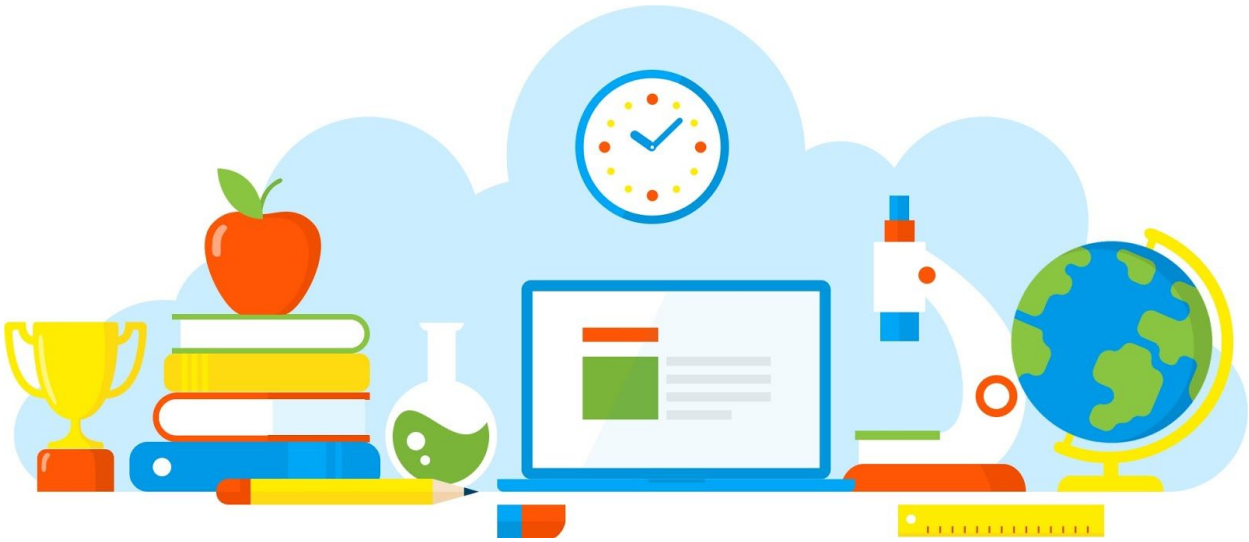

Internet Technology

Ruikang Xu 463255

Artem Avetyan 464761

Personal reflection

For this project, to be honest it is the biggest project I've done before in my life, it consumed so much time and effort, and I'm happy to see this project is done now.

This assignment should have been done by two, but my teammate Artem and me choose to upload it separately because of at the very beginning we both delivered our own structure and we developed basing on our own project for next two levels. It doesn't mean we have no communication or link, we still talk about the technical details and other problems, and I personally pretty enjoy the way we deal with it. Actually from 1st year, we have been working together on several projects, he is capable and smart, so turning out that even we know that our structure is totally different, we would still discuss and share about protocol and ideas, we have confidence in ourselves.

The following shortenings are used in this document and in the program code of the client and server implementations:

- message — msg,
- command — cmd.

Protocol

The server and client programs send each other text in the UTF-8 format. This text is split into *lines* ending with a terminating sequence U+000D, U+000A (also known as CR, LF). This is in accordance to RFC 5198 “Unicode Format for Network Interchange.” The server also may end its lines with U+000A, so this protocol is compatible with the provided server executable which ends its lines with different characters depending on the OS where it's running. The content of a line without the terminating sequence is a *protocol message*. Protocol messages will be shown each on a separate line in a fixed-width font.

The following placeholders will be used:

- <user> — user name,
- <group> — group name,
- <msg> — user message.

A user name or a group name is *valid* if and only if it contains only alphabetic characters, digits (according to Java's Character class), or underscores.

In various parts of this protocol, there is a need to transform binary data into text because the protocol is text-based. The encoding used to this end is the basic variant of the Base64 without padding according to RFC 4648 and RFC 2045.

Handshake

Initially, after the client connected to the server, the server sends

HELO <welcome msg>

where <welcome msg> is a free-from welcome message. If <welcome msg> has the form

LEVEL <level> <text>

where <level> is an integer number, then the server supports the level <level> of the

protocol. The client replies with

HELO <user>

where <user> is the client's user name. The server replies with

+OK HELO <user>

if the server accepted the client or with one of

-ERR user already logged in

-ERR user has an invalid format (only characters, numbers and underscores are allowed)

if it does not. If the server accepted the client, they may exchange messages described below.

