Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Dmytro Kovalenko

Communication application using the UDP protocol

Computer and Communication Networks

AIS ID: 122551

Cvičenie: Štvrtok 10:00

Cvičiaci: Ing. Matej Janeba

Rok: 2024/2025

Contents

Aim of the assignment

Design and implement a P2P application using a custom protocol built on top of User Datagram Protocol (UDP) in the transport layer of the TCP/IP network model. The application will allow communication between two participants in a local Ethernet network, including text exchange and transfer of arbitrary files between computers (nodes). The two nodes will act simultaneously as receiver and sender.

The assignment consists of two parts: theoretical and practical. In the theoretical part, you will design your own protocol on top of UDP. In the practical part, you will then implement and demonstrate its functionality in communicating between two nodes, highlighting the modifications made during the implementation compared to the original design.

Beginning

Before I start with my protocol, I need to understand UDP, because my head will be over it and P2P.

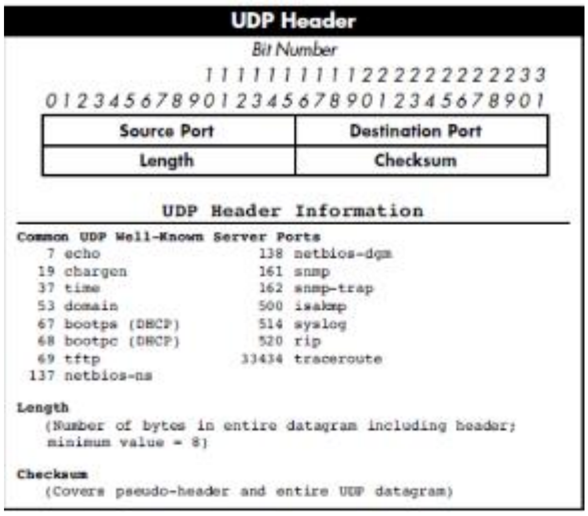
Peer to peer

Some applications are designed in a more distributed fashion where there is no single server. Instead, each application acts both as a client and as a server, sometimes at both at once, and is capable of forwarding requests. Some very popular application. These applications are called peer-to-peer or p2p applications. A concurrent p2p application may receive an incoming request, determine if it is able to respond to the request, and if not forward the request on to some other peer. Thus, the set of p2p applications together form a network among applications, also called an overlay network [1].

UDP

The User Datagram Protocol gives application programs direct access to a datagram delivery service, like the delivery service that IP provides.

UDP is an unreliable, connectionless datagram protocol. "Unreliable" merely means that there are no techniques in the protocol for verifying that the data reached the other end of the network correctly. UDP uses 16-bit Source Port and Destination Port numbers in word 1 of the message header to deliver data to the correct applications process.



UDP Header Information

Common UDP Well-Known Server Ports

7	echo	138	netbios-dgm
19	chargen	161	snmp
37	time	162	snmp-trap
53	domain	500	isakmp
67	bootps (DHCP)	514	syslog
68	bootpc (DHCP)	520	rip
69	tftp	33434	traceroute
137	netbios-ns		

Length
(Number of bytes in entire datagram including header; minimum value = 8)

Checksum
(Covers pseudo-header and entire UDP datagram)

UDP Header[2]

If the amount of data to be transmitted is small, the overhead of establishing connections and ensuring reliable delivery can be greater than the cost of retransmitting the entire data set. In this case, UDP is the most efficient choice of transport layer protocol. If a response is not received within a certain period of time, the application simply sends a new request. [3]

