COMP 7005 - Final Project

Data Communication Principles

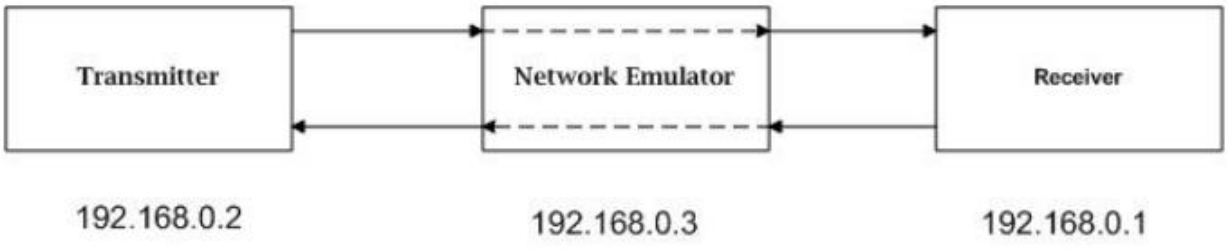
Kuanysh Boranbayev, Parm Dhaliwal

December 2, 2019

Table of Contents

Overview

Final Project Objective is to design and implement a basic Send-And-Wait protocol simulator. The protocol will be half-duplex and use sliding windows to send multiple packets between to hosts on a LAN with an "unreliable network" between the two hosts. The following diagram depicts the model:



Mission

Protocol can be written using any language of the students' arsenals. Minimum of three components: transmitter, receiver, and network emulator must be implemented. Usage of codes from previous assignments are suggested.

Designing an application layer protocol must be on top of UDP. The protocol should be able to handle network errors such as packet loss and duplicate packets. It is necessary to implement timeouts and ACKs to handle retransmissions due to lost packets (ARQ).

The network emulator will act as an unreliable channel over with the packets will be sent. This means that transmitter will send the packets to the network emulator which in turn will forward them to the receiver. The receiver in turn will send ACKs back to the transmitter via the network emulator.

Network emulator implementation must include a "noise" component which will randomly discard packets (and ACKs as well) to achieve a specified bit error rate. This can be specified as part of command line arguments.

One side will be allowed to acquire the channel first and send all of its packets. An End of Transmission (EOT) will indicate that it has completed sending all of its packets, after which the other side can start sending packets.

The following structure depicts a suggested packet format:

```
struct packet
{
        int PacketType;
        int SeqNum;
        char data[PayloadLen];
        int WindowSize;
        int AckNum;
}
```

Constraints

The basic protocol is Send-And-Wait; however, it is a modified version in that it will use a sliding window to send multiple frames rather than single frames. Implement a timer to wait for ACKs or to initiate a retransmission in the case of a no response for each frame in the window.

Window will slide forward with each ACK received, until all of the frames in the current window have been ACK'd.

Both the transmitter and receiver will print out ongoing the session as simple text lines containing the type of packet sent, type of packet received, the status of the window, sequence numbers, etc.

Application will maintain a log file at both the transmitter and the receiver. This can be used for both troubleshooting and for validating the protocol.

Network module will take arguments such as the BER (Bit Error Rate), average delay per packet, and will also have a configuration file specifying IP addresses and port numbers for the transmitter and receiver.

Introduction

The program is written on C and compiled with the current stable version of GCC (9.2). The program consists of three main parts: transmitter (*client.c*), network (*emulator.c*), and receiver (*server.c*). Also, required supporting extension files are *packet.h, handlers.h*, and *handlers.c*.

packet.h: consists of user defined packet structures

handlers.h: consists of declared function prototypes that handle the most of the program actions; such as printing output to a console or to a file, error checking, round-trip time calculations, total data packets, etc.

handlers.c: user defined functions that handle the code for each of the function prototypes.

Packet structure

```
struct Packet {
        int PacketType;
        unsigned int SeqNum;
        unsigned int AckNum;
        int WindowSize;
        char data [PAYLOAD_LEN];
        int last;
        int re;
};
```

PacketType: numeric flag that shows the type of the packet; The following types are implemented in designing the application:

1 – SYN, 2 – SYNACK, 3 – ACK, 4 – DATA, 5 – NAK, 8 – EOT, 9 – TO (Timeout)

SeqNum: a sequence number used to number data packets.

AckNum: a numeric field to indicate the previous data packet being acknowledged and the next expected sequence number.

WindowSize: this field is manipulated so that it will have two purposes.

1. It's used at the handshake session to exchange information regarding the total number of data packets and how many are allowed to transmit at a time.
2. After the handshake session is established, it's value will indicate how many packets are in-flight.

data: an array field that stores only characters or substrings at a time when sending the string data. Note: the default payload length for the data field will be **500 bytes**.

re: a boolean field that will indicate whoever receives the packet containing 1 for this field has to expect more packets before sending the next data packets or ACK packets.

last: a boolean field that will indicate the last packet. For example, whoever received re=1 has to know when to stop waiting for more packets but to start transmitting next packets.

Being inspired by the TCP protocol system, the application designed so that it somewhat resembles similar design. Overall application structure has three main phases or sessions.

1. Establishing the connection with the host
2. Data transmission
3. End of transmission

Establish connection

Establishing the connection with the host also known as the "*handshake session*" is basically agreement between two hosts before exchanging the data. To invoke an analogy, think of manufacturer-retailer relationship. Retailer needs items that manufacturer produces in order to distribute. However, manufacturer has to know how much item to produce. It can overproduce or underproduce. So, before producing and distributing, both of them need to agree on terms. Just like that, transmitter and receiver have to come to terms. Therefore, the handshake session will give both of them necessary information before proceeding to data transmission phase. First off, transmitter will create SYN packet that will contain **no data** but just a unique **sequence number**, an **acknowledgement number**, and **window size** which will hold the total number of data packets needed in order to send a single data file. Once, receiver receives the SYN packet, it can examine and generate its own response packet called SYNACK which will contain **a sequence number** containing the same value that SYN packet had which help the transmitter to know that it is not some random SYNACK packet but the one responding to its sent SYN packet; an **acknowledgement number** which does not serve any purpose than indicating the next expected packet's sequence number; **window size** that will contain depending on receiver's wish to receive the maximum number of packets at a time; and of course, no data as we are still on bargaining phase.

Data transmission

Once, the SYNACK reached the transmitter, it can start creating the first DATA packets. But how much to send? As mentioned above from the receiver side, SYNACK window size will contain the value of desired number of packets.
1st DATA packet will have following entries:
A sequence number containing the SYNACK's acknowledgement number;
An acknowledgement number which will be tricky as we are not expecting immediate ACKs assuming SYNACK's window size value is larger than 1. So, to make it simple, we will only store the maximum desired number of packets which is SYNACK's window size value.
Now, window size field's purpose start changing to indicate the number of packets in-flight. So, for 1st DATA packet, it will be 1.
Data field stores the first five sequential characters of the entire data file string.
2nd DATA packet will have following entries:
A sequence number containing the 1st DATA packet sequence number added to SYNACK's window size value.
An acknowledgement number is going to follow the same procedure which will be 1st DATA packet acknowledgement number plus SYNACK's window size value. So, it basically gets the twice the value of the 1st DATA packet's acknowledgement number.
Window size will store 2 indicating now 2 packets are en route.
Data field will store the next five sequential characters indicating the sliding window.

From here, the formula must become clearer. To summarize the creation DATA packet entries

$$DATA.SeqNum = \sum_{i=1}^{n} DATA[i-1].SeqNum + SYNACK.AckNum$$

$$DATA.AckNum = \sum_{i=1}^{n} DATA[i-1].AckNum + SYNACK.WindowSize$$

$$DATA.WindowSize = \sum_{i=1}^{n} i, \qquad n - total\ desired\ number\ of\ packets$$

$$DATA.data = \sum_{i=1}^{m} file\_contents[i,m], \qquad m - an\ offset, set\ to\ the\ default\ data\ length$$

After the desired number of DATA packets have been sent and reached to the receiver, the receiver starts creating ACK packets uniquely for each arrived DATA packet. Additionally, the receiver also has to respect the terms and wait until the last DATA packet arrived.

So, the process of creating each ACK packet is as follows:

$$ACK.SeqNum = \sum_{i=1}^{n} DATA[i].AckNum$$

$$ACK.AckNum = \sum_{i=1}^{n} DATA[n].SeqNum + SYNACK.WindowSize$$

$$ACK.WindowSize = \sum_{i=1}^{n} DATA[i].WindowSize, \qquad n - total\ desired\ number\ of\ packets$$

End of Transmission

Indication of the successful entire data file transmission has to be implemented despite both side's knowledge of the amount of data packets. The structure of the EOT packet is similar to ordinary DATA packet: sequence number, acknowledgment number, and window size are adopted by the same procedures respectively. The only difference is no data carried by EOT packet. Its sole purpose to indicate that the transmitter is cutting its connection with the host.

As of now, the application is designed as half-duplexed and data packets are being transmitted in one and ACK packets are transmitted in other way.

Network Emulator Design

In the above situations, we considered the transmission is occurring on reliable channel. But, in reality, we need to consider situations such as sudden packet drops, delays, receiver is offline during data transmission, and so on.

So, designing network emulator that simulates random packet drops and delays packets are necessary to experience and learn real situations.

The simplest design of the network emulator is that it just forwards the packet received. So, for the sake of simplicity, we could design a single socket on UDP. The implementation was not hard. The problem occurred when both sides are transmitting and listening at the same time. Remember, the UDP is designed so it does not need to be ACK'd so transmitting from both sides at the same time, emulator gets congested and crashes inevitably. So, when simulating real packet losses and delays, both sides have to act accordingly without disrupting the flow control.

A delay feature has been implemented with the use of creating additional threads for the session. After the delay function has finished its purpose, it will be destroyed. The lack of knowledge in C multithreading and short deadline forced us to use the tools and knowledge available at the moment.

So, the reasonable solution is the implementation of two sockets for each end. With two sockets, when one end crashes we would immediately figure out the source of the problem.

After implementing the network emulator, we recall the initial three phases of application:

1. Establish connection

   Above we assumed no disruptions, no packet losses, and no delays. Now let's assume at least the last two situations.

   a) Transmitter sends SYN to network emulator which will be dropped not reached to the receiver.
   Transmitter has to resend the SYN at some point. So, we implement a timeout for Transmitter and set to 3 seconds which is reasonably fine for now.
   The same situation can happen from receiver side when sending SYNACK in response to SYN. So, we implement 3 second time-out here too. But hold up, this means there is a chance that right before sending the SYNACK, the receiver might receive the DATA meaning successful previous SYNACK. So, we need to increase the receiver's timeout a little longer than 3 seconds, 5 seconds.
   Transmitter resends SYN after timeout and this time, the packet can still be dropped but even if it was dropped, transmitter will have enough time resend SYN.
   And receiver when resending SYNACK will have enough time wait for SYN again in case of previous SYNACK drops.

b) Implementation of delay drastically changes the constant timeouts for both sides. So, the we need the actual round-trip time and use it to update the timeout value for the next transmission. Adapt and reuse!

2. Data transmission
   a) Packets drops can occur more than once. So, it can be tedious when dealing with not only single packet drops but multiple DATA or ACK packet drops. Fortunately, both sides share knowledge of the number of packets are in-flight which will be helpful. After timeouts, receiver will know how much packets and which packets are missing. Receiver sends ACKs only for those received packets and update for extra timeouts. Once, ACKs reach the transmitter, transmitter has to resend the corresponding not ACK'd DATA packets, fast-retransmission is implemented. Fast-retransmission means not the speed but the order in which DATA packets are transmitted. So, fast-retransmission DATA packets will have priority in the next data transmission over the next slide window packets. So, if the situation keeps repeating, the transmitter will keep resending the not ACK'd DATA packets until all of them ACK'd. Once, all current DATA packets have been ACK'd, transmitter proceeds to the next window DATA packets.