The server explicitly sends the protocol level so the client will not send messages that the server does not support. The behavior of the server when it receives an unsupported message is undefined.

Sending a broadcast message

If the client's user <user> wants to send a message <msg> to all users currently logged into the server, the client sends

BCST <msg>

The server replies with

+OK BCST <msg>

and sends

BCST <user> <msg>

to all other clients.

Sending a private message

If the client's user <sender user> wants to send a private message <msg> to another

user <recipient user>, the client sends

PM <recipient user> <packet>

The server replies with

-ERR no such user

if it has not accepted the message or with

+OK PM

if it has. If the server accepted the message, it sends

PM <sender user> <packet>

to <recipient user>'s client.

The text <packet> has the form

<crypto> <data>

where <crypto> is PLAIN or CRYPTO. If <crypto> is PLAIN, <data> is <msg>, so the message is unencrypted. If <crypto> is CRYPTO, see the section “Using encryption.”

Reading the list of users

The client sends

GET_USERS

The server replies with

+OK GET_USERS <user list>

where <user list> is the list of names of the users logged into the server excluding the name of the client's user, separated by the space character.

Groups

A group is a set of users. Groups are stored on the server. Members of a group can

exchange messages not visible to people outside that group.

The administrator of a group is the user who created it. The administrator of a group can dismiss the group and prevent users from joining the group.

The list of groups, their members, and administrators is independent of the list of users logged into the server. So a user who logged out remains in the groups and keeps the role of administrator.

Reading the list of groups

The client sends

GET_GROUPS

The server replies with

+OK GET_GROUPS <group list>

where <group list> is the list of names of groups on the server, separated by the space character.

Creating a group

If the client's user wants to create a group <group>, the client sends

CREATE_GROUP <group>

The server replies with

+OK CREATE_GROUP <group>

if the group has been created or with one of

-ERR group "<group>" already exists

-ERR group name has invalid format

otherwise.

Dismissing a group

If the client's user wants to dismiss (delete) a group <group>, the client sends

DISMISS_GROUP <group>

The server replies with

+OK DISMISS_GROUP <group>

if the group has been dismissed or with one of

-ERR no such group

-ERR you are not the administrator of group "<group>"

Only the administrator of <group> can dismiss <group>.

A user joining a group

If the client's user wants to join a group <group>, the client sends

JOIN <group>

The server replies with

+OK JOIN <group>

if the user has joined the group or with one of

-ERR no such group

-ERR you were kicked from group "<group>"

-ERR you are already in group "<group>"

otherwise.

A user leaving a group

If the client's user wants to leave a group <group>, the client sends

LEAVE <group>

The server replies with

+OK LEAVE <group>

if the user has left the group, or with one of

-ERR no such group

-ERR you are not in group "<group>"

otherwise.

Kicking a user from a group

If the client's user wants to kick a user <user> from a group <group>, the client sends

KICK <group> <user>

The server replies with

+OK KICK <group> <user>

if <user> was kicked from the group, or with one of

-ERR no such group

-ERR no such user

-ERR user "<user>" is not in group "<group>"

-ERR you are not the administrator of group "<group>"

If <user> has been kicked, the server sends

KICKED <group>

to <user>'s client. Only the administrator of <group> can kick from <group>. Any user <user> who was once kicked from a group <group> cannot join <group>. However, if <group> was dismissed, and a new group with the same name was created, <user> can

join the new group.

Sending a message to a group

If the client's user <user> wants to send a message <msg> to a group <group>, the client sends

GROUP_MESSAGE <group> <msg>

The server replies with

+OK GROUP_MESSAGE <group> <user> <msg>

if the server accepted the message or with

-ERR no such group

-ERR you are not in group "<group>"

otherwise. The servers sends to the clients of the other users in the group who are logged in

GROUP_MESSAGE <group> <user> <msg>

Sending a file

If the client's user <sender user> wants to send a file to a user <recipient user>, the client sends to the server

FILE_OFFER <recipient user> <params>

where <params> contains transmission and file parameters, see below. The server replies with