The structure of my protocol header

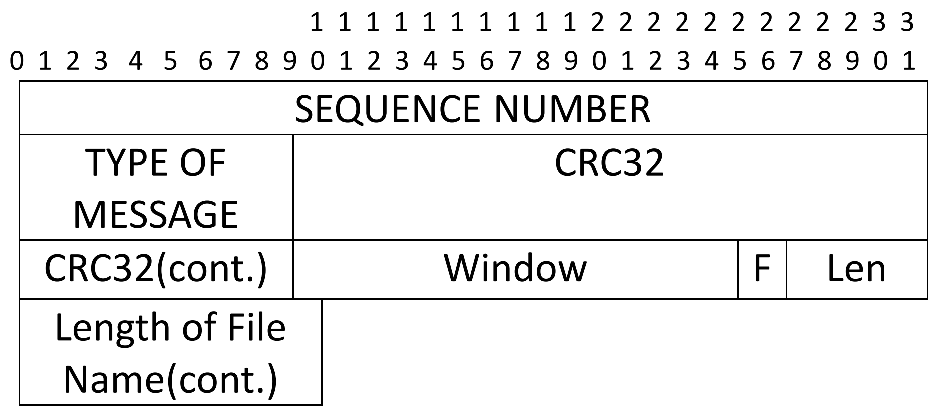
Now I'd like to present to you my protocol header.

```
                    1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

SEQUENCE NUMBER			
ACKNOWLEDGMENT NUMBER			
TYPE OF MESSAGE		CRC32	
CRC32(cont.)		Window	
WINDOW SCALING	FLAG	LENGTH OF PATH	Len
LENGTH OF FILE NAME(cont.)			

In total, as you can see from my interpretation of the protocol header, it will take 155 bits. It takes up a lot of bits, but it contains everything that is important.

New version of my protocol



In the new version of my protocol, I added fields such as acknowledgement number, path length. The reason why I removed the acknowledgement number field is that I pass the packet id in the seqNumber, not the byte stream, plus to send a message that is responsible for acknowledging acceptance, I use a special type of message in which the seqnumber will be equal to the acknowledged packet. I removed the length of path because I misunderstood the task, and now it is not needed at all, because the user enters where he will save the file at the end of acceptance. The total size of my header is 13 bytes + 2 bits. F stands for Flag, it contains 2 bits The maximum fragment size is now 1458 bytes.

Sequence number

The sequence number is a counter used to keep track of each byte sent outward by the host. If a TCP packet contains 1400 bytes of data, the sequence number will be incremented by 1400 after the packet is transmitted. As in TCP, the sequence number will take 4 bytes.

Correction:

