Writeup for Project 2

Name: Bram Kineman

Netid: kinemanb

Date: 3/27/2022

Question 1: Explain your code and how it works.

File: rSender.cpp

This source code file contains code to act as a sender with application layer functionality to ensure data is sent reliably using UDP. This file works in conjunction with rReceiver.cpp.

Structure: struct args

This structure is used to capture the required inputted arguments from the command line. Having a structure containing these arguments makes it easier for my passing them in future function calls. It also makes the code more readable by being able to provide clear variable names.

Structure: struct socketInfo

This structure contains information for the server-side socket. Like the ‘struct args’, it is handy to ‘containerize’ arguments that are related to each other and that will be used often. This structure will be used when setting up the server socket and when calling other functions that use the server socket.

Structure: struct packet : public PacketHeader

This structure contains everything needed and contained within a packet that will be sent over the network. It extends the provided PacketHeader struct which contains the header information. My packet structure which extends the provided PacketHeader struct contains an array that will hold the data that is sent over the network. It has size 1456, accounting for the header size.

Structure: struct packetTracker

This structure is utilized heavily when sending data. It acts as a centralized structure to track important data for the rUDP algorithm. It contains map data structures with keys of sequence numbers and values of the packets. Using maps allows me to organize the packets in order by sequence number, and keep track of which sequence numbers that have been successfully sent by having a key value pair. This structure has maps each for ACKed and unACKed packets, as well as a variable for storing the highest sequence number ACK the sender has received. Knowing which packets are ACKed and unACKed as well as the highest ACKed packet allow me to create the correct window size and know what packets to send next.

Function: retrieveArgs(char* argv[])

This function utilizes the above args structure to save the arguments inputted by the invoker. It assumes that the arguments inputted are in a correct order and that all are provided. It returns the structure that can then later be passed as one argument where needed.

Function: createSTARTPacket()

This function, like many of mine, does what it says. It will utilize the provided PacketHeader structure to create a packet used for the START message for the sender. It sets the type, length, and checksum to the correct values. It creates a random number using srand() for a seqNum. Since the seqNum is put in a variable stored in this returned packet, I can later use it for the END packet.

Function: createENDPacket(unsigned int seqNum)

This function is similar to the createSTARTPacket function, but it instead creates a packet with variables in the PacketHeader struct that indicates an END packet. This function takes in the seqNum that was generated in the createSTARTPacket() function, as they should be the same according to the project specs. It then returns the generate END packet.

Function: setupSocket(char* portNum, char* host)

This function uses the arguments inputted by invoker to set up a socket that will be used to send data out of. This function uses the portNum and host variables inputted to properly set up the socket on the indicated portNum and host machine. Most importantly, this function sets up the socket using UDP. This function uses the above socketInfo structure to store the needed variables for sending data through that socket. Those needed variables are found in the socketInfo structure. This function returns that socketInfo structure for later use.

Function: setSocketTimeout(int sockfd, int timeout)

This function does what it says, which is setting a timeout for a specified socket. It takes in the socket file descriptor for the socket to set a timeout on and an integer for the length of timeout in milliseconds. This function will of course only be used for the server socket. It will make the server socket timeout after the indicated amount of time while attempting to receive data into the socket. On second thought, I did not need to parametrize the timeout variable since I only need to set the timeout to a value of 0.5 seconds. Although, it does allow for more variability if someone wanted to change the timeout for potentially more efficient data transfer.

Function: readFileInToTracker(char* inputFile)

This function is a set up function used before sending data. Instead of reading the data from a file while sending data, I thought it would make more sense to have pre-read the data from the file in to the correct size packets before sending anything. This allows me to better track the amount of packets I will need to send, and allowing me to know when all of the packets have been sent by checking the size of the splices of data I read from the file. This function takes in the inputted file name and reads from that file. All data read from the file is put in to the above packetTracker struct, specifically the unACKed packets map, since no data has been sent yet. This function creates the entire packet before putting those packets in to the map, correctly incrementing the seqNum for each packet. It returns the packetTracker with a filled unACKed packet map.

Function: writeToLogFile(char* logFilePath, string type, string seqNum, string length, string checksum)

This function is a product of a refactor since the code within it is used in a lot of places. I need to write to the log file whenever I receive or send data, so it was an obvious move to make it its own function. This function takes in the necessary parameters that need to be logged according to the project specs. Additionally, it takes in the log file path.