+OK FILE_OFFER

if it approved the offer or with

-ERR no such user

otherwise. If the server approved the offer, it sends to <recipient user>'s client

FILE_OFFER <sender user> <params>

If <recipient user> wants to receive the file, their client sends to the server

FILE_ACCEPT <sender user>

The server replies to <recipient user>'s client with

+OK FILE_ACCEPT

if it approved the acceptance or with

-ERR no such user

otherwise. If the server approved the acceptance, it sends to <sender user>'s client

FILE_ACCEPT <recipient user>

When <sender user>'s client receives the protocol message FILE_ACCEPT, it splits the file offered to <recipient user> into parts and sends every part to the server as protocol messages

FILE_DATA <recipient user> <data>

The text <data> is the part of the file transformed (for example, encrypted) as described below.

To every such message, the server replies with

+OK FILE_DATA

if it accepts the message or with

-ERR no such user

otherwise. If the server accepted the message, it sends to <recipient user>'s client

FILE_DATA <sender user> <data>

When all parts are sent, <sender user>'s client sends to the server

FILE_END <recipient user>

The server replies with

+OK FILE_END

if it accepts the message or with

-ERR no such user

otherwise. If the server accepted the message, it sends to <recipient user>'s client

FILE_END <sender user>

Therefore, at most one file transmission is allowed between any pair of users at any moment.

The text <params> has the form

<file length> <crypto>

where <file length> is the length of the file in bytes and <crypto> is PLAIN or CRYPTO. If <crypto> is PLAIN, then all <data> are Base64-encoded file parts. If <crypto> is CRYPTO, see the section “Using encryption.”

The FILE_END message is required for sending an encrypted file. Although <params> contains the unencrypted file length, it is hard to determine the encrypted file length from the unencrypted file length, so <recipient user>'s client cannot determine when the transmission has ended.

Using encryption

The protocol allows any pair of users to exchange encrypted information such that neither the server nor other users are able to decrypt it. The protocol uses Advanced Encryption Standard (AES) block cipher with a key size of 128 bits and Galois/Counter Mode (GCM) of operation. Encrypted data should be Base64-encoded as text before inserting them into protocol messages.

Sending a private message

If <crypto> in <packet> is CRYPTO, <data> is the Base64-encoded concatenation of the initial vector used during encryption and the encrypted <msg>. An initial vector must be randomly generated for every private message.

Sending a file

If <crypto> in <params> is CRYPTO, then <data> is

- the Base64-encoded concatenation of the initial vector used during encryption and the encrypted part of the file if the first part is transmitted,
- or the Base64-encoded encrypted part of the file otherwise.

Client implementation

User interface

The client can be started by executing the class chat_client.Main with arguments:

- chat server host name,
- chat server port number,
- <user> — the name of the user controlling the client.

The client connects to the server and registers itself as the user <user>. Then the user can control the client by typing commands and messages into the console. Commands start with a semicolon. Command names are case-insensitive. Empty lines typed by the user into the client are ignored. Error and informational messages from the client to the user start with an exclamation mark.

Terminating

If the user closes the standard input (presses Control-D) or types

:q

then the client terminates.

Reading the list of users

The user types

:us

The client shows the list of users logged into the server excluding the name of the user.

Groups

Reading the list of groups

The user types

:gs

The client shows the groups on the server.

Creating a group

If the user wants to create a group <group>, the user types

:gcreate <group>

The user joins the group automatically. The client shows the server's reply.

Dismissing a group

If the user wants to dismiss a group <group>, the user types

:gdismiss <group>

The client shows the server's reply.

A user joining a group

If the user wants to join a group <group>, the user types

:join <group>

The client shows the server's reply.

A user leaving a group

If the user wants to leave a group <group>, the user types

:leave <group>

The client shows the server's reply.

Kicking a user from a group

If the user wants to kick a user <user> from a group <group>, the user types

:kick <group> <user>

The client shows the server's reply.

If the user was kicked from a group, the client informs the user.

User messages and groups

The user can choose the current group and send a message to the current group. The current group name is stored in the client. A special *common* group encompasses all users. When the user sends a message to the common group, the client sends a broadcast message. The user can make the group current even if the group does not exists on the server.

