetric key  152) Send your symmetric key  0) Exit client |

**Error on the server side**

In any case of an error, the client will print a message to the screen: "server responded with an error" and wait for the next input.

**Possible actions:**

**Registration request** – input "110"

1. The client will receive a username from the terminal and send

a registration request to the server.

2. The client will save the

name and unique identifier it receives from the server in a file called me.info.

Note! If the file already exists, the client will not

allow the registration request and will print an error to the terminal.

**Request for client list** – input "120 "

1. The client will send a request for a client list to the server.

will decode the response and print the names of the list of clients

**Public key request** – input "130 "

1. The client will send a public key request to the server.

**Request for pending messages** – input "140"

1. The client will send a request for pending messages to the server. It will decode the response and print the messages to the screen in the following form:

|  |
| --- |
| From: <user name>  Content:  <content>  .  .  -----<EOM>-----  \n |

1. For the message type "Symmetric key request" you must write in the message content:

"Request for symmetric key"

1. For the message type "Sending symmetric key" you must write in the message content:

"Symmetric key received". In addition, the client software will store in memory the symmetric key for this client.

1. For the message type "Send text message" the message must be decrypted using the

symmetric key and the message displayed. If there is no symmetric key or it is invalid, write "can't decrypt message".

**Request to send a message** – input "150"

1. The client will wait for the target username to be received from the terminal

2. The client will wait for the text message to be received

3. The client will send a "send message" request of type "send text message" to the server.

**Request to receive a symmetric key** – input "151"

1. The client will wait for the target username to be received from the terminal

2. The client will send a "send message" request of type symmetric key.

**Request to send a symmetric key** – input "152"

1. The client will wait for the target username to be received from the terminal

2. The client will generate a symmetric key. It will store it in memory for the target client and

send a "send message" request of type symmetric key send.

**Exit** – input "0"

1. The client will release system resources and terminate its execution.

**Communication protocol**

Note! The described protocol is a basic implementation of end-to-end encryption.

**General**

- The protocol is binary and implemented over TCP.

- All numeric fields must have values ​​greater than zero (unsigned) and are represented as little

endian

- This protocol supports requests to the server and responses to the client. Requests or responses can contain "message".

- A message is passed between clients

**Remember!** The protocol is binding and cannot be changed. As a result, any server and client that implements the protocol can work with each other.

**Registering with the system**

1. Each client that connects for the first time registers with the service with a name (a string with a maximum length of 255 bytes) and passes its public key
2. The server will return to the client a unique identifier created for it or an error if the name already exists in the database.

**Requests from the server**

1. A client can request the list of other users

2. A client can request a public key of a particular client

3. A client can request all messages waiting for it

4. A client can send a message to another client

**Message exchange**

1. Client A asks the server for the public key of client B

2. Client A sends a message (via the server) to client B of the type "Symmetric encryption key

request" The message is encrypted by B's public key

3. The server receives the message and stores it

4. Client B pulls the messages waiting for it from the server

5. Client B decrypts the message using the private key

6. Client B asks the server for the public key of client A

7. Client B sends a reply of the type "Symmetric encryption key" to client A

The reply is encrypted by A's public key

8. The server receives the message and stores it

9. Client A pulls from the server the messages waiting for it

10. Client A decrypts the message using the private key

11. Now, Client A and Client B can communicate using a symmetric encryption key

**Protocol details**

**Requests**

Request structure from the client to the server. The server will decrypt the content (payload) according to the request code.

**Request to the server**

|  |  |  |  |
| --- | --- | --- | --- |
| **Request** | **Field** | **Size** | **Meaning** |
| **Header** | Client ID | 16 bytes (128 bits) | A unique identifier for each customer |
| Version | byte | Client version number |
| Code | 2 bytes | Request code |
| Payload size | 4 bytes | Request content size |
| **payload** | payload | Variable | The content of the request varies depending on the request. |

**Content (payload)**

The content varies depending on the request. Each request has a different structure.

**Request code 600** – Registration

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Name | 255 bytes | An ASCII string representing a username. Including a terminating character! |
| Public Key | 160 bytes | Client public key |

**Note:** The server will ignore the Client ID field

**Request code 601** – User list

The payload field does not exist. Payload field size=0

**Request code 602** – Retrieve client public key

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Client ID | 16 bytes | Unique client identifier |

**Request code 603** – Send message to client

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Client ID | 16 bytes | Unique identifier of the target client |
| Message Type | byte | Client message type |
| Content Size | 4 bytes | Message content size |
| Message Content | Variable | The content of the message. Encrypted by the target client's public key or by a symmetric key. Depending on the type of message. |

**Request code 604** – Retrieve pending messages

The payload field does not exist. Payload size=0

**Message Type to Client (Message Type)**

A client can send various messages to another client.