My seq number now works in such a way that it does not transmit the size of the dates, but it is like the id of this fragment. First, it is generated as a random number for ST, and for the fragment transfer itself, it starts from 0, and depending on the number of fragments, it increases.

Acknowledgment number

Next in line is the acknowledgement number, which is required by my protocol. An acknowledgement number is sent by the TCP server, indicating that it has received the accumulated data and is ready for the next segment. This way, I can check whether the other user is ready to receive the next data and whether the data has been lost. This field will occupy 4 bytes in the protocol header.

Correction:

In the final version of my protocol, I do not have this field, it can be said to be replaced by the type of message. Example: to confirm the acceptance of a file fragment, I use type 9 (which is in my SM program), to accept a fragment of a regular message, I use type 4 (which is in my CM program).

Type of message

In my protocol header, this field will play an important role in what message the user sent and what the response was from the one who received it. At the moment I have implemented only a few message types such as: ST – Start, AN – answer to start, CO – confirm start, FI-finish, SE – send message(string), CM – confirm message (string), SF- send file, CF-confirm file. Type of Message field will occupy 2 bytes.

Correction:

In the final version, I changed the length of this field to 1 byte. I also added new message types and changed the old ones, namely: 0 - ST(start), 1 - AN(answer to start), 2 - FI (finish), 3 - SE (send message), 4 - CM (confirm message), 5 - RE (resend message), 6 - KI (keep alive), 7 - KR (reply keep alive), 8 - SF (send file), 9 - SM (confirm file).

Checksum (CRC32)