Function: sendSTARTEND(socketInfo &socket, PacketHeader &packet, char* logfile)

This function operates to send the START and END packets. I originally had these as two separate functions for outputting logs for debugging purposes, but I figured I could combine them as they are identical, other than the different packet headers that it sends depending on the time of execution. This function takes in my socketInfo structure that contains the necessary information to send information out of that socket. It also takes in the packet header it wishes to send (START or END), and the logfile to write to after sending. I chose to have the program exit execution if the start or end packets fail to send because if the sendto() function returns -1, it is like an issue with the receiver socket and will likely do it again so putting it in a loop to retry would not be beneficial.

Function: getSTARTENDACK(socketInfo &socket, PacketHeader ACKPacket, char* logfile)

This function does what it says and operates to receive the ACK packet for the sent START packet. Like the previous function, I originally had this function split up in to separate functions, on for getting the ACK for the START, and the other for getting the ACK for the END. This allowed me to debug and see what was happening better with outputting to the terminal in the separate functions. I combined them because they are functionally doing the same thing. This function takes in the socketInfo structure that contains the necessary data about a socket to receive in to it, an empty ACKPacket to fill with the received data, and the logfile for logging purposes.

Function: getDataACK(socketInfo &socket, PacketHeader ACKPacket, char* logfile)

This function is used to receive ACK packets after sending data. I chose to not have this function be a generic getACK function to be used with getting the START and END packet ACKs because I wanted to be able to further examine the ACK packet received. This function differs from the getSTARTENDACK because it returns the packet it received rather than a Boolean indicating that a packet was properly received. This allows me to keep the logic related to rUDP inside of my rUDP send function. This function takes in the socket information needed to receive data into that socket, an empty ACK packet to receive data in to, and the log file for logging.

Function: rUDPSend(socketInfo &socket, char* windowSize, packetTracker &tracker, char* logfile)

This function is the meat and potatoes of the rSender.cpp file. This function is where all the sending of data happens, as well as the ensuring of data being sent reliably. This function takes in the socket information for the socket to send out of, the size of the window the sender should use, the packet tracker to track what packets have been ACKed, and the logfile for logging purposes. My rUDP send function uses an ‘infinite’ loop with a nested loop that iterates through the window which works through sending all the packets in my tracker’s unACKed packet map (which was filled prior to this function being called with the readFileInToTracker function). I achieve reliable sending by first sending all the packets in the window size, then collecting all available ACKs while tracking the highest ACK seqnum I receive. The sender then updates the window accordingly, then updates my ACKedPackets map within my tracker, then updates my window before repeating. I break out of the encompassing infinite loop when I confirm that all packets have been ACKed. This function is lengthy and has the capacity to be refactored, however I liked having all 'reliable' logic inside one function for the purposes of this project.

Function: main(int argc, char* argv[])

This function is where execution starts for the sender. It is here that I call many of the previously mentioned functions. This is where I retrieve the inputting arguments, set up the socket, and have simple logic for the START, END, and DATA packet sending and the related ACK receiving. This is where I ensure that I send the END packet with the same seqNum used for the START packet. Additionally, this is where I will resend either the START or END packets if I do not receive an ACK for them in the timeout of half a second.

File: rReceiver.cpp

This source code file contains code to act as a receiver with application layer functionality to ensure data is received reliably using UDP. This file works in conjunction with rSender.cpp.

Structure: struct args

Like the args structure in rSender.cpp, this structure is used to hold the arguments inputted by the invoker of the file. Having them contained within a structure is convenient for organization and later passing of argument variables.

Structure: struct serverSocketInfo

This structure contains the necessary data about the server socket to allow for port binding between the sender and receiver sockets. This structure contains a socket file descriptor and the server address. The server socket information is necessary because it indicates what socket to receive from. Like the other socketInfo structures, it is beneficial to have the related data bundled in to a socket so it can be sent as a single argument to future function calls where I receive data from the server.

Structure: struct clientSocketInfo

This structure is almost identical to the serverSocketInfo structure as well as the socketInfo structure in the sender. This one differs because it has an additional server_len variable that is necessary for receiving data. This structure is used for the same purposes and benefits denoted above.

Structure: struct packet : public PacketHeader

This structure is identical to the packet structure found in the sender file. It extends the provided PacketHeader structure and provides an additional variable of 'data' that is an array that will carry the received data. This packet structure will be used to receive data in to.