      Receiver can experience similar situation when some ACKs are being dropped. So, to fix this we cannot implement the same fast-retransmission algorithm but we need to implement slow-retransmission meaning ACKs have to be sent after the recent ACKs in order to not disrupt the flow. As the recent ACKs can occupy the full window size, we need to inform the transmitter somehow that the receiver is missing some unordered DATA packets. The solution is to implement additional boolean flag "re" that indicate transmitter or receiver to wait for more packets before sending the next window. And, transmitter or receiver has to know for how much more to wait. So, we implement another boolean flag indicating the last packet of the retransmissions.

3. End transmission

As EOT packet does not require to be ACK'd, but we need to implement it special case. For instance, transmitter sends the last DATA packet and sees a window space for EOT packet and includes it for the last transmission. As mentioned above, EOT means transmitter is cutting its connection with the receiver without knowledge of successful delivery of the previous packets. So, we need to ensure that transmitter keeps waiting for ACKs until the last DATA packet have been ACK'd. After the previous DATA packets have been ACK'd, transmitter resends EOT just in case the previous one is not delivered and closes the connection.

Diagrams:

Handshake session:

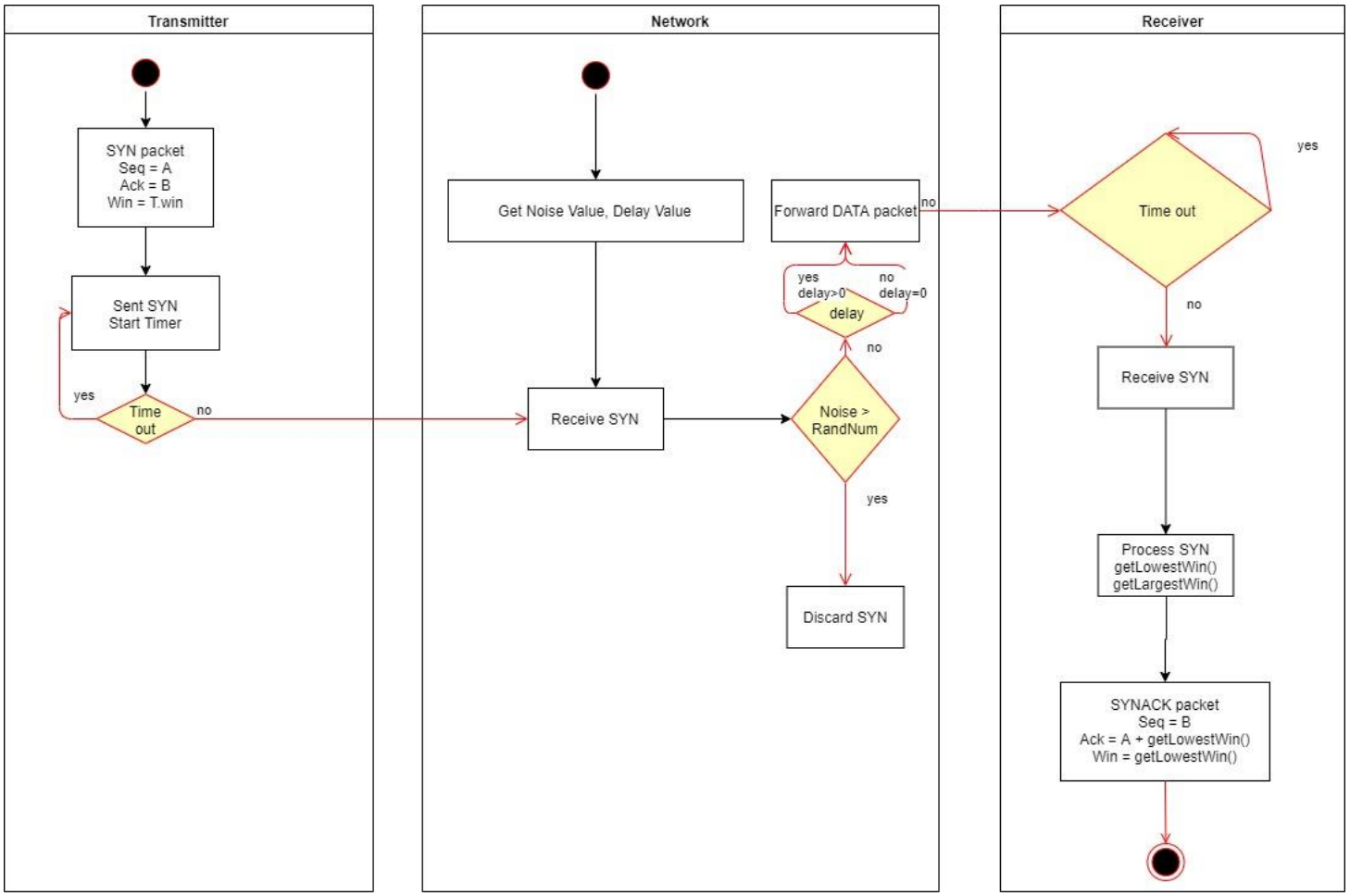


Figure 1. a. SYN packet is being sent to receiver

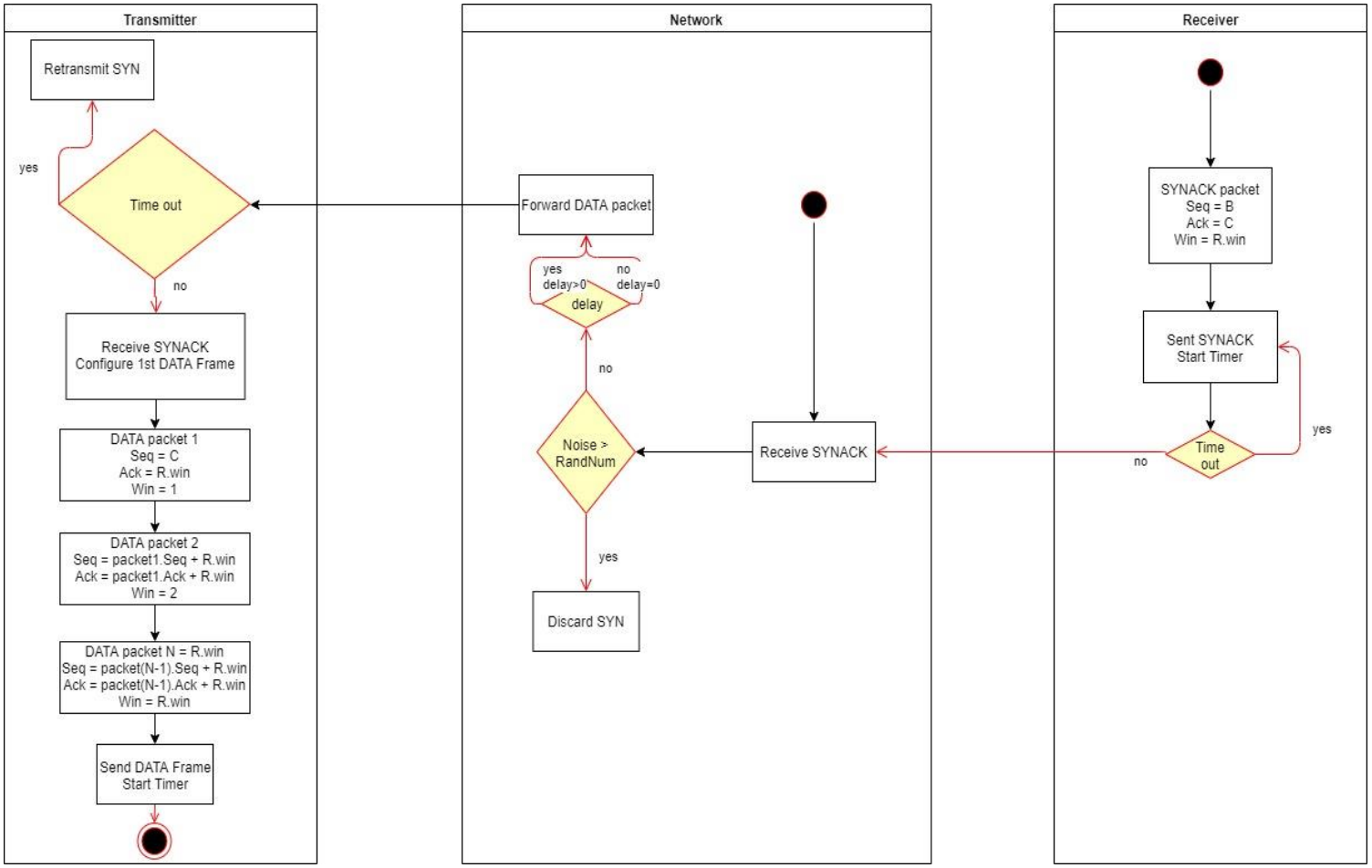


Figure 1. b. SYNACK packet is being sent back to the transmitter

Data transmission session

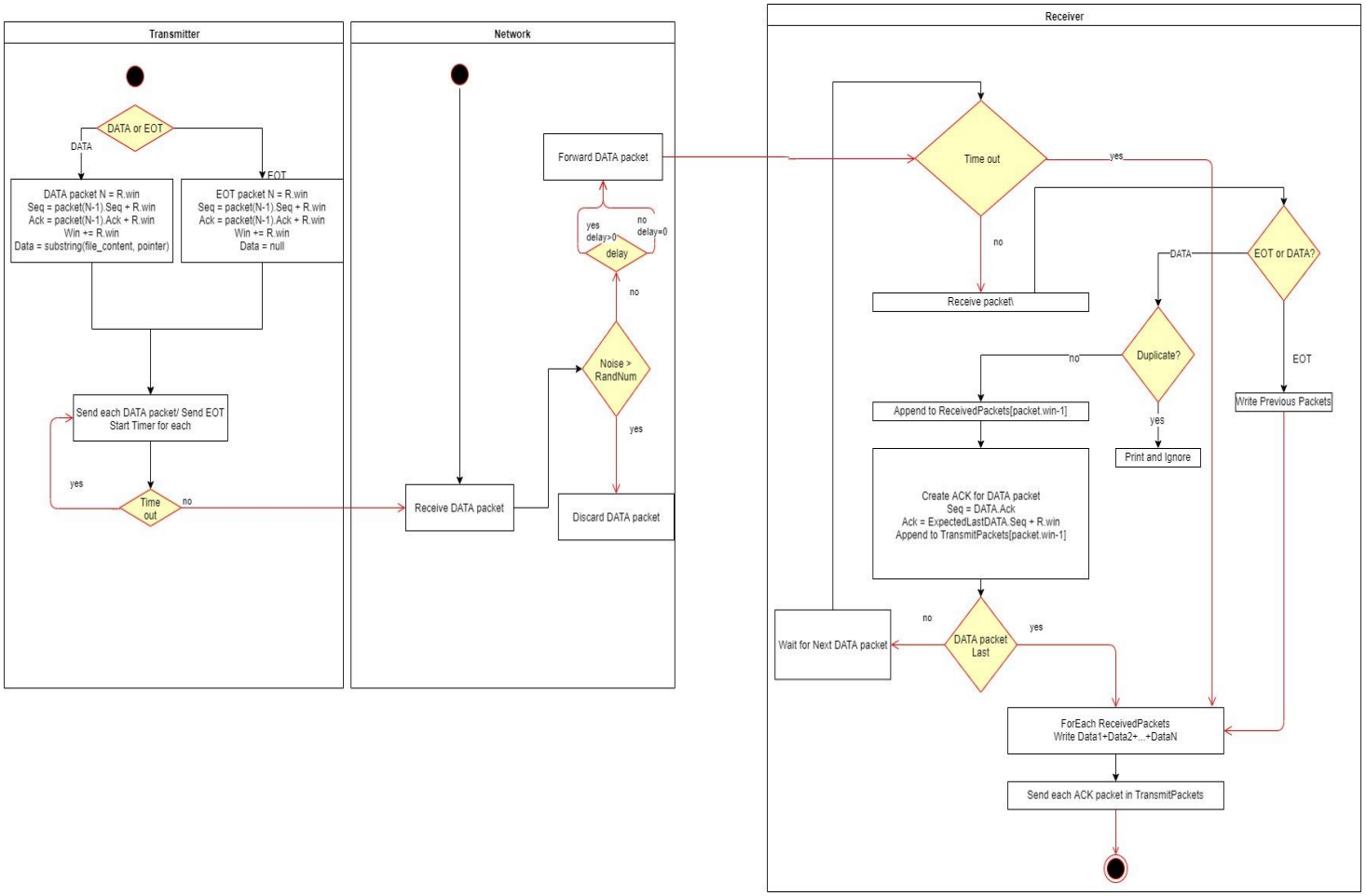


Figure 1. c. DATA or EOT packet is being sent to the receiver

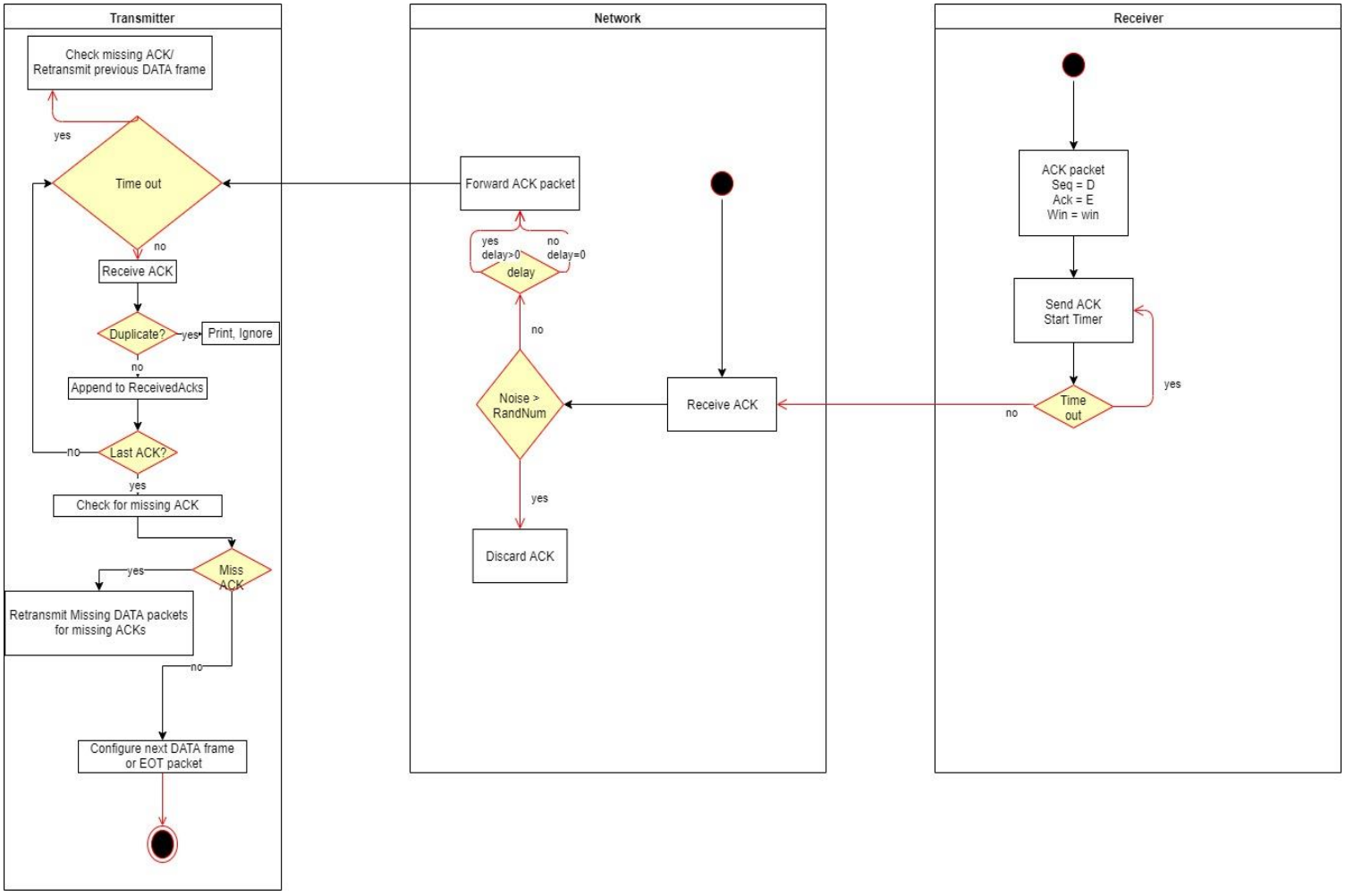


Figure 1. d. ACK packet is being sent to the transmitter

Modules

Transmitter (client.c)

Requires the following arguments in program execution:

1) A filename followed by "-f" flag. At the moment, the program takes only text files, e.g. -f data/send.txt.
2) Host IP address (or the emulator IP) with no flag. E.g. 192.168.0.1
3) Host Port number. E.g. 7000

A program behaves as a client end point that tries to establish a connection with the server before transmitting the data itself. At first, client binds its local IP address to the network and configures the SYN packet. SYN packet will have a PacketType, a SeqNum, an AckNum, WindowSize, and data as a predefined array of size 5.

PacketType: an integer type; a packet can have one of following types:

1 – SYN, 2 – SYNACK, 3 – ACK, 4 – DATA, 5 – NAK, 8 – EOT, 9 – TIMEOUT

SYN: flags that a client wants to establish connection with a server, and indicates the number of packets that client wants to transmit to the server.

SYNACK: a response packet to the previous SYN packet. It flags an approval by the server for the establishing the connection with the terms.

ACK: an acknowledgment for a DATA packet.

DATA: a packet that contains a portion of a whole data.

NAK: a negative acknowledgement packet that indicates that the server is overwhelmed by arrival of more than 3 unordered packets.

EOT: an end of transmission packet that flags the server that the client is finished transmitting the data. It does not require an acknowledgment packet. So, a client will terminate the connection once the all DATA packets have been acknowledged.

TIMEOUT or TO: a time out sequence trigger, usually contains previously transmitted packet info (SYN, SYNACK, ACK, DATA or NAK)

SeqNum: an unsigned integer type; as every packet will have its own sequence number, it will be regarded as an id for a packet. For example, 1st data packet will have a sequence number of 100, then the 2nd will have 1st packet's sequence number plus an established window size. So, each of the packets in sequence will differ with only window size amount.

AckNum: an unsigned integer type; acknowledgement number's like a sequence number, but it will be used for later ACK packet's sequence number for identifying an acknowledged packet.

WindowSize: an integer type; window sizes will typically be used at the start of the session to establish the number of packets can derived from the data and the number of packets allowed to send at once. It will also be used to track the packets in sequence and estimate for the next coming packet sequence number.

Data: an array of characters or a string; typically holds the small fraction of the data. At the end, all the received DATA packets will be used to concatenate each packet's data to each other.

Then, sends SYN packet and initiates a timer for time-out sequence. In case of, no response after SYN packet sent in duration of the given time-out, the client will retransmit the SYN packet with no changes.

Handshake session:

A client.c initiates a handshake by sending the first SYN packet to a network which network will forward to a server. In case of, no response from the server after given time period, a client will retransmit the SYN packet.

Once, SYN packet is arrived at a server's side, server will have knowledge of the total amount of data packets to be sent and the next coming DATA packet after sending SYNACK packet. Depending on an amount of data, server will calculate the advertising window size for a client. And, server sets a SYNACK packet's SeqNum to SYN's AckNum. As an example,

SYN: SeqNum = 100, AckNum = 225, WindowSize = 125, data = null

Then, SYNACK is going to be

SYNACK: SeqNum = 225, AckNum = 116, WindowSize = 16, data = null