I use CRC32 to compare files because in a wide practical use involving billions of files, it is extremely rare for CRC32 checksums and file sizes to match without deliberate manipulation. When I compare the CRC32 checksum and the data size, I would like to have a much lower probability of collisions. I would like my application to be able to send several frames at a time and with a larger size. So for this purpose, I have allocated 4 bytes.

Window

Window is the amount of data (in bytes) that can be buffered during a connection. The sending host can only send this amount of data before it has to wait for the receiving host to acknowledge and update the window. In my opinion, the data transfer process with this field will become more efficient. The window field will occupy 2 bytes.

Correction:

In the final version, I changed the behaviour of the window itself, it no longer contains the length of the data without confirmation, but contains how many fragments I can send without confirmation.

Window Scaling

Scaling a window effectively increases the capacity of my Window field from 16 to about 24 bits. Instead of resizing the field, the header remains a 16-bit value, and a parameter is defined that applies a scaling factor to the 16-bit value. This factor effectively shifts the value of the window field to the left by the amount of the zoom factor. This essentially multiplies the window value by 2^s, where s is the scaling factor. [1] The number of shifts per byte ranges from 0 to 8. The maximum scaling value of 8 provides a maximum window of 16776960 bytes (65535*2^8). Window Scaling field will occupy 1 byte.

Correction:

