Internet Technology Report

CHAT SERVER AND CLIENT

ARTEM AVETYAN (464761)

Version – 1.0

Created – 21/01/2020

Modified – 22/01/2020

Version history

Version 0.1:

The first version of the document. Contains initial layout and chapter headers

Version 0.2:

Added protocol specification

Version 1.0:

Added rest of the content

Contents

1 Summary

This paper is a report on the “Chat server and client” assignment for the course “Internet Technology” by Artem Avetyan.
The assignment is following:

Implement chat server and accompanying client.

2 Notes

1) Encryption is implemented only for private messages. For the session key there is no expiration time, because every time users logs out and in again a new session key will be generated.
2) About why client is disconnected if login is not valid. Client is being disconnected in case of submitting invalid or non-unique login in order to keep it compatible with Level 1 server
3) To calculate file’s checksum I decided to come up with my own (very simple) algorithm rather that choose something from the Internet
4) In my implementation I have decided to not let admins of the groups to leave the group. There was nothing specifying about that in the assignment description.
5) When I attached class diagrams to this document, I have noticed that the quality is lost, and because those diagrams are relatively big, I have decided to attach links to Google Drive where you can see them

3 How to run

I have added four configurations to the program:

1) Server
2) Client
3) Client (1)
4) Client (2)

Please first start the Server, and only then clients

4 Brief description of the design

4.1 Client

Each client connects to the server and starts two threads:

1) To listen to the user input
2) To listen to the server messages

Both threads do the same thing:

1) Read the input
2) Wrap it into a Message object
3) Path this object to the special class that takes care of it (ClientSideMessageInterpteter)

Message interpreter then performs actions (like sending messages to the server, printing messages out etc.) depending on the type and sender of the Message

4.2 Server

Server waits for a client and starts two threads:

1) To listen to the client's messages
2) To periodically exchange ping-pong messages with the client

First thread does as follow:

a. Read the client's input
b. Wrap it into a Message object
c. Path this object to the special class that takes care of it (ServerSideMessageInterpteter)

Message interpreter then performs actions (like sending messages to the client, printing messages out, disconnect client, send group messages etc.) depending on the type of the Message

Ping-pong thread does as follow:

1) Sends to a connected client a Ping message
2) Waits 3 seconds
3) Asks message interpreter to check whether that client have sent a pong back

5 Client class diagram

https://drive.google.com/open?id=10zEFO9KDQbYzNZy9Ff8Bm7upi3NXsdra

6 Server class diagram

https://drive.google.com/open?id=1xHt_hrjsiTk_oCoBcCTb5sO3ZFyEBRqQ

7 Protocol specification

The protocol of the chat server is described below. The protocol provides the following functionality:

1. Setting up a connection between client and server.
2. Broadcasting a message to all connected clients.
3. Periodically sending heartbeat to connected clients.
4. Requesting a list of all connected users
5. Sending private messages
6. Creating a group of users
7. Joining a group
8. Sending a message to everyone in the group
9. Leaving a group
10. Kicking a user out of the group (for group admin only!)
11. Sending a file to a user
12. Disconnection from the server

Each of the above will be described. “C” represents a message from the client and “S” a message from the server. Parts of the messages in capital letters are ***not*** written by user!

7.1 Setting up a connection between client and server.

The client first sets up a socket connection to which the server responds with a welcome message. The client supplies a username on which the server responds with a +OK if the username is accepted or an -ERR in case of an error.

Happy flow:

Client sets up the connection with server.

S: HELO <welcome message>

C: HELO <username>

S: +OK HELO <username>

Error message:

S: -ERR user already logged in

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

7.2 Broadcasting a message to all connected clients.

Sends a message from a client to all other clients. The sending client does not receive the message him/herself but gets a confirmation that the message has been sent.
C: BCST <message>
S: +OK BCST <message>
Other clients receive the message as follows:
S: BCST [username] <message>

7.3 Periodically sending heartbeat to connected clients.

Sends a ping message to the client to check whether the client is still active. The receiving client should respond with a pong message to confirm it is still active. If after 3 seconds no pong message has been received by the server, the connection to the client is close (before closing the client is notified with a DSCB message). The server periodically sends a ping message.
Happy flow:
S: PING
C: PONG
Error message:
S: DSCN Pong timeout

7.4 Requesting a list of all connected users

The client can request a list of all connected users. Since the server also shows the user who is requesting, the list will never be empty.
Note: if a user from the list is connected, but not yet logged in, (s)he will show up in the list as "unknown".

*C: GET_USERS *all*
S: +SUCCESS [Rogier, Steven, unknown]

7.5 Sending private messages

A client can send another client a private message. This function makes use of **encryption** (symmetric and asymmetric)

Note: in this description, "*SC*" will stand for "sender client" and "*RC*" for "receiver client".
First, sender client writes the following.