Once, client receives the SYNACK, it will record the round-trip time and updates the time-out period. After, client configures the DATA frame with following example structures:

DATA1: SeqNum = SYNACK.AckNum (116), AckNum += SYNACK.Win, WindowSize = 1, data = "substring1"

DATA2: SeqNum = DATA1.SeqNum + SYNACK.Win, AckNum += SYNACK.Win, WindowSize = 2, data = "substring2"

DATA3: SeqNum = DATA2.SeqNum + SYNACK.Win, AckNum += SYNACK.Win, WindowSize = 3, data = "substring3"

The last packet N for the current frame which is N = SYNACK.Win:

DATAN: SeqNum = DATA(N-1). SeqNum + SYNACK.Win, AckNum += SYNACK.Win, WindowSize = N, data = "substringN"

After sending, each of the packets above, the client will record the system time for each packet sent. Once, ACK for each of the packets sent is received, the client will calculate the delay for each packet.

Network Emulator (*emulator.c*)
The program will take the following arguments:

1) Noise: followed by"-n", an integer between 0 – 100, meaning the noise level in percentage. E.g. -n 40 will be 40% chance of randomly discarded packets in following transmissions.
2) Delay: followed by "-d" flag, an integer that will be translate into milliseconds; e.g. -d 5 will be 5ms
3) Configuration file name: followed by "-f", a string that will be used to read the client's and server's IP addresses and associated ports. E.g. -f config.txt

Network Emulator is designed so it acts like an unreliable channel over with the packets will be sent. This means that transmitter will send the packets to the network emulator which in turn will forward them to the receiver. The receiver in turn will send ACKs back to the transmitter via the network emulator.

Noise

If network noise argument is set to a value that is not default 0, the network will start randomly generate number. The random number will be generated in a range from 0 to 100. If the random number is larger than the noise number, the network will not drop the packet. If it is lower than the noise number, the network will drop the currently received packet.

Delay

Delay is given in a millisecond. If network delay argument is set to a value that is not default 0, the network will create a new thread to handle the packet.

Receiver (*server.c*)

The program will take no arguments as it will act as a declared end point for all clients. And, all packets will be delivered from the network emulator.

The program listens until it receives a SYN packet which will be the initializer for establishing the connection and configuring the window size for the next data transmissions. As of now, window size is set for a constant value of 22 packets. After receiving the 22 DATA packets, server will send ACKs corresponding to each of the DATA packets. After transmitting SYNACK packet, server will start timer which will be a little longer than a client's period. The following scenarios will depict the situations after sending SYNACK packet:

a) Server receives DATA packet with expected SeqNum; meaning client received the SYNACK successfully and started transmitting DATA packets accordingly.
b) Server receives DATA packet with not expected SeqNum; meaning client received the SYNACK but the DATA packet arrived in out-of-order. Server will keep the DATA packet, and waits for more DATA packet or the last DATA packet before sending ACKs.
c) Server receives SYN; meaning the client may not receive the SYNACK. Rational choice would be to immediately re send the SYNACK, but server will wait for another little time before resending the SYNACK. In case of during this wait period, server receives DATA packet, server will not send SYNACK but will update its timer for the next DATA packets.
d) Server receives SYNACK; meaning network simulator failed in sending the SYNACK to the right end. Program shuts down.
e) Server receives correct DATA but from the different port; meaning network simulator might have created new socket for the sake of resuming the transmissions.

Instructions

1. Compile source files with GCC (version 9.2 was used in developing the applications)

   Example usage:

```
/# gcc -W -Wall -lm handlers.c client.c -o client
```

```
/# gcc -W -Wall -lm handlers.c emulator.c -o emulator
```

```
:/# gcc -W -Wall -lm handlers.c server.c -o server
```

   Or run provided BASH script that executes commands above and deletes previous log files and previous compiles to speed up the process like so:

```
:/# sh compile.sh
```

2. Next, run the server-side program first:

```
:/# ./server
```

   Note: server does not take any arguments as its designed so that it binds to the local address on default port of 8000 and immediately starts listening for SYNs.

3. Next goes the network emulator. Network emulator requires the server already running and listening beforehand. And, emulator will be running on port 7000 apart from server in case of testing occurring in 127.0.0.1:7000 (localhost).

```
:/# ./e -n 0 -d 0 -f config.txt
```

   Note: emulator requires arguments with values
   "-n" – noise component flag which value can be between 0 – 100 meaning 0 is 0%-bit error rate (BER) and 100 – 100% BER
   "-d" – average delay flag for forwarding each packet. Its value is in milliseconds so 5 means 5ms delay before sending a packet.
   "-f" – a file name flag; the program will read the file if it exists (if it does not, the program will crash); the file must contain two lines of strings: 1st line corresponds to the server-side configurations e.g. 192.168.0.1:8000 – server is running on 192.168.0.1 on port of 8000; and 2nd line must contain client's (sender) info e.g. 192.168.0.15:6000.