After testing my program on many messages and files, I came to the conclusion that I can remove the window scaling field, because I cannot reach a very large window size.

Flag

In total, I currently have 4 flags in my protocol. The first one is N (next fragment), W (for window scale to confirm that it will be used), T (file or message), F (final or no). The flag field will take 4 bits.

Correction:

In the final version of the program, I changed the number of flags to 2. I removed the N flag (the following snippet) because it was unnecessary and the W flag because I don't use window scaling anymore.

Length of path

This field will be responsible for the size in bytes of the route where the file should go. With the first request, I will send the route and the fragments themselves, and using this field, I will be able to find out how many bytes this route takes. Length of Path field will take 2 bytes.

Correction:

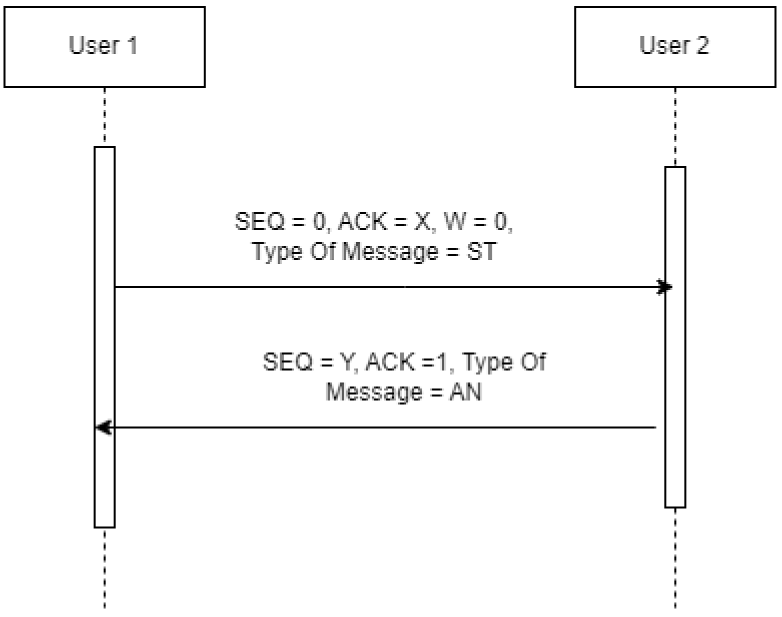
In the final version of the program, I removed this field altogether, because I misunderstood the task at the start. I now specify the final route where to save the file, already at the end of full acceptance on the recipient's side.

Length of name

This field, like the length of path field, will be responsible for the length, but now not of the path, but of the file name itself. It will also be sent with the first request with the route. First the route, then the file name, and only after that the data itself. The Length of Name field will take 2 bytes.

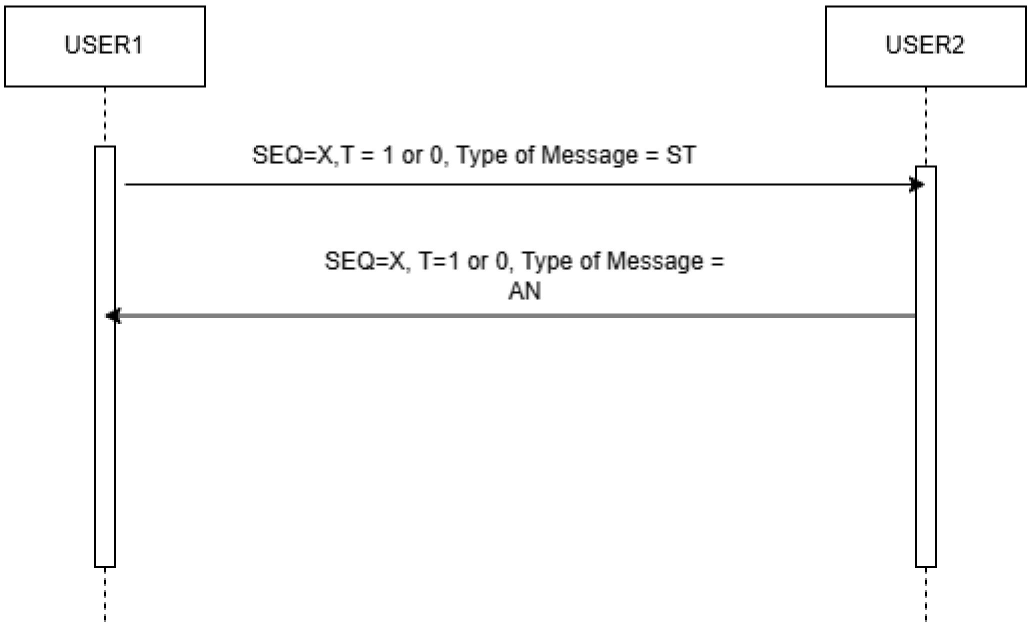
Handshake

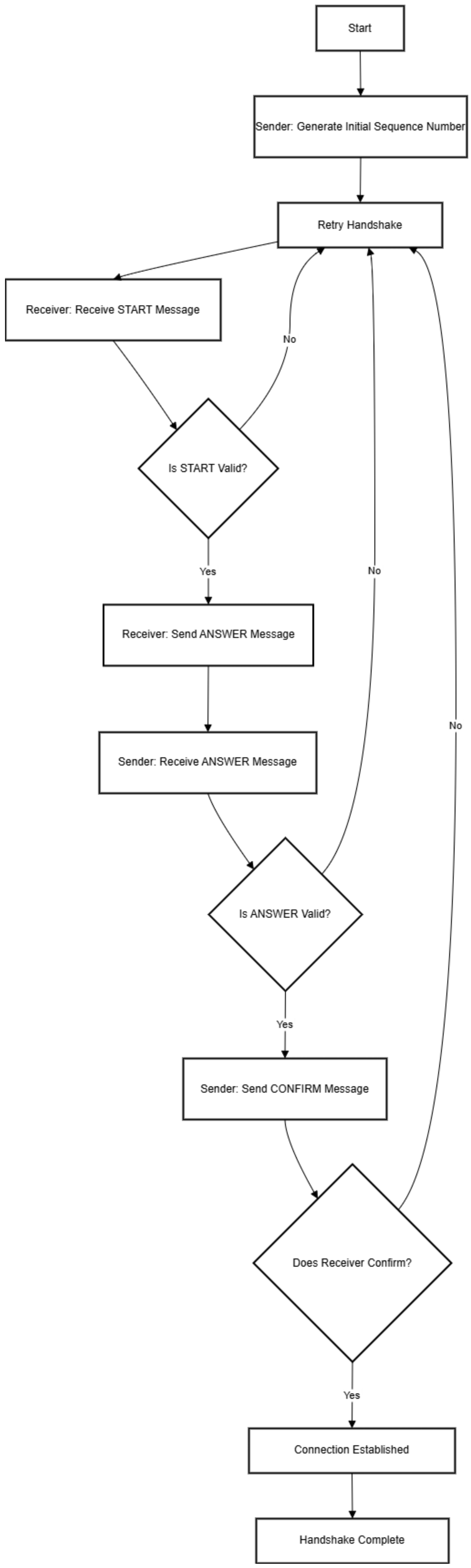