Setting the current group

If the user wants to make the group <group> current, the user types

:c <group>

If the user wants to make the common group current, the user types

:c

Sending a message

If the user <user> wants to send a message <msg> to the current group, the user types

<msg>

The message <msg> must not start with a semicolon. If the current group is the common group, other users will see the message as follows:

<user>: <msg>

If the current group <group> is not the common group, other users will see the message as follows:

<group>/<user>: <msg>

If the server did not accept the message, the client shows the server's reply.

Using encryption

The client uses the Java Cryptography Extension (JCE) for cryptographic operations. A key is exported from the client or imported into the client as a text encoded with the basic variant of the Base64 encoding without padding according to RFC 4648 and RFC 2045.

If a user <user a> wants to send encrypted information to a user <user b>, one of these users needs to generate a key (this can be done in the client) and send the key to the other user via a confidential channel besides the chat server. Then <user a> needs to assign the key to <user b> in their client, and <user b> needs to assign the key to <user a> in their client.

When the client assigns a new key to a user, the client tries to securely destroy the key assigned to that user before (if it existed). If JCE does not support this, the client will show the message “failed to securely erase existing key material.”

Generating a key for a user

If the user types

:kgen <user>

the client assigns a randomly generated key to <user> and shows the key in the format described above.

Assigning a key to a user

If the user types

:kassign <user> <key>

the client assigns <key> typed in the format described above to <user>.

Clearing the key for a user

If the user types

:kclear <user>

the client removes the key from <user>.

Sending a private message

If the user <sender user> wants to send a private message <msg> to another user <recipient user>, <sender user> types

:pm <recipient user> <msg>

If <sender user> did not assign a key to <recipient user>, the <sender user>'s client will transmit <msg> unencrypted, and <recipient user> will see the message as follows:

%<sender user>: <msg>

If <sender user> assigned a key <key a> to <recipient user>, then <msg> will be encrypted with <key a> into <ciphertext> and sent to <recipient user> via the server.

If <recipient user> assigned a key <key b> to <sender user>, then <ciphertext> will be decrypted with <key b> to <msg b> and shown to <recipient user> as follows:

%#<user>: <msg b>

If <recipient user> did not assign a key to <sender user>, <recipient user> will see an error message.

Obviously, <key a> should be equal to <key b> if <sender user> wants <recipient user> to

see <msg>.

Sending a file

If the user <sender user> wants to send a file to another user <recipient user>, <sender user> types

:foffer <recipient user> <path>

where <path> is the file system path referring to the file. If <sender user> assigned a key to <recipient user>, their client will do an encrypted transmission of the file, and the unencrypted one otherwise.

If <recipient user> wants to receive the file, <recipient user> types

:faccept <sender user> <path>

where <path> is the file system path where the received file will be saved to. If <recipient user> did not assign a key to <sender user> and the file transmission is encrypted, <recipient user> will see an error message. Otherwise, <recipient user>'s client sends a FILE_ACCEPT message to <sender user>, and the file transmission begins. The user <recipient user> is forbidden to change the key assigned to <sender user> before the first part of the file arrives. At the end of the file transmission, <sender user> and <recipient user> are informed by their clients.

Program code

All Java classes implementing the client are in the package chat_client. A UML class diagram is below.

*The best to click it to get access to download svg file, and open it in your browser.