4. At last, executing the client will start the chain reaction:

```
./client -f data/send.txt 192.168.0.20 7000
```

   Note: above, the client takes two additional arguments because we are assuming the network emulator is running on 192.168.0.20:7000. If we do not supply the optional arguments to the client program, it will assume that the emulator is running on localhost on default port of 7000. Also, the program will need to read a data file to send.

   After successful executions of above commands, each program will generate its own log files under **logs/** directory with corresponding names e.g. client will produce *client_logs.txt*, etc.

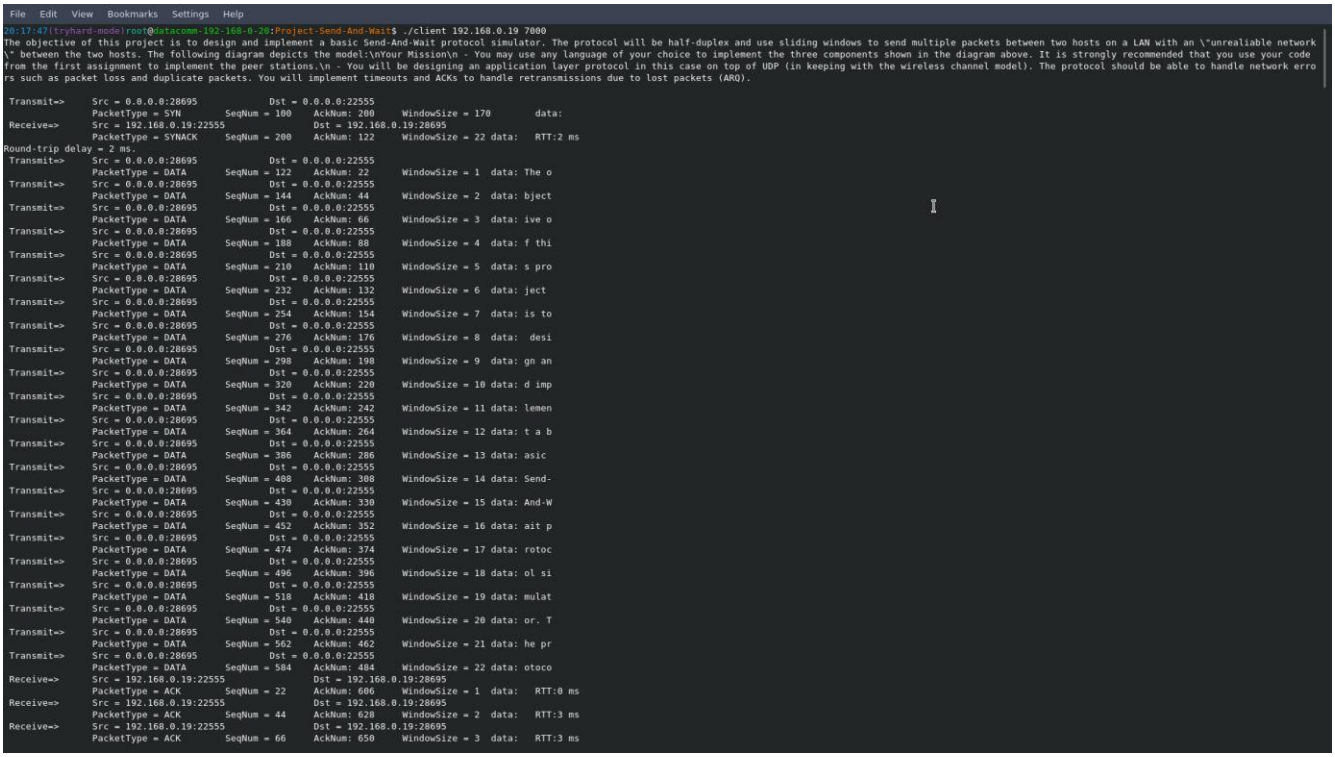
5. Server will store the received file content under the same directory as client **data/** but with different name of *received.txt*.

Tests and Verifications

Note: the programs have been tested and verified with the packets carrying only 5 bytes of data instead of final release's set of 500 bytes. The final implementation did not change the overall architecture of the programs.

1. The test results of executing the programs with the initial 0% noise ratio and 0ms delay.

Client screenshots



Note: the long string at the start of the program is the return from the reading the *data/send.txt* file contents.

Handshake session gives three essential information for both client and server.

Client is sending SYN with acknowledgement number of 200 and window size of 170 (meaning to transfer the whole string of data above, it will require 170 packets each containing only 5 characters or a substring of size 5), and note, no data supplied.

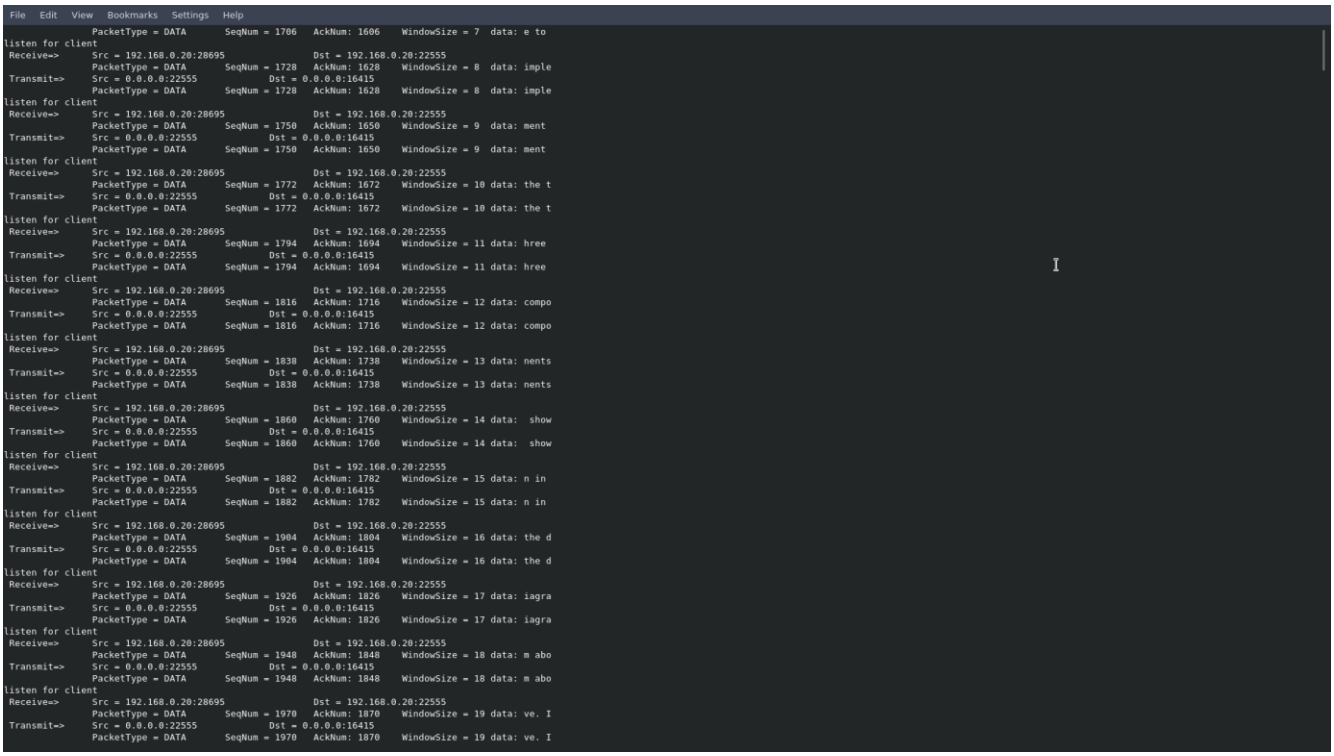
After receiving the SYNACK from server with sequence number of 200 (which you guessed it, the acknowledgement number of SYN packet), and acknowledgement number (which origin will be explained in server side), and different window size of 22 (meaning server is aware of total size of the data but only can accept 22 packets at a time), and again, no data as it is a ACK packet for SYN, client starts sending the first 22 DATA packets one at a time as server requested.

Notice the first DATA packet which has sequence number of 122 (which was SYNACK packet's acknowledgement number), acknowledgement number of 22 (the number is derived from the requested max window size), window size of 1 (meaning there is only 1 packet in-flight, but there are some more space for additional 22 - 1 = 21 packets), and data which contains the first 5 characters (a space is also a character by GCC standards) of the whole string.

Next DATA packet has sequence number of 144 (1st DATA packet.SeqNum + SYNACK.WindowSize), acknowledgement number of 44 (1st DATA packet.AckNum + SYNACK.WindowSize), window size of 2 (2 packets in-flight), and data of "bject" which is next 5 characters of the whole string.

As you can see from the console output above, the client is transmitting SYN and DATA packets and receiving the SYNACK and ACK packets with no loss or delay.

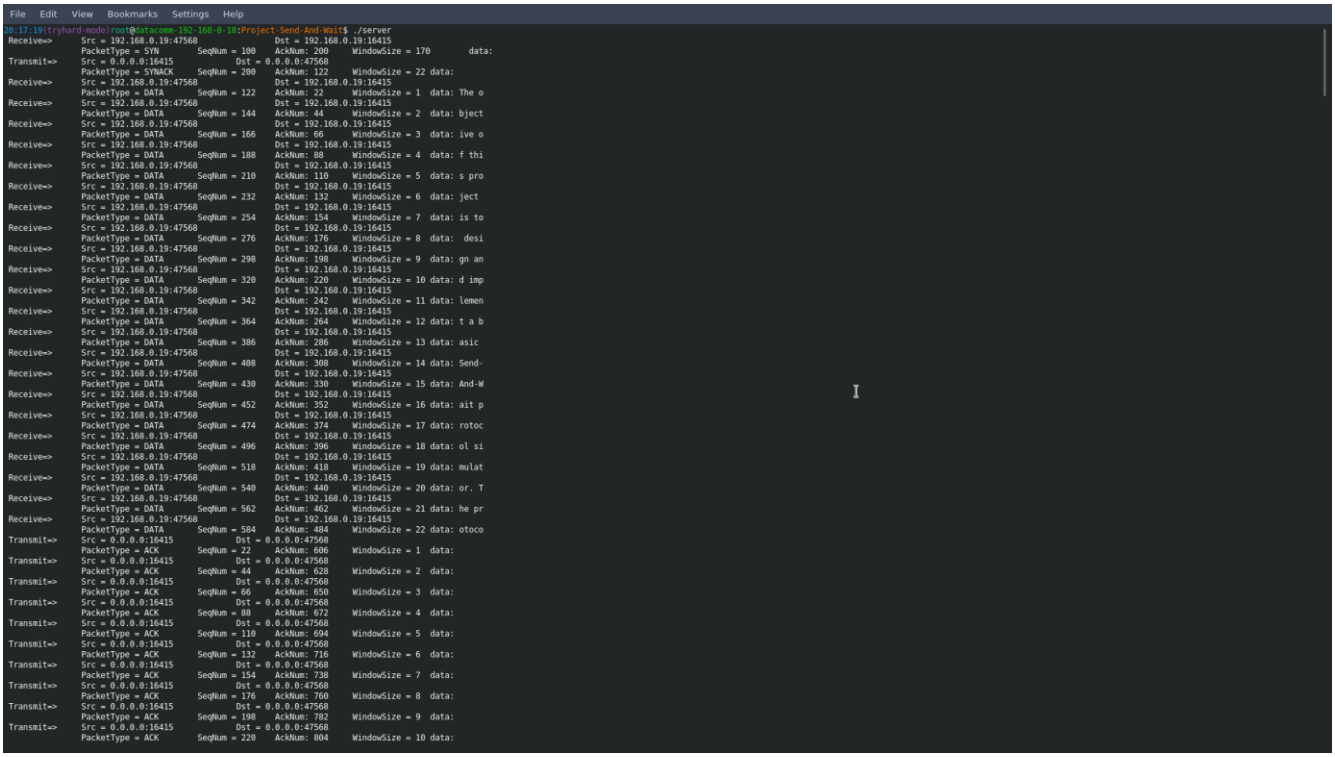
Network Emulator screenshots with the same conditions: 0% BER and 0ms delay



Note: the output above only shows the packets from client-side with corresponding sequence numbers, acknowledgement numbers, window sizes (as the data transmission is already underway here, window sizes indicate the current number of packets on-flight), and data that shows what packet is carrying what substring of the *data/send.txt* content.