My Handshake works like this: first, the user who wants to send the message sends the first signal with SEQ = 0, ACK = X, W = 0 (can be sent with a signal of 1 if the user wants to send more bytes), and the T flag can be sent, which is responsible for whether it is a file (1) or just a message (0), and the type of the message itself (in this case it is a start, so the message type is ST). After the other user receives this message, he sends a response indicating whether he acknowledges the W flag, if set, and sends the message type AN. In general, I have a 2-stage handshake, so there may be problems with the fact that the other user can simply disconnect immediately, but I solved this problem with keep-alive, which checks if the user is inactive for 15 seconds, then it will send a keep-alive signal.


User 1
User 2
SEQ = 0, ACK = X, W = 0, Type Of Message = ST
SEQ = Y, ACK =1, Type Of Message = AN

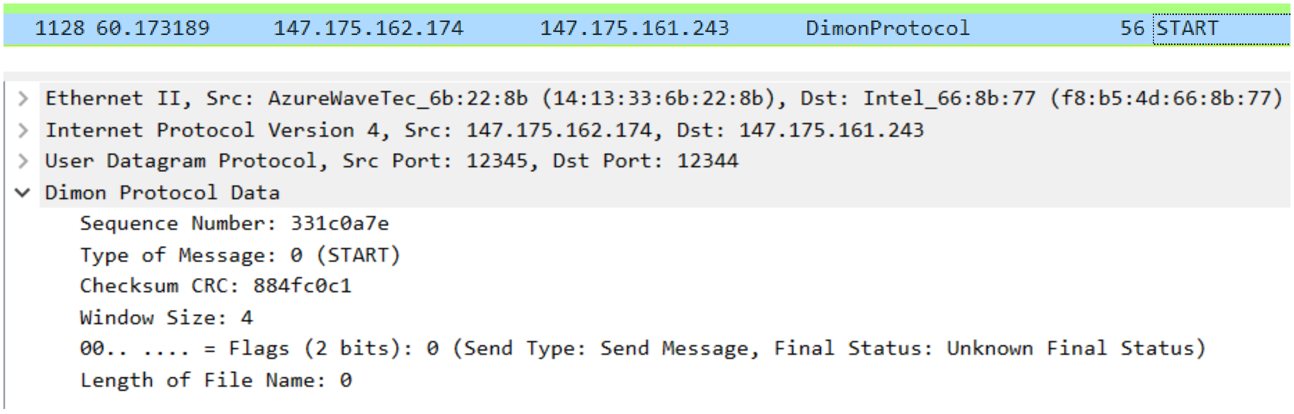
Correction:

In the final version, I changed the handshake a little bit, but I only changed the variables that I pass, but the attachment itself remained unchanged, I have it on the 2nd hand, because using keep alive, I always check the connection between the other person, and if I don't get a response within 3 seconds at the beginning, I send ST again, and so on 3 times.

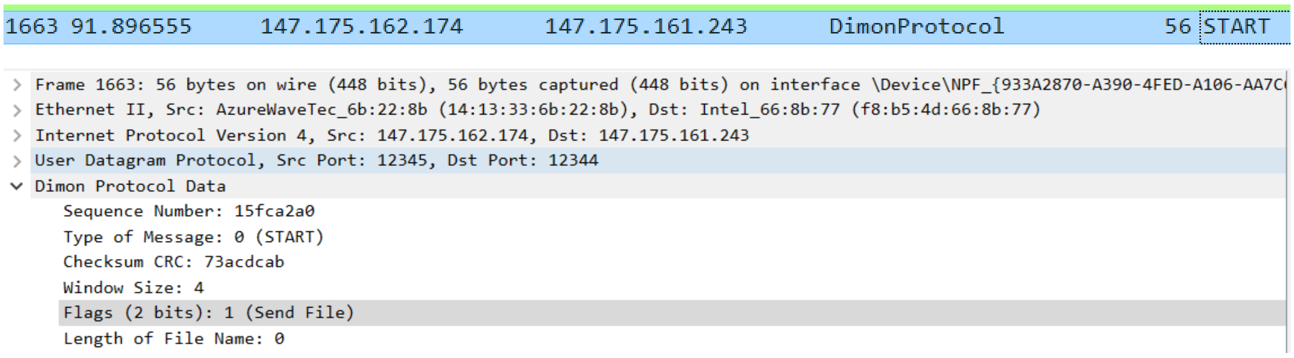

USER1
USER2
SEQ=X,T = 1 or 0, Type of Message = ST
SEQ=X, T=1 or 0, Type of Message = AN


Start
Sender: Generate Initial Sequence Number
Retry Handshake
Receiver: Receive START Message
No
Is START Valid?
Yes
No
Receiver: Send ANSWER Message
No
Sender: Receive ANSWER Message
Is ANSWER Valid?
Yes
Sender: Send CONFIRM Message
Does Receiver Confirm?
Yes
Connection Established
Handshake Complete

Wireshark

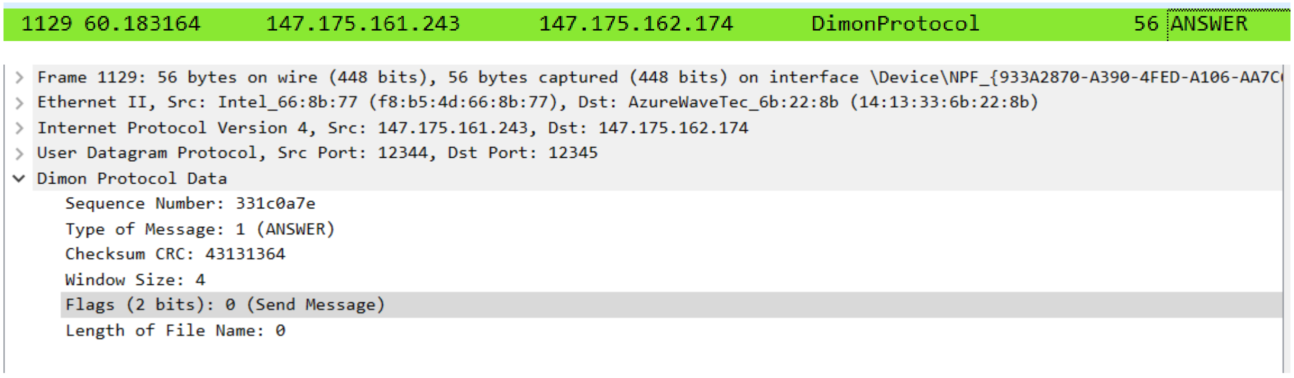