Structure: struct packetTracker

This structure is similar to the packet tracker structure found in the sender file in the way that it uses maps for automatic ordering by keys. This is even more important on the receiver side because part of the rUDP specifications is to handle out of order packets. By putting packets I ACK in to map with a key that is the sequence number, I am confirming that when I write from that map in order by key in to a file that the data will be in order. This structure also contains a bufferedPackets map that I use to store packets when packets are received out of order. I later read from the bufferedPackets in to the ACKed packets.

Function: retrieveArgs(char* argv[])

This function is practically identical to the retrieveArgs function found in the sender file. This one is adjusted to accept the expected arguments used to invoke the rReceiver file. Like the sender function, this one also expects that all arguments are provided and in correct order. It also bundles those arguments inside of my args structure which is then returned from this function.

Function: setupServerSocket(char* portNum)

This function is similar to the socket setup function in Sender.cpp. This function will create and initialize the variables in my serverSocketInfo structure. It sets up the connection using UDP using the inputted port number. Most importantly, it binds the newly created socket, allowing us to later receive data from the binded socket. It returns my serverSocketInfo structure for later use whenever it is needed to receive from that socket.

Function: setupClientSocket()

This function is rather simple as all that is needed from the client socket is the length of the address. This uses my clientSocketInfo structure to hold this information. The client address is later used whenever receiving data.

Function: createACKPacket(int seqNum)

This function does what it says it does. It will use the provided PacketHeader structure to create an ACK packet. This function takes in a seqNum so it can be used for creating an ACKPacket for all types of received packets (START, DATA, and END packets). This function initializes the length and checksum variables of the header to 0, and a hardcoded type of 3 to indicate an ACK packet. It returns the created ACK packet.

Function: writeToLogFile(char* logFilePath, string type, string seqNum, string length, string checksum)

This function is identical to the writeToLogFile function found in rSender.cpp. It takes in the path to the log file and the necessary variables to be written to the log file as dictated by the project specifications.

Function: receiveStart(serverSocketInfo &serverSocket, clientSocketInfo &clientSocket, PacketHeader &STARTPacket, char* logFilePath)

This function receives the start packet from the sender and returns whatever packet was received at that time. When this function is invoked, if there are no packets lost, there should be a START packet incoming. However, there is a possibility that it is an END packet from the previous session. This is why I chose to return the packet itself so I can examine it and decide where execution should go from there. This function also writes to the log file. Using blocking sockets allows me to sit and wait at this function to keep the receiver open.

Function: sendACK(serverSocketInfo &serverSocket, clientSocketInfo &clientSocket, PacketHeader &ACKPacket, char* logFilePath)

This function does exactly what it says. It is my function used to send any and all ACKs, whether for STARTs, DATA, or ENDs. This function takes in the provided PacketHeader structure which acts as my ACK packet, requiring me to build the ACK packet before invoking this function. This gives me the liberty to set the seqNum to the correct value before calling this function. I also write to the log file whenever I successfully send an ACK.

Function: peekAtNextPacket(serverSocketInfo &serverSocket, clientSocketInfo &clientSocket)

This function uses the MSG_PEEK flag for the recvfrom() function in order to look at what type of data the next incoming packet contains, but it doesn't actually take that data out of the receiving pipeline. This is beneficial in the face of a lost ACK packet for a START packet. If I send an ACK packet for a START packet and that ACK packet is lost, I can't move the rReceiver.cpp straight to a 'receive data' state. I need to ensure that the next packet is in fact a data packet before changing states, which also confirms that my ACK packet was received. This function allows me to do that.

Function: ensureSTARTACKReceived(serverSocketInfo &serverSocket, clientSocketInfo &clientSocket, PacketHeader &STARTPacket, PacketHeader &ACKPacketForSTARTEND, char* logFilePath)

This function works in conjunction with the previously defined peekAtNextPacket() function. After I send my ACK for the received START packet, I need to ensure that that ACK packet was not lost. I do this by peeking at the next packet. If the next packet is a data packet, I know that the ACK was received. However, if the peeked at packet is a start packet, then I know that the ACK was lost, causing the sender to resend it's start packet. I then send another ACK for this new START packet. I do not use the same methodology in the face of a lost ACK for an END packet, as it create race conditions for me that created unpredictable outcomes. I explain how I handle lost ACKs for END packets in the 'would you do anything differently' section.