As seen above, emulator is forwarding the packets to the server (receiver) immediately whenever it receives a packet from the client.

Server screenshot with the emulator (0% BER and 0ms delay)



Note: server above is receiving the first 22 packets as it previously requested from the client before sending the corresponding ACK packets.

Server creates ACK packet for each DATA packet in the same manner but with different values.

Recall the first DATA packet sent by client, it had SeqNum = 122, AckNum = 22, WindowSize = 1, data = "The o".

So, the 1st ACK is going to be generated as so:

ACK.SeqNum = DATA.AckNum = 22 (so the client knows that the DATA packet has been acknowledged by the server)

ACK.AckNum = the last DATA packet SeqNum + 22 (window size)

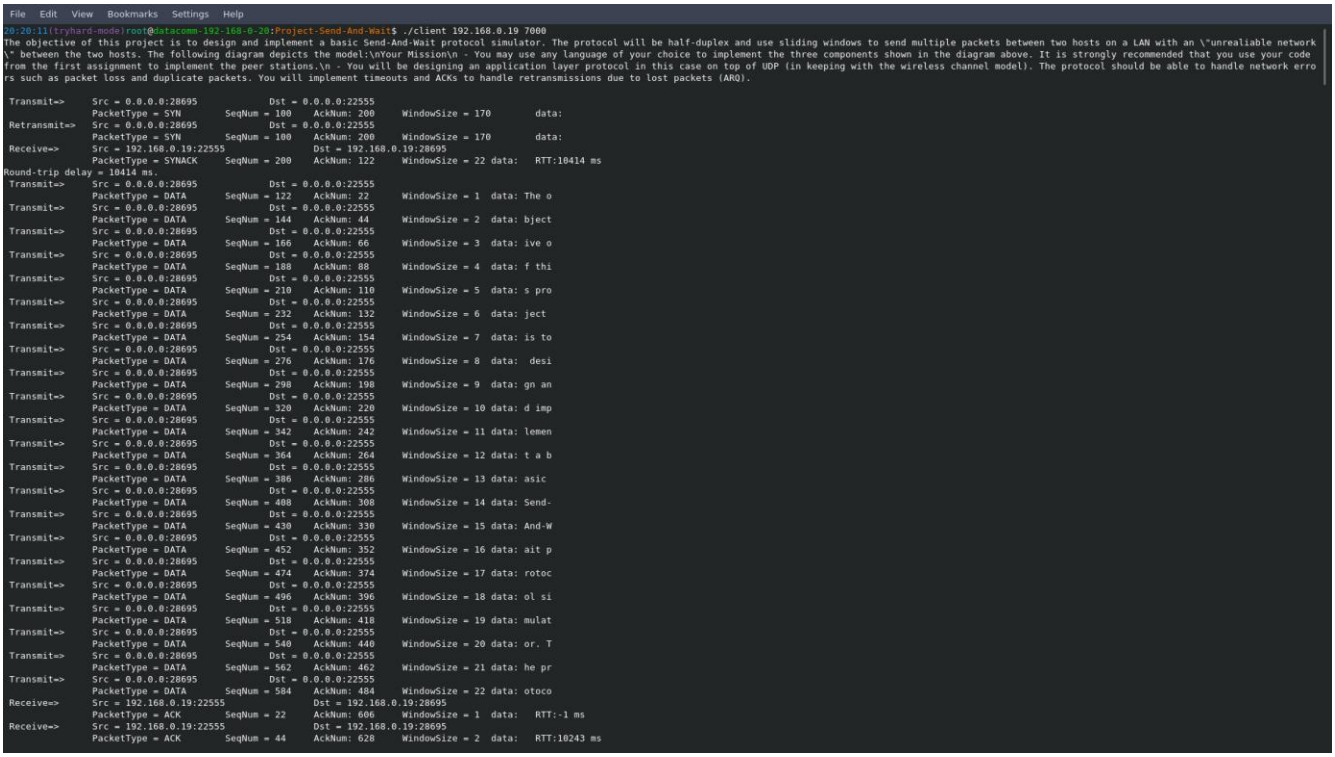
ACK.WindowSize = DATA.WindowSize = 1 (as the ACK will only packet in-flight)

And no data because it is ACK packet for DATA packet.

The rest is of the ACK packets are generated in the same way.

2. The test results of executing the programs with the 30% noise ratio and 0ms delay supplied with the same data.

Client

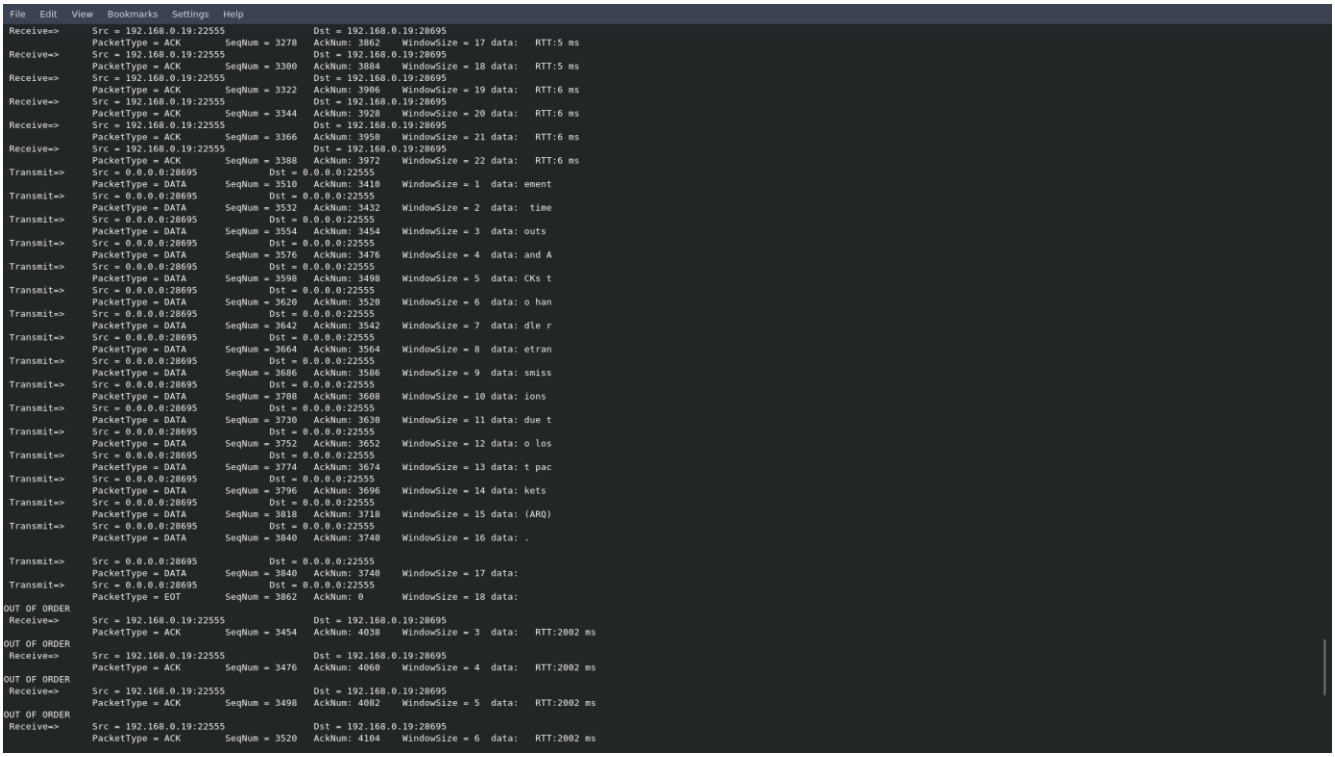


At the beginning, we already see that client experienced a time-out after sending the 1st SYN packet and retransmitted SYN packet again.

Notice the round-trip time which is 10414ms considerably larger than the previous test with no noise. As client's initial time-out value is hardcoded with 10 seconds, it had to wait for 10 seconds before retransmitting the SYNACK again.

After that, no DATA packet is being retransmitted (or at least not now) because the client has to send the 22 DATA packets first before waiting for the ACK packets.

Client-side monitoring continues



Note: client is receiving ACK packets but out-of-order meaning some ACK packets might have been lost in transit.

Although client receives ACK packets out-of-order, it will not ignore them unless it received the same ACK packets twice or more times.

Client will keep the ACK packets and analyze which DATA packets have not been acknowledged, and it will retransmit only those DATA packets.

Network emulator executing with 30% noise ratio or BER and 0ms delay.

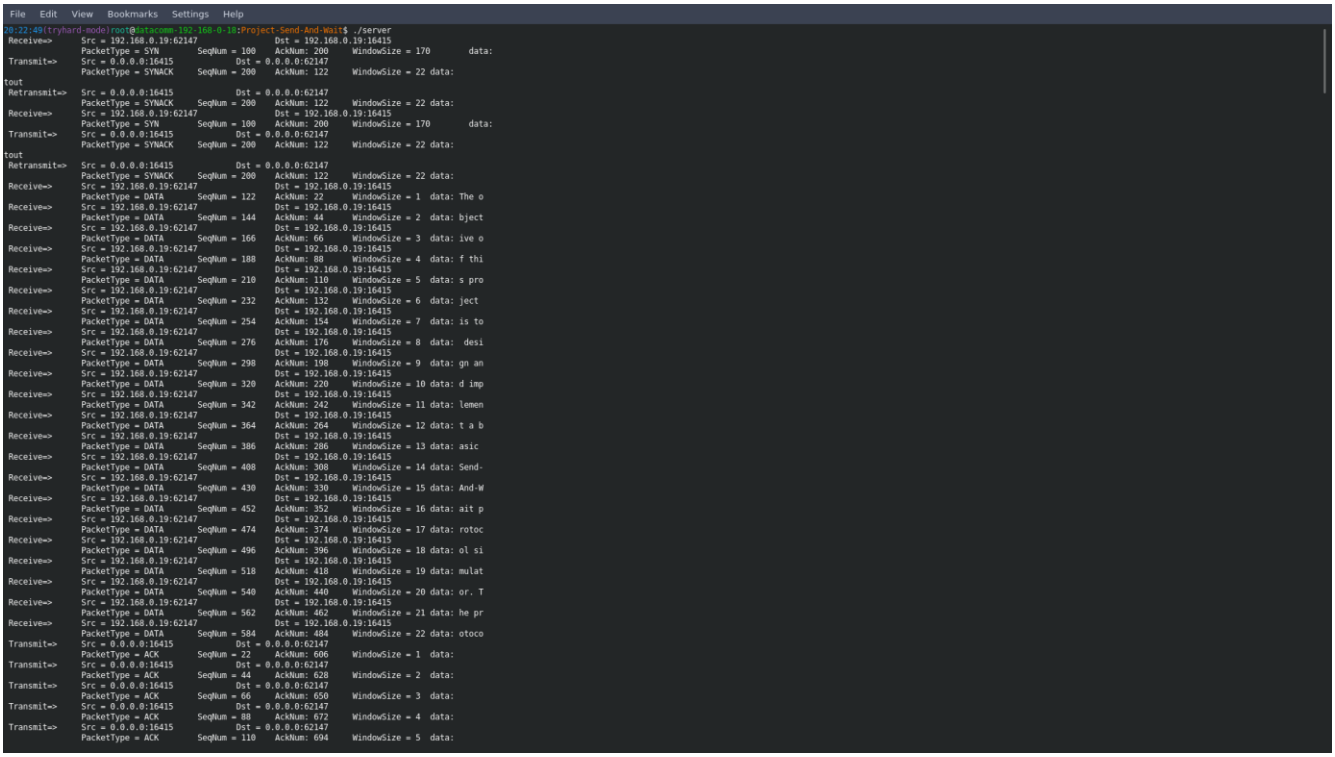
```
File   Edit   View   Bookmarks   Settings   Help
20:22:55(tryhard-mode)root@datacomm-192-168-0-19:Project-Send-And-Wait$ ./e -n 30 -d 0 -f config.txt
noise = 30% delay = 0 ms
listen for client
 Receive=>      Src = 192.168.0.20:28695               Dst = 192.168.0.20:22555
                PacketType = SYN        SeqNum = 100    AckNum: 200     WindowSize = 170        data:
 Transmit=>     Src = 0.0.0.0:22555             Dst = 0.0.0.0:16415
                PacketType = SYN        SeqNum = 100    AckNum: 200     WindowSize = 170        data:
listen for server
 Receive=>      Src = 192.168.0.18:16415               Dst = 192.168.0.18:22555
                PacketType = SYNACK     SeqNum = 200    AckNum: 122     WindowSize = 22 data:
packet lost from server
listen for server
```

Note: above we see that emulator is dropping not SYN packet as we might have thought but SYNACK meaning server successfully received the SYN in the first time.

