**Data Communications P2P Project:**

**A P2PAPP!**

**Project Contributers:**

Jenny Ta

Lyndsie Whalen

Thavisha Lokuge

**Course Details:**

CPS706

Professor Jelena Misic

COMPILING AND RUNNING

**NOTE: All source files are located under cps706/src/datacomm/**

**NOTE: It is recommended to use Netbeans to compile this project, instructions are for Netbeans only.**

**NOTE: All pictures should be located in the root directory of the project (where the documentation will be). Three random sample pictures will be included.**

To compile and run the program is as follows:

Open Netbeans

Go to File => Open Project

Select the project from the list of files

Right click DataComm, (not the datacomm package) and select clean and build.

Right click DataComm once again and select run, or alternatively right click Main.java and select run

A frame will pop up with two buttons – one to spawn a server and one to spawn a client. Only one server can be running at one time, but multiple clients can be running. Each client requires a client name – this is so that any files that are transmitted will be saved in a directory pertaining to that specific client.

First in the client, select the upload button and select one to several JPEG image files from the following prompt.

You can now observe that the server screen has updated.

To search for a file in the directory, type the file name into the text field and hit Search.

To rate a file in the directory, select the file and hit rate.

To download a file, first search for the file, select a file name and hit download. You can now observe that a file has been downloaded into the client’s folder.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*NOTE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

During the running of this program, the server might run into a Windows Socket Exception Error: 0 Cannot Bind.

This error seems to occur randomly, most frequently when a timeout event occurs happens. All socket connections are closed after use as well as during all error handling and due to the random nature of this error, we were not able to determine the source before submission.

It must be noted that the program CAN run without any errors and should the error occur, restarting the program will cause the exact same sequence of events to happen properly and without errors.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

CLIENT.JAVA

*Client*

A port number is assigned to the client as new connection is established

*initRateFrame*

This function simply creates a new JFrame that handles getting input to rate a selected file.

*InitComponents*

This method initializes the GUI components for the client

*UpdateDirectory*

This method updates the client’s current directory listing with the server’s current directory listing. Used for display in the QUERY\_FOR\_CONTENT method.

*waitForAck*

This client method waits for an acknowledgment to be received from the server process. The waitForAck also starts the timeout using a previously calculated timeout variable. If the ack does not happen before the timeout, the function exits with a REPLY\_TIMEOUT which lets SendUDP know that the packet was not transmitted.

*SendWelcomePacket*

Sends the handshake to the UPD server from the client to obtain the working ports for the duration of the client’s session

*UDPSend*

When this method is executed, packets are sent to the server. As each packet is sent waitForAck is called. If a timeout Event occurs at this point, the packet is resent up to 5 times. When all smaller segments of a larger packet (if any) have been sent to the server, the FIN packet lets the server know that there are no more segments being sent. The timeout for previously sent packets is also handled here using the timeout = ERTT + 4\*DEVRTT function. The timeout is calculated every time a packet is sent based on previous values.

*sendExitPacket*

This method creates and sends a packet of type EXIT, which will inform the server that the client wishes to terminate the connection.

*InformAndUpdateActionPerformed*

Gets the files to be send from the GUI, creates a message, converts the message to a DatagramPacket that has been segmented if necessary, calls UPDSend which handles the sending, timeout and retransmissions.

*QueryForContentActionPerformed*

Gets the filename to search from the GUI, calls UPDSend, gets the reply from the server, and displays the results in the listfield of the GUI.

*RateContentActionPerformed*

Gets the selected file, asks for a rating. Sends the rating to the server which handles the directory updating, and updates the current directoryListing for the client side. Redisplays the listing with the altered rating.

DIRECTORYSERVER.JAVA

*DirectoryServer*

The constructor creates a new array to store the directory contents as well as calls initComponents to initialize it’s GUI components. It then calls packetWaitLoop to start listening for packets

*InitComponents*

Initializes the GUI components in the server for the displaying of the files

***private class ServerThread*** *–* when a server receives a new connection request it spawns a new thread to handle all traffic with that client.

*run*

The run method executes the handlePacket method.

*HandlePacket*

This method waits to receive packets from the client: If a WELCOME packet is received by the server, it will respond a welcome reply back to the client. If a FIN packet is received, the server will proceed to assemble smaller segments into the original larger packet if necessary, and then clears the stored array list of packets. If packets are of neither type, the server will store the packets in an array list until it receives a fin.

*sendWelcomeReply*

When a client first connects to the server, the client must be assigned an unused port number from the server. During the duration of the connection, the assigned port number will be used to transfer packets back and forth. This new port is sent in a reply packet to the client.

*UpdateFileListing*

The query message of type INFORM\_AND\_UPDATE triggers this method; UpdateFileListing updates the current directory with the new file entry (or entries) as long as the entries are unique

*queryListing*

This method is called upon when the query message is of type QUERY\_FOR\_CONTENT. The header of this packet will contain the search query, which the server uses to find matching file entries in the directory. The results are placed in a packet (or segmented into packets) and sends them to the client.

*rateContent*

This method is executed when the packet message is of type RATE\_CONTENT. The header of the packet will contain the rating sent. The rateContent method will call the DirectoryListEntry’s rateContent method with the rating, which rates the file.

*registerClientExit*

When a packet message is of the type EXIT, this method will be executed. All entries associated with the client that sent the packet will be deleted from the server.

*parsePacket*

The parsePacket method extracts the packet type from the request message and calls upon the needed methods accordingly. The four types INFORM\_AND\_UPDATE, QUERY\_FOR\_CONTENT, RATE\_CONTENT, and EXIT.