```
1128 60.173189      147.175.162.174      147.175.161.243      DimonProtocol       56 START

> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 331c0a7e
    Type of Message: 0 (START)
    Checksum CRC: 884fc0c1
    Window Size: 4
    00.. .... = Flags (2 bits): 0 (Send Type: Send Message, Final Status: Unknown Final Status)
    Length of File Name: 0
```

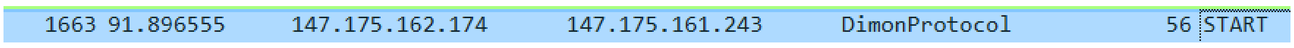


```
1663 91.896555      147.175.162.174      147.175.161.243      DimonProtocol       56 START

> Frame 1663: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 15fca2a0
    Type of Message: 0 (START)
    Checksum CRC: 73acdcab
    Window Size: 4
    Flags (2 bits): 1 (Send File)
    Length of File Name: 0
```

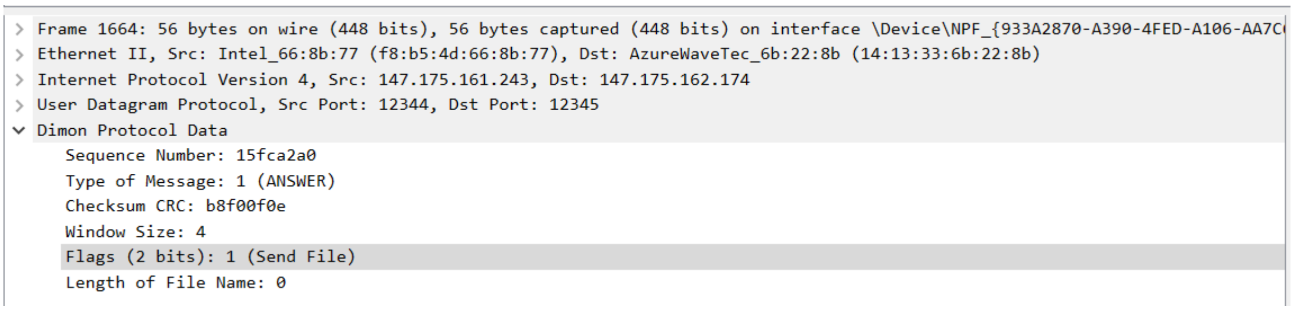
In these screenshots, we can see a message like START. After the user has established a connection and is typing the message he wants to send, the handshake starts. First, a START message is sent with a random sequence number and a flag of 0 or 1, 0 indicates that a normal message will be sent, and 1 indicates that a file message will be sent.



```
1129 60.183164      147.175.161.243      147.175.162.174      DimonProtocol       56 ANSWER

> Frame 1129: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: Intel_66:8b:77 (f8:b5:4d:66:8b:77), Dst: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b)
> Internet Protocol Version 4, Src: 147.175.161.243, Dst: 147.175.162.174
> User Datagram Protocol, Src Port: 12344, Dst Port: 12345
v Dimon Protocol Data
    Sequence Number: 331c0a7e
    Type of Message: 1 (ANSWER)
    Checksum CRC: 43131364
    Window Size: 4
    Flags (2 bits): 0 (Send Message)
    Length of File Name: 0
```



```
1663 91.896555      147.175.162.174      147.175.161.243      DimonProtocol       56 START
```