Later, we see that server-side experienced time-out and retransmitted the SYNACK again. But, the reason why the SYNACK retransmit is before the SYN retransmit is because the emulator was designed so it prioritizes dropped packet side first. Emulator was intentionally designed so it can listen for one side at a time as stated in the project send-and-wait requirements.

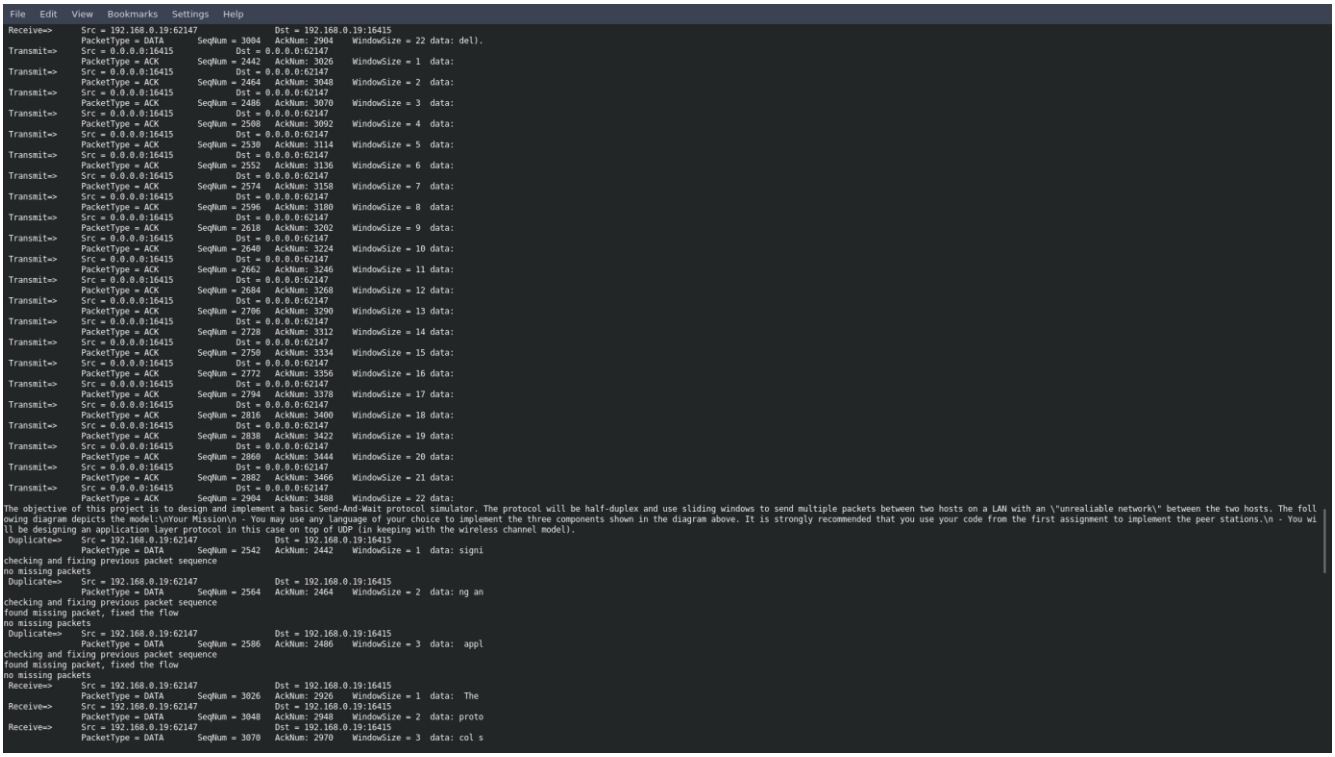
```
File   Edit   View   Bookmarks   Settings   Help
20:22:55(tryhard-mode)root@datacomm-192-168-0-19:Project-Send-And-Wait$ ./e -n 30 -d 0 -f config.txt
noise = 30% delay = 0 ms
listen for client
 Receive=>      Src = 192.168.0.20:28695               Dst = 192.168.0.20:22555
                PacketType = SYN        SeqNum = 100    AckNum: 200     WindowSize = 170        data:
 Transmit=>     Src = 0.0.0.0:22555             Dst = 0.0.0.0:16415
                PacketType = SYN        SeqNum = 100    AckNum: 200     WindowSize = 170        data:
listen for server
 Receive=>      Src = 192.168.0.18:16415               Dst = 192.168.0.18:22555
                PacketType = SYNACK     SeqNum = 200    AckNum: 122     WindowSize = 22 data:
packet lost from server
listen for server
 Receive=>      Src = 192.168.0.18:16415               Dst = 192.168.0.18:22555
                PacketType = SYNACK     SeqNum = 200    AckNum: 122     WindowSize = 22 data:
 Retransmit=>   Src = 0.0.0.0:22555             Dst = 0.0.0.0:28695
                PacketType = SYNACK     SeqNum = 200    AckNum: 122     WindowSize = 22 data:
listen for client
 Receive=>      Src = 192.168.0.20:28695               Dst = 192.168.0.20:22555
                PacketType = SYN        SeqNum = 100    AckNum: 200     WindowSize = 170        data:
 Retransmit=>   Src = 0.0.0.0:22555             Dst = 0.0.0.0:16415
                PacketType = SYN        SeqNum = 100    AckNum: 200     WindowSize = 170        data:
listen for server
 Receive=>      Src = 192.168.0.18:16415               Dst = 192.168.0.18:22555
                PacketType = SYNACK     SeqNum = 200    AckNum: 122     WindowSize = 22 data:
packet lost from server
listen for server
```

Server running under the same conditions (30% noise ratio and 0ms delay)



Note: server is received SYN twice and responded with SYNACK twice. It can be explained so that after retransmitting SYNACK, server waited but received SYN instead of DATA packet. After receiving SYN 2nd time, server gave a little more time for DATA packet to arrive before sending the SYNACK packet again. As it did not receive the DATA packet, it immediately assumed that client is reset and waiting for the SYNACK and fast-retransmitted SYNACK.

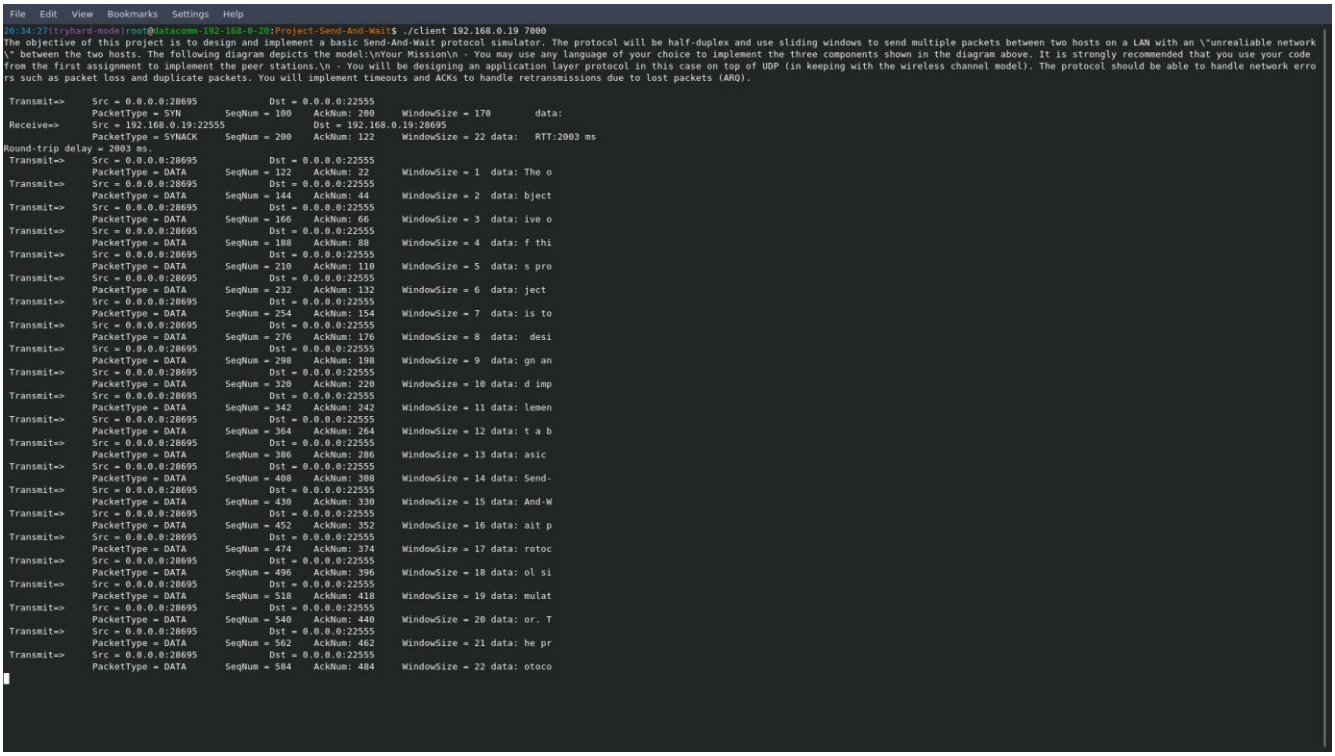
More situations on server-side:



As seen above, server is receiving duplicate DATA packets because client might have not received ACK packets for these duplicate DATA packets.

Server will send the ACK packets for these duplicate DATA packets after it is done sending the ACK packets for the recent ones. The difference is client is waiting only for 22 ACK packets, so server supplies additional Boolean value “re” to the last ACK packet with window size of 22. So that, client will keep listening for ACK packets even after receiving the 22 ACK packets.

