Report

Transport Layer Design

Packet Loss – timer + retransmit (ACK/Sequence num)

We use a timer and ACK/Sequence to detect the packet loss. If the packet loss is detected, the packet will be retransmitted. As the UDP packet is sent from the client side, a timer will start. If the packet is lost, whether it's lost on sending the request or on receiving the response, the timeout exception will be triggered, and the packet will be resent. The program will keep resending the packet if the response is not received. At the client/server sides, we maintain the ACK and Sequence number. The ACK and Seq are initialized to 0. Each packet will be sent with ACK, Seq number and the Length of the data(the length of the actual command excluding the ACK, Seq, Length parts). On the server side, we will check if the ACK maintained on server side is equal to the Seq + Length. If it doesn't match, there's no packet loss and sending the response normally. If matched, there is a duplicated packet, which means the packet is lost and we retransmit the messages. Every time we send the UDP message, it is stored in a buffer, which is globally shared. The retransmission will send the message in the buffer. We only check whether the packet is lost or not and we don't check if the content of the file is corrupted or not.

Concurrency - Multi-threading

We use multi-threading to solve the concurrency issue in TCP transferring. As the specification described, the maximum size of the UDP segment is 65535 bytes. So we think there won't be too many problems with UDP flow. Another reason is that our program would be thread-unsafe if we use multi-threading on UDP. Even though we assume the process is atomic, we still need to consider the safety of variables shared between threads. So we keep UDP single thread. We set the TCP transferring multi-threading as the TCP may transfer a large file and we don't want the file transferring congesting the network. We start the thread when the server receives a large file and sends a large file. As soon as the TCP welcome server builds the handshaking, the file transfer will be done on another thread and the user can continue to do other things. We don't handle the error in the TCP file transferring, which means we don't know if the content of the file is corrupted or not. We don't use multi-thread at the client side as it won't have a concurrency issue in this assignment.

Program Overall Design

Data Structure

There is no data maintained on client side. On the server side, we mainly maintained two data structures – userInfo and threadInfo. The data structure userInfo is a map, the key is the usernames and the value is the map of the user information, including the password and user status(online/offline). The threadInfo is a map, the key is the name of the thread and the value is the map of thread information, containing the owner of thread, the message counter and the files, where files is the map of the files maintained in the thread, the key is the file name, the value is the new file name(thread + filename).

Message Format

There is no fixed message format for the transport layer. Most of the messages sent by the client contain the command issued and the user name wrapped with ACK, Seq and Length. Below is a brief conclusion(in format: Command – client side – server side(variables in italics)):

(CommandSpec is the specification input after the command(e.g. RMV 3331 -> command: RMV, CommandSpec: 3331))

AUTH – AUTH name *userName* – TRUE(username exist)/NEWUSER(creating a new user)/ONLINE(user already online)

AUTH – AUTH psw *psw* – TRUE(password correct/new user created)/FALSE(incorrect password)

CRT – CRT *username commandSpec* – TRUE(success)/FALSE(thread exists)

MSG – MSG *username commandSpec* – TRUE(success)/FALSE(thread does not exist)

DLT – DLT *username commandSpec* – NOTHREAD(thread does not exist)/NOMESSAGEID(message id does not exist)/NOUSER(user does not exist)/TRUE(success)

EDT – EDT *username commandSpec* – NOTHREAD(thread does not exist)/NOMESSAGEID(message id does not exist)/NOUSER(user does not exist)/TRUE(success)

LST – LST *username* – FALSE(no thread to list)/*the list of thread in String*

RDT – RDT *username commandSpec* – FLASE(thread does not exist)/EMPTY(empty thread)/*the thread information in String*

UPD – UPD *username commandSpec* - NOTHREAD(thread does not exist)/ FILEEXIST(file already exist in the server)/TRUE(be able to accept the file/the file is already uploaded)

DWN - DWN *username commandSpec* - NOTHREAD(thread does not exist)/ NOFILE(file does not exist in the server)/TRUE(file successfully sent)

RMV - RMV *username commandSpec* – FLASE(thread does not exist)/ NOPERMISSION(user has no permission to delete the thread)/TRUE(success)

EXIT – XIT *username* - TRUE(success)

Functions

The main function - set up the client socket, do the authentication, then go to the while loop. Inside the loop, show the menu, parse the command and switch to the corresponding function. Each command has a function named as below:

CRT – createThread(); MSG – postMessage(); DLT - deleteMessage(); EDT - editMessage(); LST - listThreads(); RDT - readThread(); UPD -uploadFile(); DWN – downloadFile(); RMV - removeThread(); XIT – exit(); AUTH – authentication().

We add **AUTH** such that the authentication could also be processed.

UDPSend() – take the sentence(String) and wrap with ACK, Seq, Length, send it to the server as a UDP packet.

UDPReceive() – receive the packet from the server and parse the data, check if the ACK, Seq, Length is correct, and return the response(String).

TCPSend() – Set a new TCP socket, take the file name, find the file in the directory, read the file and send it. The TCPSend() on server side will send the file on another thread.

TCPReceive() – Set the welcome socket, take the file name and create the file, read the byte into the file. The TCPReceive() on the server side will accept the file on another thread.

At the server side, we create a class called **ClientThread** which extends the Thread. This is for the TCP concurrency issue. We set the function we need to use(i.e. TCPSend/TCPReceive) and start the thread. After the sending/receiving is done, it will clear the setting automatically and exit the thread.

Reference

stringIsInteger(): https://stackoverflow.com/questions/5439529/determine-if-a-string-is-an-integer-in-java - check if a string is an integer