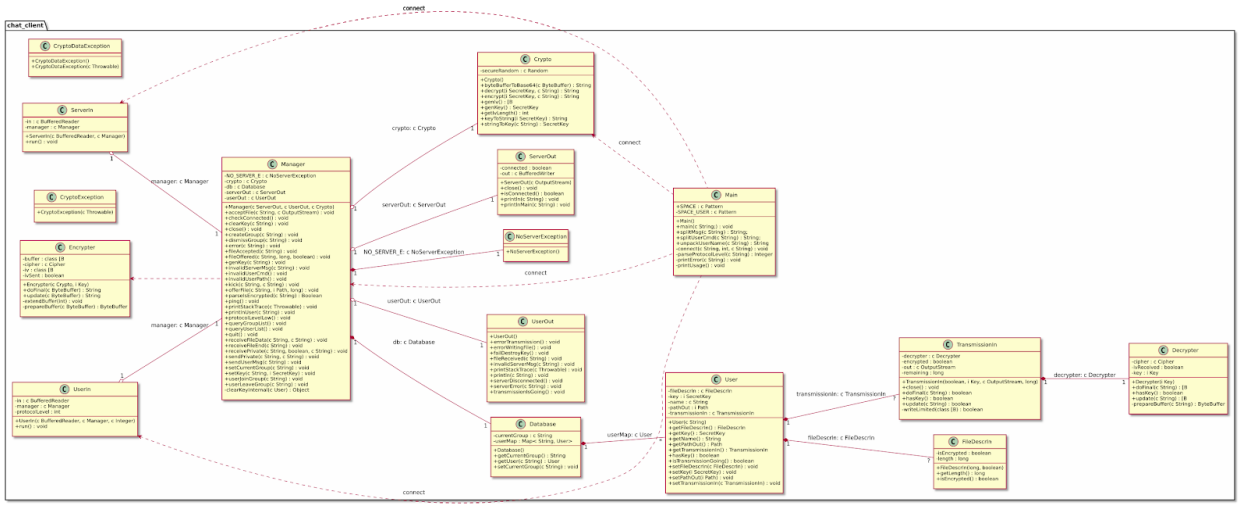

Figure 1. The UML class diagram for the chat client.

The method Main.main

- connects to the chat server whose address is given as a command-line argument,
- performs a handshake (in the method connect),
- starts a thread running the class ServerIn which handles protocol messages from the server,
- the main thread runs the class UserIn which handles commands and messages from the user.

Since there are two threads changing the state of the client, they can change it simultaneously causing glitches. For example, if both threads send messages to the server, one message may be inserted into the middle of the other one. Or consider the following scenario:

- the thread running UserIn receives the command to clear the cryptographic key for <user b> and clears the key,
- the thread running ServerIn receives an encrypted message from <user b> and informs the user that it cannot decrypt the message because there is no key,

- the client informs the user that the key has been cleared.

It seems to the user that the client cannot decrypt the message even when the key has not been cleared yet.

To prevent such glitches, all state changes, printing messages to the user, and sending protocol messages to the server are done via an object of the class Manager created in the method Main.connect. Most methods of Manager are *synchronized*, and every such method performs some *transaction* on the state of the client. The Manager object prints messages to the user via an object of the class UserOut in its field and sends messages to the server via an object of the class ServerOut in its field. The Manager object stores users and their attributes in an object of the class Database in its field. The Database object creates a User object for any user on demand, if that user offered a file, if a cryptographic key was assigned to that user, and so on.

The thread running ServerIn receives a protocol message from the input stream from the server, parses the message, and calls the appropriate method of Manager in a loop. If the input stream is closed or an error happens when reading (IOException is thrown), the thread exits the loop. If an error happens during sending a message to the server (for example, because the TCP connection timed out), the ServerOut object becomes disconnected (see its method isConnected). Every transactional method of Manager checks whether the ServerOut object is connected. If it is disconnected, the Manager object throws NoServerException instead of performing the transaction. This causes the thread running ServerIn to exit the loop. The exception NoServerException is not thrown during a transaction in order to let the transaction finish and get a consistent database state.

The thread running UserIn (the main thread) receives a user input, parses it, and calls the appropriate method of Manager in a loop. If the input stream is closed or an error happens when reading (IOException is thrown), the thread exits the loop. Also if the ServerOut object of the Manager object becomes disconnected, the thread exits the loop. After exiting this loop, the main thread terminates. When the main thread terminates, the thread running ServerIn terminates too because it is a daemon thread.

When a user <user b> offers a file to the user controlling the client, the parameters of the file are stored in a FileDescrIn object in the User object for <user b>. When the user accepts the file, the FileDescrIn object is deleted, but a TransmissionIn object is

created. The client receives data and gives them to the methods update and doFinal of the TransmissionIn object. If the data are encrypted, the TransmissionIn object decrypts the data with the help of a Decrypter object (see below). Then the TransmissionIn object writes the data to the file. Hence the TransmissionIn object is created for every file transmission.