3. *Client* executing under emulator conditions of 0% noise and 1200ms delay



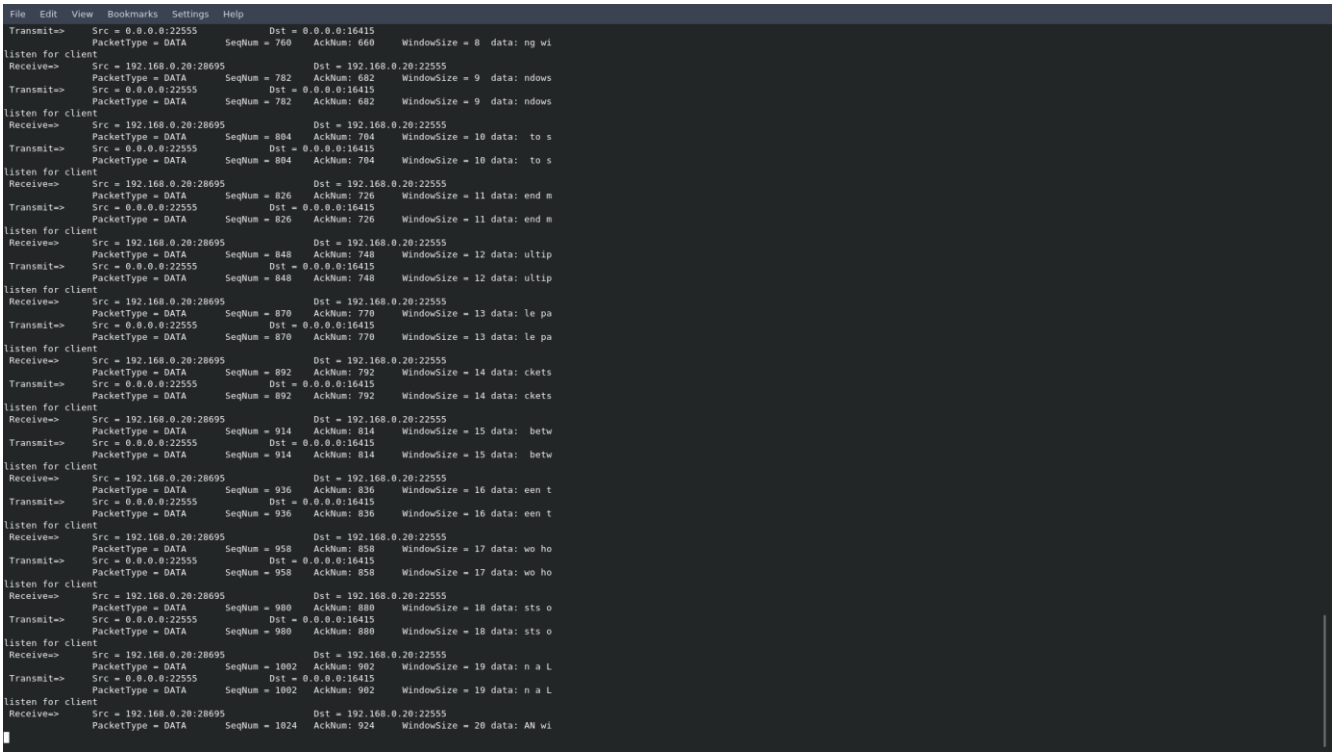
Note: as the time-out is 10 seconds for the client, it would have enough wait time to receive the SYNACK.

Round-trip time for handshake session is 2003ms which is ~ 2 x 1200ms; 2s ~ 2.4s

Notice after transmitting the 22 DATA packets, the client is not receiving the ACK packets immediately as it will arrive later because emulator will delay each of these DATA packet to 1.2s before forwarding to the server.

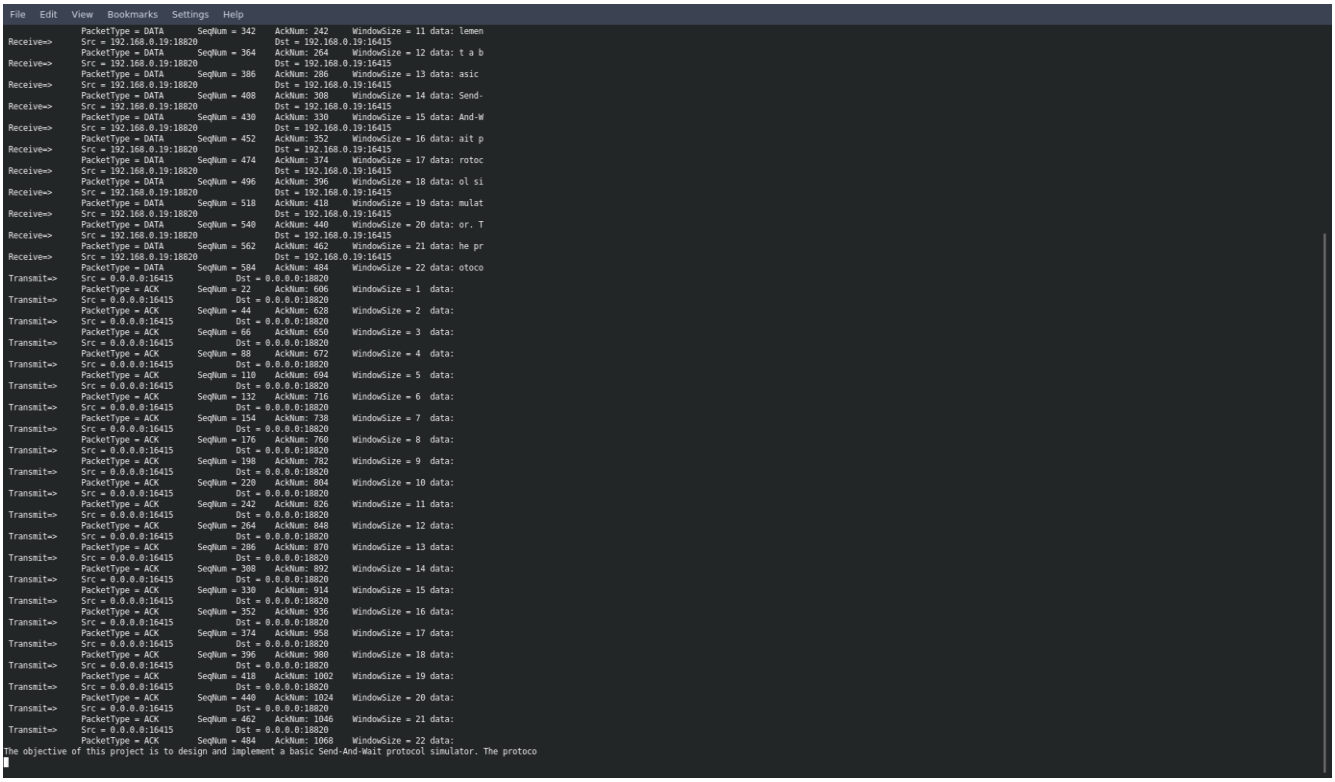
So, 22 packets * 1s ~ 22+s which means the client would have to retransmit the whole frames again.

Network Emulator executing the conditions of 0% noise and 1200ms delay



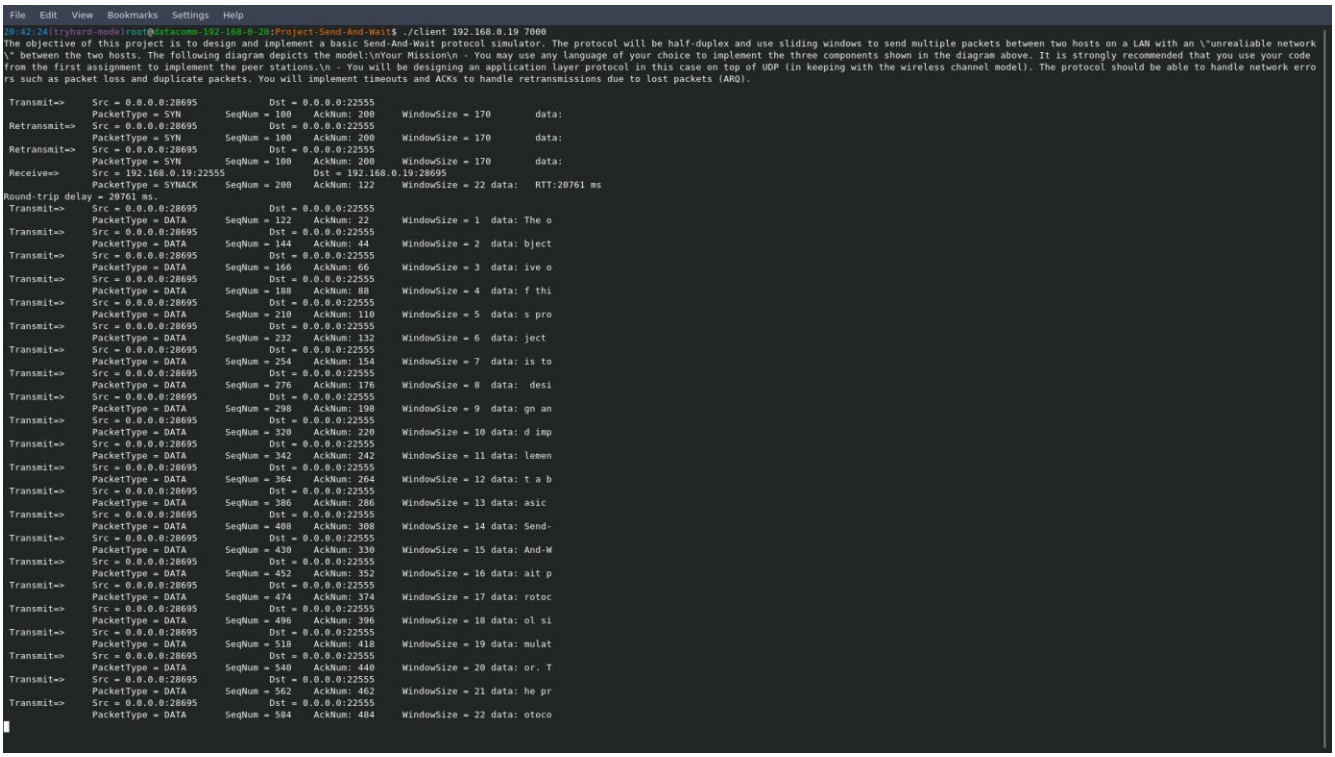
As the executions are underway, the emulator is holding the packets for 1200ms before forwarding to the server. No sudden drops are caught during the process.

Server execution is underway with 0% noise ratio and 1200ms delay per packet



The process is much alike the one with no 0ms delay as the time-out was updated for server making it longer wait time.

4. *Client* is executing under quite extreme conditions of 60% noise ratio and 400ms delay per packet



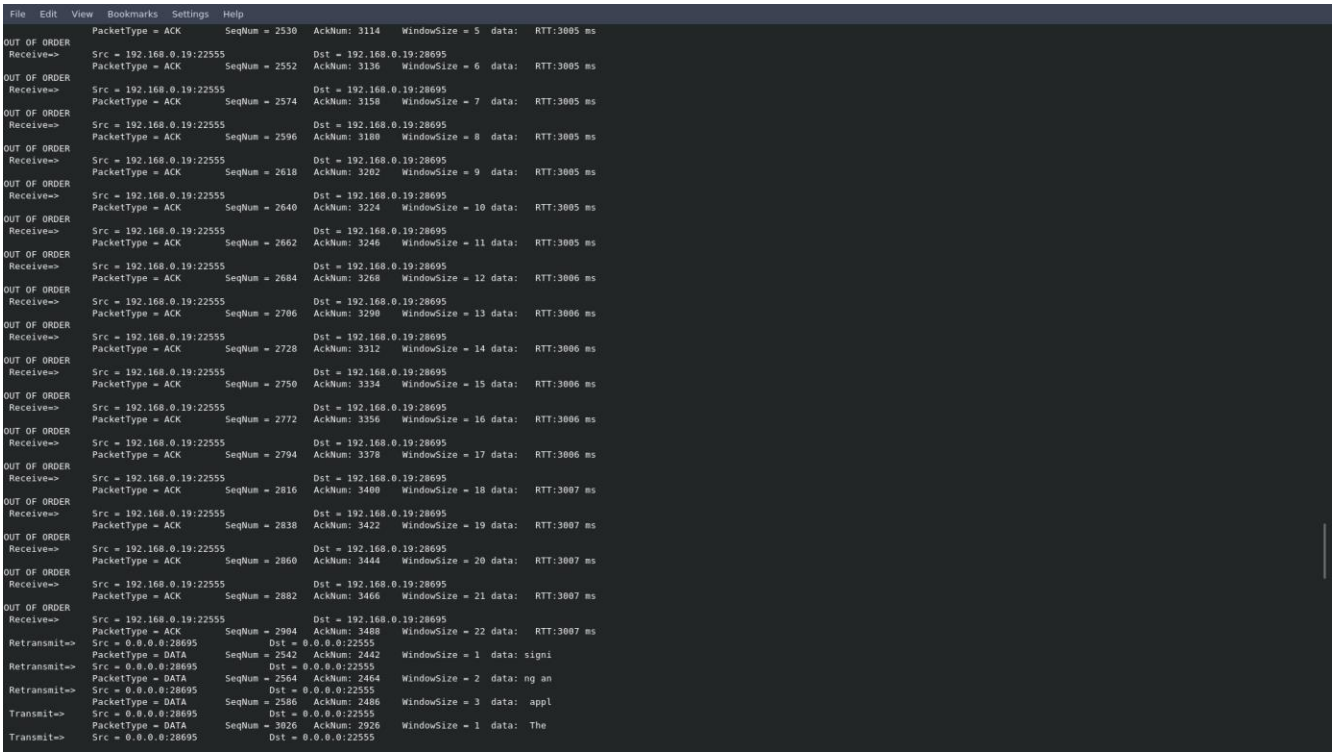
Note: client is already experiencing two time-outs before finally receiving the long waited SYNACK.