SC: PM @username message
Before sending this to the server, on the client-side checks whether a **session** established between these users. If **it is** so, the message is encrypted using the session key and then the following is sent to the server:

SC: PM @username encryptedMessage

Once server received that message, first it checks whether such a user to whom the message is being sent does exist. If **not**, the following message will be sent back to the sender:

S: -SOFT_ERR no such user!

If the user exists, then server will send the message to that user:

S: PM senderClient encryptedMessage

The receiver client will first check is it has a **session** established with the sender. If yes (which *is* in our case), (s)he will decode the message and print it:

[sender] message

Now, let's look at scenario where session has **not** yet been established between clients.
In that case, the sender cannot find a session key for the receiver client. So, instead of sending the message right away, it first will do the following:

1) Temporary store the contents of private message on client's side using time stamp as a key (because time stamp is unique, you cannot send to messages at the very same moment. Later that time stamp will be used to find that private message's content)
2) Request **public key** of the receiver client from the server by sending the following message:

   SC: REQUEST_PUBLIC_KEY @receiverClient timestamp

   When server receives that message, it first checks whether a user with the name of *receiverClient* exists. If not, the following will be sent back to the sender:

   S: -SOFT_ERR no such user!

   If the receiver client exists, then the server will send him/her the following message:

   S: REQUEST_PUBLIC_KEY @senderClient timestamp

   When receiver client gets that message, it will send its public key to the server:

   RC: PUBLIC_KEY @senderClient publicKeyOfReceiver timestamp

   Server then will send that public key to the sender client:

   S: PUBLIC_KEY @receiverClient publicKeyOfReceiver timestamp

   Once sender client receives that message, it will do the following:

   1) Get the content of the private message using the **timestamp**

2) Generate a new session key for communication between him/her and the receiver
3) Encrypt that session key using **public key** of the receiver
4) Encrypt the content of private message using that session key
5) Send the following to the server:

SC: PM @receiverClient encryptedPM encryptedSessionKey

Server will send the following to the receiver client:

S: PM senderCLient encryptedPM encryptedSessionKey

Receiver client will now check if there is a session established between him/het and the sender. In this case, there is **no** session, which means that the message contains a new session key! Therefore, receiver client will do as follows:

1) Decrypt the new session key using its **private key.**
2) Save that session key for the future communication with the sender.
3) Decrypt the contents of private message using that session key.
4) Print out the following:

[sender] decryptedPrivateMessage

7.6 Creating a group of users

Clients can create a group by sending the following message to the server:

C: CREATE_GROUP /create_group groupName

Server will first check if a group with such a name already exists. If yes, the following will be sent back to the client:

S: -SOFT_ERR a group with that name already exists!

If no, then the server will create that group, set the client as group's admin and send the following message back:

S: +SUCCESS group "groupName" has been created!

7.7 Joining a group

Clients can join a group by sending the following message to the server:

C: JOIN_GROUP /join groupName

Server will check following things:

1) If group with that name exists. If not, the following will be sent back to the client:

   S: -SOFT_ERR no such group!

2) If the client is already a member of admin of that group. If yes, the following will be sent back:

   S: -SOFT_ERR you are already a member of that group!

3) If user is in the banned list (meaning he has been once kicked out of that group). If yes, the following is sent back:

S: -SOFT_ERR you are banned from that group!

4) If none of the above applies, then server will add the client to that group and send following message back:

S: +SUCCESS you have successfully entered group "groupName"

7.8 Sending a message to everyone in the group

Clients can send group messages as follows:

C: GROUP_MESSAGE @@groupName message

When server received that message, it will do the following:

1) Check if a group with such name exists. If no, it will send the following message back:

S: -SOFT_ERR no such group!

2) Check if the sender is a member (or an admin) of the group. If no, it will send the following message back:

S: -SOFT_ERR you are not allowed to write in that group!

3) If none of above applies, it will send the message to each member (and admin) of the group:

S: GROUP_MESSAGE [sender] message

Clients who receive that message will print out the following:

[sender] message

7.9 Leaving a group

Clients can leave groups as follows:

C: LEAVE_GROUP /leave groupName

When server received that message, it will do the following:

1) Check if a group with such name exists. If no, it will send the following message back:

S: -SOFT_ERR no such group!

2) Check if the sender is a member of the group. If no, it will send the following message back:

S: -SOFT_ERR you are not a member of that group!

3) Check if the sender is an admin of the group. If yes, it will send the following message back:

S: -SOFT_ERR you are an admin, you can't leave the group!"

4) If none of the above applies, the server will remove sender from the group and send back following message:

S: +SUCCESS you have successfully left the group "groupName"

7.10 Kicking a user out of the group (for group admin only!)

Admin of a group can kick out a group member by sending the following message:

C: KICK /kick groupName clientNameToKick

When server received that message, it will do the following:

1) Check if a group with such a name exists. If no, then it will send the following message back:

S: -SOFT_ERR no such group!

2) Check if sender is admin of the group. If no, then it will send the following message back:

S: -SOFT_ERR you are not allowed to kick members out that group!