Function: rUDPReceive(serverSocketInfo &serverSocket, clientSocketInfo &clientSocket, char* filePath, char* windowSize, packetTracker &tracker, char* logFilePath)

This function is where my reliable UDP receiving is done. Like the rUDPSend() function in rSender.cpp, this function is lengthy but I felt it made sense to have all rUDP functionality in side of a single function for simplicity. This function also uses an 'infinite' loop, as there is no hard fast way to know how many iterations it will take to receive data/send ACKS in the face of possible lost packets. In this 'infinite' loop, I try to receive all packets I can for the given window size, writing to the log file as I receive packets. I check the packet types, ignoring new START packets, breaking out of the loop when receiving END packets, and sending a correct ACK back when receiving DATA packets. The meat and potatoes of rUDPReceive occurs when I receive DATA packets. This is where I ensure the checksum is correct, and the decide what to do depending on the received packet sequence number. I drop packets that are outside of the window size but send an ACK with the next expected sequence number otherwise. This is also where I make use of my packetTracker struct to track what packets I have ACKed, as well as buffer packets that are not the next expected sequence number. Last, I update the window to begin at the highest sequence number in my ACKed packets map.

Function: putBufferedPacketsIntoACKedPackets(packetTracker &tracker)

This function is the product of a small refactor to help split things up. This does what it says. It takes in my packetTracker that contains the ACKed packets and the buffered packets. It will then read from the buffered packets and put them in to the ACKed packets. Since I am using maps with a key that is the sequence number, I am ensuring that the data stays in order and I do not have duplicates.

Function: writeDataToFile(char* filePath, packetTracker &tracker, int &fileNum)

This function writes all the received data to the file. I do not write to the file until all data has been received (including the END packet). Importantly, this function takes in to account the correct file name that I should create and write to (File-i.out), where I increment the file number (i) after each execution of receiving data.

Function: main(int argc, char* argv[])

This is where execution starts for rReceiver.cpp. This is where I retrieve the inputted args from the command line, set up the server and client sockets, then enter the waiting for start state. The simple logic found in my main() will keep the receiver inside the wait for start state even after all data has been received and the sender has closed its connection. This main function calls many of the previously mentioned functions with logic in between to ensure properly ordered execution of sending and receiving packets.

Question 2: Did you finish the entire Project? If not what was complete and what is incomplete?

Project Completely Finished

Question 4: Did you complete the extra credit?

N/A – I would have loved to, but unfortunately Capstone has required my attention and I think I am comfortable with where my grade is currently at for this course.

Question 4: What did you learn doing this project?

A ton! Recreating the windowing reinforced my understanding of methods that TCP implements for reliable sending and receiving of data. There were multiple aspects of our reliable UDP methodology that I didn’t understand at first, such as sending all the packets in the window before trying to receive any ACKS. This certainly wrinkled out any misunderstandings I had. Additionally, I learned some more about socket programming, such as the difference between blocking and non-blocking sockets, as well as other flags that can be used such as the MSG_PEEK flag and the timeout option for sockets. I also learned new things about the fstream library, such as its ability to create new files, and to stream binary files rather than the text files I have used it for in the past.

Question 5: What was the hardest part of this project?

That is a tough question. There were many tough aspects that I struggled through. Something I spent a great deal of time on was conceptualizing how to handle blocking sockets and the timeout. I let myself struggle with this concept probably for too long before asking on piazza (piazza @121). There was of course difficulty debugging, even with Wireshark, that I overcame by using the console for cout-ing instead, which I would argue was a better way to debug. The rUDP logic itself wasn't too complex and was laid out well in the project specs, there were just some strange edge cases that needed to be figured out. I also struggled with how to handle lost ACK packets that were sent for the START and END packets (piazza @179).

Question 6: If you were to start this project again, would you do anything differently?

Yes! There is some inconsistency in the way I handle lost ACKs for STARTs and ENDs. Even though I white boarded and planned how execution should flow between the sender and receiver before writing any code, I still managed to get some spaghetti code here and there. I could have done better planning if I knew what issues I would run in to with losing ACK packets for START and ENDs. Overall, I am not incredibly proud of my code structure as there are some inconsistency and room for more refactoring for easier to comprehend functions.

Although, I am glad that I started the day it was released! Even then, between Capstone and other courses, I am finishing this writeup only with a couple days to spare.

Question 7: Do you have any suggestions for improving this project?

For the most part, I thought things were clear in the project specs with a good amount of unknown sprinkled in. It would have benefited me greatly to learn earlier on that I should use blocking sockets, as well as the best way to handle a timeout and lost ACKs for START and ENDs. Although, I understand that you do not want to give too much information in the specs and there needs to be some things for the student to figure out on their own.