The class Crypto contains the state of the cryptographic system. The client encrypts and decrypts private user messages via the methods of Crypto. The client encrypts files using the class Encrypter. An Encrypter object is created for every file transmission. The client decrypts files using the class Decrypter. An Encrypter or Decrypter object contains the state of the cipher since it processes a file by parts. The classes Encrypter, Decrypter, and TransmissionIn may throw CryptoException, CryptoDataException, or IllegalArgumentException if an error happens during data processing. IllegalArgumentException means that the unencrypted data received from the server are invalid, for example, the text is not Base64. CryptoDataException means that the cryptographic data received from the server are invalid, for example, an encrypted private message is not Base64 or is encrypted by the wrong key. Invalid data may also be sent by a malicious user. If IllegalArgumentException or CryptoDataException is thrown, the client informs the user about the issue and continues operating. CryptoException means that there is an error in the program code, so the client terminates.

Further information can be found in the source code.

Server implementation

User interface

The server can be started by executing the class chat_server.Main with arguments:

- chat server port number.

Program code

All Java classes implementing the server are in the package chat_server. A UML class diagram is below.

*The best to click it to get access to download svg file, and open it in your browser.

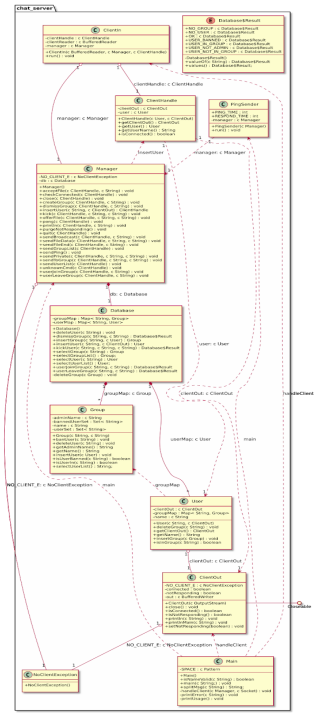

Figure 2. The UML class diagram for the chat server.

The method Main.main

- starts a thread running the class PingSender,
- creates a listening ServerSocket on the chat server port,
- for every connected client, calls the method Main.handleClient which
    - performs the handshake, obtaining the name <user> of the user controlling the client,
    - inserts <user> into the database into the list of logged-in users,
    - starts a thread running the class ClientIn which will handle messages from

that client.

The state of the server is stored in an object of the class Database. Since there may be more than one thread changing the state of the server, the threads can change it simultaneously causing glitches. For example, consider the following scenario. There are three distinct users: user0, user1, user2. Messages from user0 and user1 are handled by threads thread0 and thread1 respectively. Suppose that user2 belongs to a group. Suppose that user0 sends a message doc0 to the group, and user1 kicks user2 from the group. Then the following sequence of events is possible:

- thread0 reads a list of users belonging to the group, and user2 is among them;
- thread1 removes user2 from the group and sends the corresponding notification to user2;
- thread0 sends doc0 to user2.

Thus user2 sees the notification that user2 was kicked from the group, then a message from that group.

To prevent such glitches, all state changes and sending protocol messages to clients are done via the object of the class Manager created in the method Main.main. The Manager object contains a Database object in its field. Most methods of Manager are *synchronized*, and every such method performs some *transaction* on the state of the server. The Manager object sends messages to a client via the ClientOut object contained in the User object for that client contained in the Database object.

A thread running ClientIn receives a protocol message from the input stream of the client, parses it, and calls the appropriate method of Manager in a loop. The thread contains a ClientHandle object that identifies the user communicating with this thread. The thread passes this ClientHandle object to the Manager object. If the input stream is closed or an error happens when reading (IOException is thrown), the thread exits the loop. If an error happens during sending a message to the client (for example, because the TCP connection timed out), the ClientOut object becomes disconnected (see its method isConnected). Every transactional method of Manager checks whether the ServerOut object is connected. If it is disconnected, the Manager object throws NoClientException instead of performing the transaction. This causes the thread running ClientIn to exit the loop. After exiting the loop, the thread deletes the user that communicated with this thread from the list of logged-in users in the database. The

exception NoClientException is not thrown during a transaction in order to let the transaction finish and get a consistent database state.

The thread running PingSender periodically calls Manager.sendPing. This method sends the PING protocol message to all clients and sets the Boolean attribute notResponding of the corresponding ClientOut object to true. If a client replies with the PONG protocol message, notResponding is set to false. When RESPOND_TIME passed after the ping, the thread disconnect all clients that have notResponding true. When the main thread terminates, the thread running PingSender terminates too because it is a daemon thread.

Since the protocol uses only end-to-end encryption, the server does not perform any cryptographic operation.

Further information can be found in the source code.