**Message Type 1** – Symmetric Key Request

The message content is empty. Content Size=0

**Message Type 2** – Symmetric Key Send

The Message Content field contains a symmetric key encrypted by the target client's public key

**Message Type 3** – Text Message Send

The Message Content field contains text encrypted by a symmetric key.

**Responses**

**Server response**

|  |  |  |  |
| --- | --- | --- | --- |
| **Response** | **Field** | **Size** | **Meaning** |
| **Header** | Version | byte | Server version number |
| Code | 2 bytes | Response code |
| Payload size | 4 bytes | Response content size |
| **payload** | payload | Variable | The content of the response. Varies depending on the response. |

**Content (payload)**

**Response Code 2100** – Registration Successful

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Client ID | 16 bytes | Unique client identifier |

**Response Code 2101** – User List

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Client ID | 16 bytes | Unique client identifier |
| Client Name | 255 bytes | An ASCII string representing a username. Including a terminating character! |

**Important:** A user list will not include the user who requested it. Also, the list may include many users. They will appear one after the other and their number can be calculated using the formula:

Payload Size / (16+255)

**Response code 2102** – Public key

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Client ID | 16 bytes | Unique client identifier |
| Public Key | 160 bytes | Client public key |

**Response code 2103** – Message to client sent (stored on server – not necessarily read)

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Client ID | 16 bytes | Unique identifier of the target client |
| Message ID | 4 bytes | Unique message identifier |

**Response code 2104** – Retrieving pending messages

|  |  |  |
| --- | --- | --- |
| **Field** | **Size** | **Meaning** |
| Client ID | 16 bytes | A unique identifier of the client from which the message came. |
| Message ID | 4 bytes | Unique message identifier. |
| Message Type | byte | The type of message to the client. |
| Message Size | 4 bytes | Message size |
| Content | Variable | Message content |

**Important:** There can be many pending messages. They will appear one after the other in sequence

**Response code 9000** – General error

Payload field does not exist. Payload size field=0

**Encryption**

The communication protocol uses symmetric encryption to encode the message between clients and asymmetric encryption to exchange keys between clients.

In this exercise, use the Crypto++ library, recommended version: 8.80

**Symmetric encryption**

For symmetric encryption, use AES-CBC.

The key length is 128 bits. You can assume The IV is always zeroed (the memory is full of zeros).

Such use of IV is not secure if the same key is used every time, but for the purpose of this assessment it is sufficient.

**Asymmetric encryption**

For asymmetric encryption use RSA.

The key length is 1024 bits.

**Note:** The Crypto++ library holds public keys in X509 format. This format contains a Header before the key itself and additional values. Therefore, its final size (in binary form) is 160 bytes (for keys of different sizes the final size of the key will change accordingly).

**Development Highlights**

1. It is recommended to work with a code management system (such as Git)

2. Work modularly and test yourself all the time

a. Identify the important classes and functions

b. **On the server side:**

Write code to handle a single request. Add multi-client support later

c. **On the client side:**

Implement the large components independently in other parts of the system (communication,

encryption, protocol, etc.).

3. Implement code for testing early in the project

a. **On the server side:**

Use screen printing or logging to track communication. You can also load the module into the

interpreter and work dynamically.

b. **On the client side:**

Write small functions that test separate parts of the system. Use these functions while writing

the code itself.

4. Writing the code

a. Implement classes and use object-oriented programming principles, i.e. use

encapsulation, inheritance and polymorphism

b. Pay attention to the representation of values ​​in memory as little-endian or big-endian

c. Be sure to document the code (comments)

d. Give meaningful names to variables, functions and classes. Avoid magic numbers!

e. A message can be very large (dynamically sized). Think about the best way to receive

and send a large amount of information.

5. **Error handling** will be done using exceptions

a. Exceptions should be thrown in appropriate places, caught and handled accordingly.

b. If it is not possible to continue, a user-friendly message should be displayed and an orderly exit

should be made

(for example, when trying to read a non-existent file, a friendly and orderly message should be

displayed and an error should be avoided).

Note! This is true for C++ and Python.

6. **Information security** – Think throughout about writing secure code according to the principles you

learned: Have you checked the input? How is dynamic memory used? Is type conversion (casting)

performed? etc..

**Emphasis for server code:**

1. Use Python version 3

2. Use only standard Python libraries!

3. You can use the struct library to work with communication data comfortably

**Emphasis for client code:**

1. Use object-oriented programming principles (i.e. use classes, inheritance

and polymorphism)

2. It is recommended (but not mandatory) to use STL libraries

3. It is possible and desirable to use C++11 capabilities (for example, functions of the lambda type,

use of auto, etc.).

4. Use boost library to implement communication

**Crypto++ encryption library**

The assessment already contains the important functionality of crypto. You can find in attached files.