Because of the two time-outs, round-trip time is drastically increased to 20761ms or ~ 21s.

In addition, as seen above, the client is not receiving the ACK packets immediately after sending the DATA packets. It's a similar situation as happened with 1200ms delay before but with additional packet drops.

From here, we are expecting more ACK packets either are arriving not in sequence or not at all.

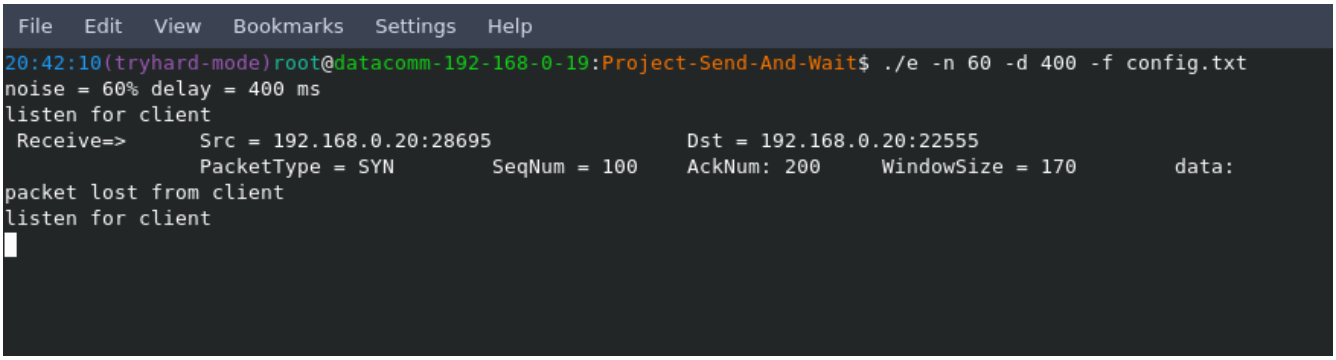
More on client-side (60% BER, 400ms)



Note: the high ACK packets are arriving as expected not in sequence and also increased round-trip times (3s) despite the fact that we are simulating with only 400ms ~ < 0.5s.

Also, note that the client is trying to fast-retransmit unacknowledged packets first than transmitting data. As stated in the project constraints, window will only slide forward only after all the current frames are acknowledged by the server. So, by fast-retransmitting, the client is trying to avoid additional wait time to wait before sending the next data frames.

Network Emulator screenshot of start of 60% BER and 400ms delay

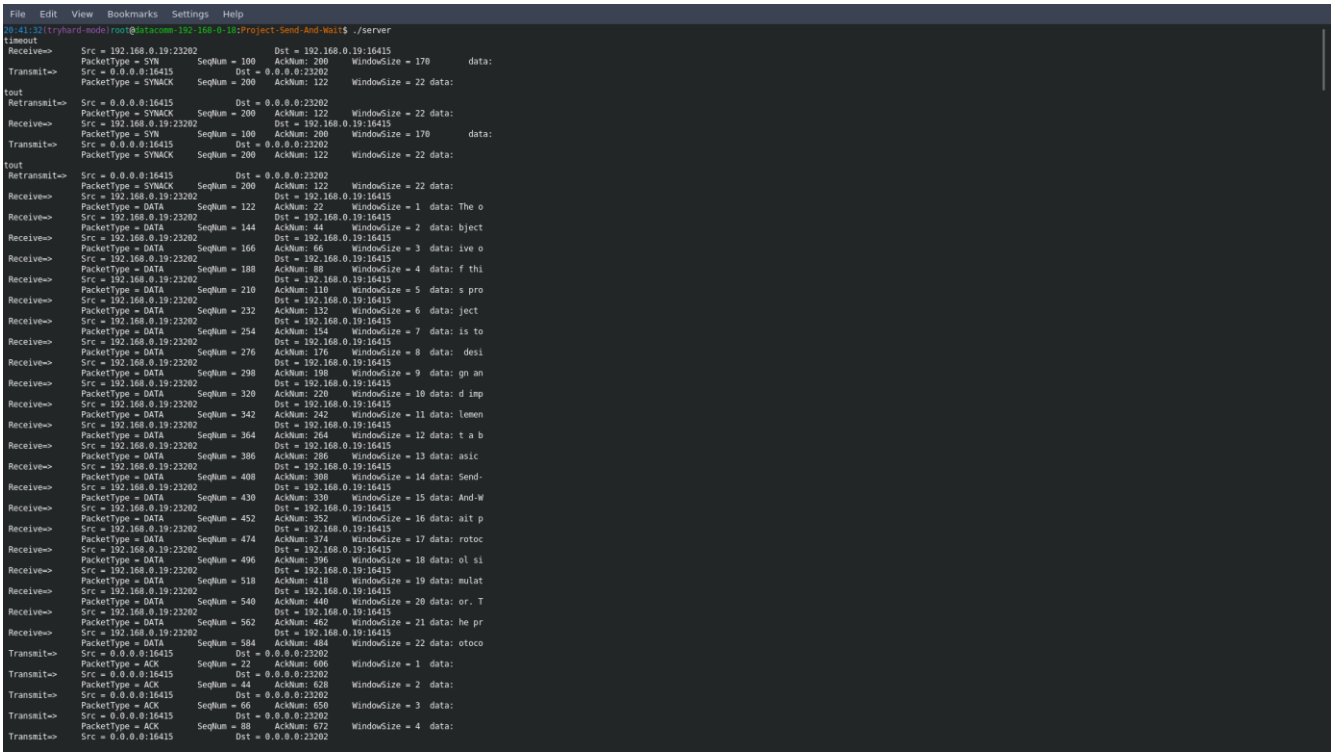


As we assumed from the client-side, network did drop the first SYN packet.

More captured random drops as shown below:

```
               PacketType = DATA       SeqNum = 3004   AckNum: 2904    WindowSize = 22 data: del).
 Transmit=>    Src = 0.0.0.0:22555             Dst = 0.0.0.0:16415
               PacketType = DATA       SeqNum = 3004   AckNum: 2904    WindowSize = 22 data: del).
listen for server
 Receive=>     Src = 192.168.0.18:16415                Dst = 192.168.0.18:22555
               PacketType = ACK        SeqNum = 2442   AckNum: 3026    WindowSize = 1  data:
packet lost from server
listen for server
 Receive=>     Src = 192.168.0.18:16415                Dst = 192.168.0.18:22555
               PacketType = ACK        SeqNum = 2464   AckNum: 3048    WindowSize = 2  data:
packet lost from server
listen for server
 Receive=>     Src = 192.168.0.18:16415                Dst = 192.168.0.18:22555
               PacketType = ACK        SeqNum = 2486   AckNum: 3070    WindowSize = 3  data:
packet lost from server
listen for server
 Receive=>     Src = 192.168.0.18:16415                Dst = 192.168.0.18:22555
               PacketType = ACK        SeqNum = 2508   AckNum: 3092    WindowSize = 4  data:
 Transmit=>    Src = 0.0.0.0:22555             Dst = 0.0.0.0:28695
               PacketType = ACK        SeqNum = 2508   AckNum: 3092    WindowSize = 4  data:
listen for server
 Receive=>     Src = 192.168.0.18:16415                Dst = 192.168.0.18:22555
               PacketType = ACK        SeqNum = 2530   AckNum: 3114    WindowSize = 5  data:
 Transmit=>    Src = 0.0.0.0:22555             Dst = 0.0.0.0:28695
               PacketType = ACK        SeqNum = 2530   AckNum: 3114    WindowSize = 5  data:
```

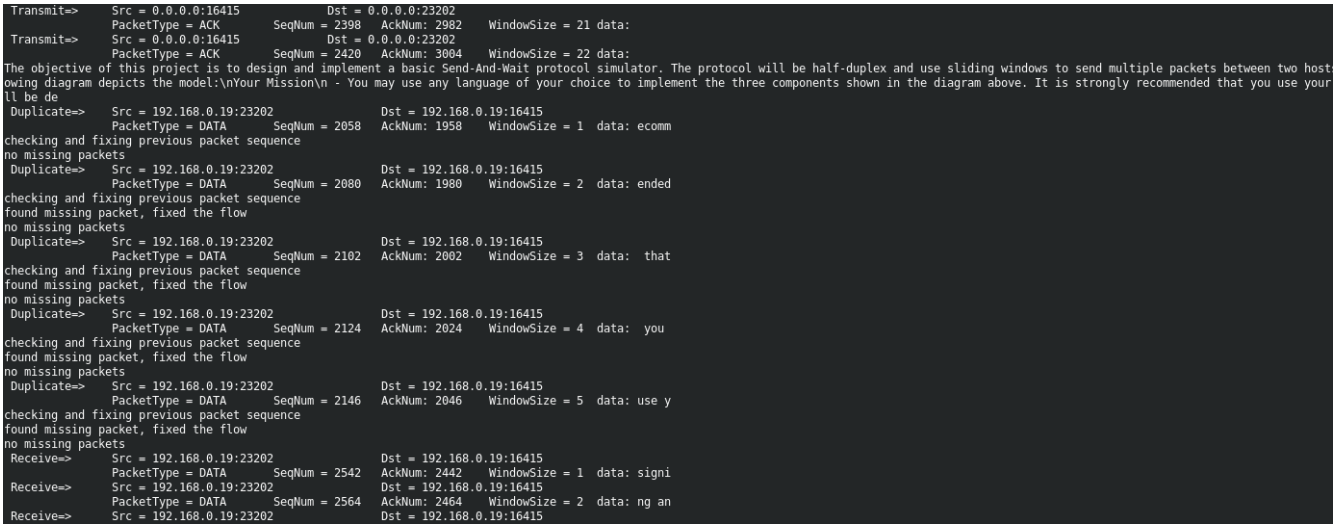
Server execution under 60% BER and 400ms delay



As seen above, the server is handling the initial handshake session issues. Both, the client and the server had to retransmit twice during the handshake session before finally settling down.

Fortunately, the server received the first 22 DATA packets without drops.

However, under the known circumstances, the server is experiencing more unordered duplicates and some missing DATA packets as shown below.



Future goals

The application is functional and can handle quite a few real-world Internet situations. However, the shortage of time, the limited knowledge of the C Programming and other circumstances are hold us to improve our application beyond its current simple state. As of now, we have listed future goals that might change the application in different good way.

1. Fully adaptable values. As mentioned, most of the values are hard-coded and given constant values such as filenames, timeouts, window sizes etc.
2. Full-duplex implementation. We were aware of the suggestion given by our instructor; but again, busyness is taken over us during the project period.
3. User friendly GUI implementation. As we have researched, to implement that we need to study additional libraries or even completely switch to C++ which would make a lot of things easier as implementation in OOP is much clearer than just procedural language as C, but as we know now even C can still hold its own.
4. Adding additional network emulator functions such as cutting down the connections during the transmission, or even changing the structure of the content which would require checksum implementations for both ends: transmitter and receiver.
5. Visual animations. Actually, that is still in progress as of now but finishing it by deadline seems impossible. Visual animation would display network emulator actions such as receiving packet and dropping it, likewise with the ACK packets.
6. More transmitter and receivers sharing a single or many routers.
7. Multithreading. More researching and studying need to be done before even looking at it.
8. BASH scripting. Although we have already implemented a script for compiling but not for the results. BASH script results could return information like the one Linux tool "ping" does.
9. Windows – Linux implementation.
10. Different data file types. E.g. images, videos, etc.