*sendAck*

Upon receiving a request from the client, the server responds with an acknowledge message so that the client knows that it has received its request. The ack will contain the segment number of the packet it is acknowledging.

*waitForAck*

This server method waits to receive an acknowledgment message from the client.

*SendPacket*

The sendPacket method sends a packet through the socket to the client.

*BuildListTextArea*

Updates the directoryList GUI with the contents of the directory

*printDirectory*

The method printDirectory prints out all the current existing entries in the directory. Used mainly for debug

*packetWaitLoop*

The server listens via port 40110 for packets. Once it receives a packet, it creates a new thread which will from then on service the client that sent the packet.

GLOBALS.JAVA

*Normalize*

This method discards of additional spaces in the string.

The methods *debug, output, error, exception, out and sendMessage* are for debugging purposes only. These methods are irrelevant to the program.

*DirectoryListEntry*

The constructor method of this class creates a new entry in the directory that contains the file name, host address of the client containing the file, size of the file, rating of the file and the port number of the client that contains the file.

*rate*

When called upon, the rate method rates the quality of the image. The rating of an image is determined by the formula r(n)= 0.5\*(n-1) + 0.5\*R. By default, the initial R value of a new file is 1. The rating is stored in the rate variable.

*convertToPacketData*

The convertToPacketData method converts the entries of the fileDirectory into a format that can be easier used in a packet – each field (name, address, size, rating, and port) are combined into a string and separated from each other with a semi-colon.

The methods g*etFile, getAddress, getSize, getRating* and *getPort* return the file name, host address, the size of the packet , the rating of the image, and the port number respectively.

*toString*

This method converts file details to a formatted String.

PACKET.JAVA

*Packet*

When called upon, the constructor of Packet creates a new Packet and a packets arraylist of type DatagramPacket . The Packet will contain the message to be sent, as well as the packet type, the port to send to, and the inet address. The array packets will hold the segmented message with the requestline and sequence number appended to the head of each chuck. Each packet segment is a total of 128 bytes. The segmenting algorithm will be discusses under the BuildPacket function.

*getPackets*

Returns the packets arraylist, whose entries are the segmented packets.

*BuildPacket*

This method is called upon by the client. A request message containing the request line, header line, and entity body are created. If the length of the request message is less than 128 bytes, the packet is then added into an array of packets which will later get sent to the server. If the request message is longer than 128 bytes, it is split into smaller segments of 128 bytes each. Each segment is assigned a header containing a sequence number and the requestline. The message is segmented using objects of type Segment. The offset is initially set to 0 and incremented by the grab length after each segment is assigned. The grab length is 128 – the length of the request line. Each segment is then appended to the end of the request line and converted into a DatagramPacket and added to the packets array.

When the server receives the smaller segments, it must reassemble the larger packet according to the segment numbers.

*buildEmptyClientPacket*

This method builds an empty packet containing only the packet type, the host name and address. When the client needs to send a **welcome** or **fin** packet to the server, this method will be called upon.

*buildClientWelcomePacket*

The method buildClientWelcomePacket returns an empty packet that is of the type WELCOME; this allows the server to know that a new client would like to establish a connection. The responding acknowledgment will assign the client a port number to use during the duration of the file transfer.

*buildClientFinpacket*

The buildClientFinpacket method returns an empty packet that is of the FIN type; when the FIN packet is sent, the server knows that the last segment of a packet has been sent. The server will not expect any more segments of a larger packet to arrive.

*buildEmptyServerPacket*

This method builds an empty packet on the server side. See buildEmptyClientPacket for more details.

*buildServerPacket*

This method builds a packet on the server side. The packet contains a request line, header line and entity body if it exists. If the packet is larger than 128 bytes, the packet will be split into smaller segments of 128 bytes. The packets get stored in an array which later gets sent to the client.

***class PacketSorter***

*compare*This method is used to compare segment numbers. It is called during the assembling of the packets to sort them by segment number.

*assemblePackets*

The method assemblePackets reassembles a larger packet if it has been broken into smaller segments. The segments are sent to the method in an array list, which is then sorted by segment number. Once they are sorted, each individual packet is taken out of the array list and combined together to form the full DatagramPacket, which is then returned out.

CLIENTP2PTCPCONTROL.JAVA

*ClientP2PTCPControl*

A new thread is created in this method to listen to all incoming activity from a specific client.

*Run*

This method calls upon the listen method

*Listen*

A TCP socket is opened to listen for data. The beginning of each data packet will contain one of two types: GET or TRANSMIT. If it is a GET message, the rest of the packet will contain data on the file (filename, host address, port). If a file with that filename exists in the client’s directory, it then transmits that file to the host address and port received. If a TRANSMIT message is received, it means that a file is being transmitted. A directory is created using the known client name (if it does not already exist) and the transmitted file is saved in that directory.

*requestFile*

Requests a file from another client by sending a GET message that contains the details of the file.

*transmitFile*

Transmits the full file to the client that is requesting it in a TRANSMIT message.

*setTransmitInfo*

Sets the host name and port than messages should be sent to.

MAIN.JAVA

*Main*

Creates and initializes the main frame and calls initComponents which initializes the individual components of the frame

*initComponents*

Creates and initializes the two buttons that spawn a server and clients, respectively. It adds a spawnServerListener which only creates a server if a server has not been created before to the spawnServerButton as well as a spawnClientListener to create a new server. Both of these will be created in their own separate thread.

**class MainThread**

*MainThread*

Creates a new thread to run a new instance of the class with the specified class name

*run*

Instantiates a new class in the thread