COMP3331 Networks Assignment Report

Program design

The program has a client server design. The server is constantly running a TCP welcome port. When a client is initiated, it connects to this TCP welcome port on the server, causing the server to create a new thread to specifically handle that client and all subsequent processing for that client is handled on this thread. The client also creates a single extra thread to handle any incoming UDP file transfers. The client commends are processed in the main thread of the file and this UDP thread servers only to receive files.

Data structures

The client does not have any data structures. It keeps track of current files through the file APIs in Java rather than a constant data structure.

The server has a credentials HashMap to hold all the device names as keys and their corresponding passwords as values for authentication. It also holds a HashMap called deviceConsecutiveLoginAttempts, which is used to keep track of how many consecutive invalid login attempts a device has made, with the device name as the key and the number of attempts as the value. There is a third HashMap called deviceTimeoutDate, which holds a davice name as trhe key and the corresponding more recent date a device was timed out. All timeout checks check if the time between the date in this HashMap and the current date is more than 10 seconds. If it is less, the device is considered timed out. The server also uses Java file APIs to keep track of existing files.

Details of the application layer protocol

TCP

The design of my program uses its own application layer protocol in the form of constant variable values to be used as response status messages, like HTTP status codes. Each command has a set of corresponding status messages for both client and server actions and these status codes are shared between the client file and the server file. These response statuses are used to communicate to the server what command the client requests to be processed and allows the server to tell the client if command processes were successful or ran into errors.

After a client issues a command to the server, it will wait for a response status before continuing. Depending on the status, the client will either display an error to the terminal or a success message.

UDP

The p2p portion of my program first uses the previously mentioned custom application layer protocol to first ask the server to send back the IP address and port number details of a specified edge device. The server checks the current edge device log for the details and sends it to the client. Once this information is obtained by the client, the client then creates a UDP socket to send UDP packets to the destination.

All clients have a thread running just for receiving files from another client for p2p file transfer. This thread is started when the authentication for the client is successful and runs in a permanent while loop. If the main thread is closed, it will send a packet to this thread telling it to also terminate.

When sending the actual files, the presenter client first sends two separate UDP packets to the audience client. The first packet holds the name of the file being transferred. The second packet holds the total size of the file.

It also creates another UDP socket for accepting ACKs (acknowledgements) that a particular UDP packet was successfully transferred before sending another UDP packet and sends the audience client the port number of this socket. Without this ACK mechanism, too many UDP packets are sent at once and some files are lost. The port number is randomly generated.

Because UDP packets have a maximum size, the file to be transferred is read in chucks using a buffer array of a constant size, specifically 32768. The presenter client will read 32768 bytes worth of data from the file into the buffer array and send it via a UDP packet to the audience client using the IP address and port number received from the server earlier. The audience then writes this UDP data into a file with the name given in the first UDP packet transferred. It then sends an ACK message back to the presenter client which causes the client to send the next chunk of data until the entire file is read.

For the final UDP packet, there is a special conditional check in the audience client, where if the total size of the file (given by the second UDP packet from presenter to audience) minus the size of the current file is larger than the buffer size, then only read the difference between the two from the buffer in the UDP packet. If this isn’t done, the audience client will read some uninitialized values from the buffer array since there just wasn’t anymore data in the file on the presenter’s side to put into the buffer.

Once this is done, the p2p transfer finishes and the while loop resents the audience thread to wait for the next file transmission.

Trade-offs considered

The p2p transfer of files uses an ACK system where for every chunk of a file send, the presenter client needs to wait for a confirmation from the audience client that the chunk was received before sending the next chuck. This prevents too many UDP chunks from being sent at once, which may overflow the packet buffer on the audience client but does slow down the file transfer.

Point out issues if program does not work under certain circumstances

The program will have an error if the machine is slow, and it takes time for the Server to write to the edge device log file combined with an invalid port being specified for the client for the UDP socket. The client will logout immediately after the port number is deemed to be taken and the server will try to remove the client’s data from its edge device log. However, if the server hasn’t updated the log fast enough before the edge device is logged out, when the server tried to remove the device data, an issue will occur since it’s not there. This will cause an array out of bound index error.

If a client is to receive multiple files at the same time via p2p, and error will occur since the second file will interrupt the transmission of the first.

If a client is being logged in from multiple terminals at once, this may cause a concurrency issue since they are both trying to access the shared data structures used for time outs and authentication.

If the data generated by the EDG command is too big (in the 10,000s), transferring the file over the TCP socket to the server will cause an error because the file is too big.

You must only pick port numbers between 1025 and 65534 for the client argument. If the port number is taken, a warning will be printed to the terminal and the client will close.

Reference any and all borrowed code

https://www.baeldung.com/java-check-string-number
https://stackoverflow.com/questions/9898512/how-to-test-if-a-double-is-an-integer
Used the code from these two websites create a function to check if a given string is an integer. It has been slightly modified to check for extra edge cases (e.g., 3.0 still not counting as an integer).

https://stackoverflow.com/questions/434718/sockets-discover-port-availability-using-java
Used code from this website to create a function to check if a given port number was in use.

https://webcms3.cse.unsw.edu.au/COMP3331/22T3/resources/80563
Used the starter code provided by the 3331 staff for the client program.

https://webcms3.cse.unsw.edu.au/COMP3331/22T3/resources/80564
Used the starter code provided by the 3331 staff for the server program.

https://webcms3.cse.unsw.edu.au/COMP3331/22T3/resources/80534
Modified the **printData** function in the 3331 22T3 week 2 lab to create a function that takes a UDP packet and returns the string value inside the packet.