3) Check if sender (who is admin) kicks out him/herself. If yes, the following message will be sent back:

S: -SOFT_ERR why would you kick yourself out of your group :)?

4) Check if user that is being kicked out is a member of thee group. If no, the following message will be sent back:

S: -SOFT_ERR user clientNameToKick is not a member of group groupName

5) If none of the above applies, the server will do as follows:
    a. Kick the client out from the group (with adding him/her to the banned list)
    b. Send a group message to all group members –

    S: GROUP_MESSAGE user memberToKick has been kicked out of group groupName

    c. Send a success message to the admin –

    S: +SUCCESS user memberToKick has been kicked out of group groupName

    d. Send a message to the kicked-out member –

    S: +SUCCESS you have been kicked out of the group groupName

7.11 Sending a file to a user

Clients can send files to each other. The procedure is described below.
Note: in this description, "*SC*" will stand for "sender client" and "*RC*" for "receiver client".
First, sender client writes the following:

SC: FILE_TRANS /send_file @receiverClient filePath

Before message is sent to the client, on the client's side there is check whether a file with such a file path exists. If no, the following will be printed out:

No such file!

If the file exists, client will perform the following:

1) Reads the bytes of the file
2) Generates meta data for that file. (meta data consists of the file's size, name and a checksum)
3) Saves the file on the client side using it's meta data as a key.
4) Sends the following message to server:

SC: FILE_TRANS /send_file @receiverClient metadata

When server received that message, it will do the following:

1) Check if receiver client exists. If none, it will send the following message back:

S: -SOFT_ERR no such user!

2) Check if sender client sends the file to him/herself. If yes, it will send the following message back:

S: -SOFT_ERR you can't send a file to yourself!

3) If none of the above applies, it will send a message to the receiver client:

S: FILE_TRANS @senderClient metadata

When receiver client receives that message, it will do as follows:

1) Save metadata on the client's side
2) Create a directory and name it by client's login (if it doesn't exist yet)
3) Print the following out:

User "sender" sends you a file!

4) Sends message to server saying that it is ready to receive the file

RC: READY_FOR_FILE @senderClient metadata

Once server received that message, it does as follow:

1) Starts separate file transfer server on new port
2) Sends following message to the sender client:

S: READY_FOR_FILE @receiverClient metadata portNumber

3) Sends the following message to the receiver client:

S: OPEN_PORT_TO_RECEIVE portNumber

Now, both clients know which port to listen to connect to file server. Let's look at what happens on the client sender side

When sender client receives a message with the receiverClient, metadata and portNumber, it does as follow:

1) Opens a new thread for file sending
2) Connects to the file server using the port number

3) Gets file from the storage using metadata (remember, the file was stored before using that same metadata as key)
4) Sends the following message to file server:

SC: FILE <senderClient @receiverClient metaData file;

At the same time, receiver client received the port number and does as follow:

1) Opens a new thread for file receiving
2) Connects to the file server using the port number
3) Sends a message to the file server to confirm that (s)he has opened a socket to receive the file

RC: OPEN_PORT_TO_RECEIVE receiverClient

Once file server receives that message, it sets the socket from where the message just came as a file transfer socket for the client under receiverClient name

Now, let's get back to where the sender client sends the file to file server.

When file server receives a message from the sender client containing the file, it sends the following message to the receiver client (using its special file transfer socket, that was stored before:

S: FILE <senderClient @senderClient metaData file

When receiver client receives the file, it does as follow:

1) Generates metadata from that file
2) Compares that metadata to the previously saved
3) If new metadata matches previously saved metadata, file is saved in the client's directory
   a. Sends the following message to the file server:

   RC: FILE_STATUS file has successfully been delivered! @senderClient

   b. Prints out following message:

   File "filePath" has successfully been saved!

4) If new metadata does not match previously saved metadata, file is not saved, and receiver client does as follow:
   a. Sends a message to the file server:

      RC: FILE_STATUS File has been corrupted and not delivered! @senderClient

   b. Prints out a message:

      File "filePath" has been corrupted and not saved!

5) In both cases, client removes previously saved metadata from the storage
6) Closes the file transfer socket
7) Stops the file receiving thread

When the file server receives file status message from the receiver client, it does as follow:

1) Sends that message to the sender client (removing the part containing sender's name):

RC: FILE_STATUS File has been corrupted and not delivered!

OR

RC: FILE_STATUS file has successfully been delivered!

2) Closes file transfer socket for receiver client
3) Closes file transfer socket for the sender client
4) Stops file transfer server

Once sender client receives the file status message, it does as follow:

1) Prints out the status:

File has been corrupted and not delivered!

OR

File has successfully been delivered!

2) Closes file transfer socket
3) Stops file sending thread

7.12 Disconnection from the server

Disconnection from the server happens as follow:

C: QUIT q

S: +OK Goodbye

/send_file @aa image.png