```
> Frame 1664: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: Intel_66:8b:77 (f8:b5:4d:66:8b:77), Dst: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b)
> Internet Protocol Version 4, Src: 147.175.161.243, Dst: 147.175.162.174
> User Datagram Protocol, Src Port: 12344, Dst Port: 12345
v Dimon Protocol Data
    Sequence Number: 15fca2a0
    Type of Message: 1 (ANSWER)
    Checksum CRC: b8f00f0e
    Window Size: 4
    Flags (2 bits): 1 (Send File)
    Length of File Name: 0
```

In these screenshots, we can see the response to the START message. In it, the sequence number is the same as the START sequence number. There are also flags that indicate what this user will receive.

Keep-Alive

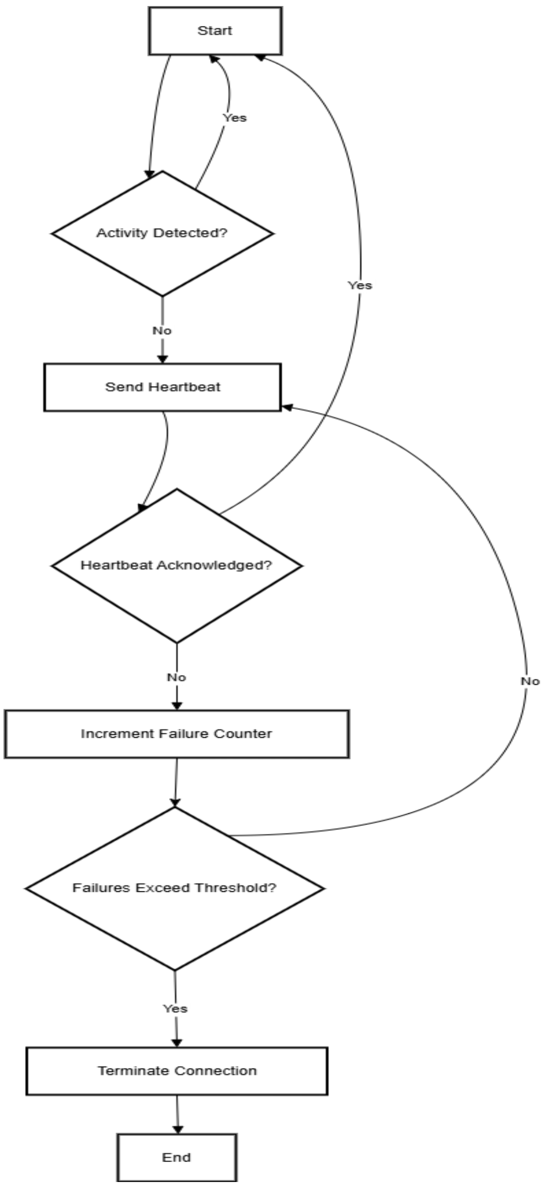
Keep-alive is a method for protocols to probe its peer without affecting the content of the data stream. It is driven by a keep-alive timer. When the timer fires, a keep-alive probe (keep-alive for short) is sent, and the peer receiving the probe responds with an ACK. [4]

My Implementation

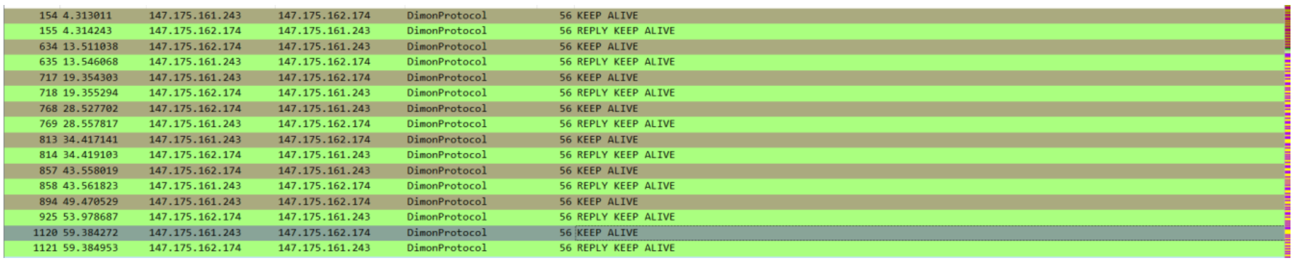
In my keep-alive program, it will be implemented so that at the start the user is connected to the port. The first keep-alive signal is sent, it will be sent every 5 seconds until the second user connects. After the other user has connected, he receives this signal and sends a response, after the first user has received the response, the keep-alive signal is stopped and the user is allowed to send a message. Also, if the user has not received a message within 15 seconds, it will start sending a keep-alive signal again to check if the user is on the other side. If no response is received within 5 requests, the connection is closed.

Correction:

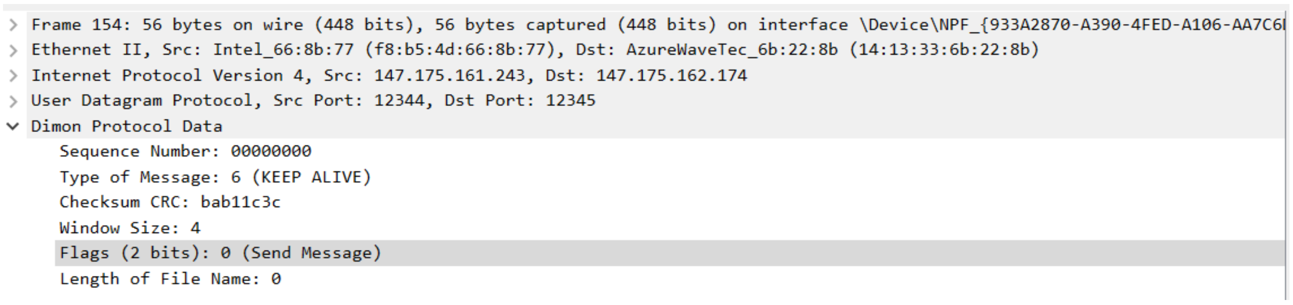
In the final version, keep alive now checks the connection for the entire duration of the application. Every 5 seconds it sends a request and waits for a response, if it doesn't respond 3 times, then the app just stop. Also, every time keep alive checks if there are any unanswered packets to which a response was not received, so that it can resend them.



Wireshark

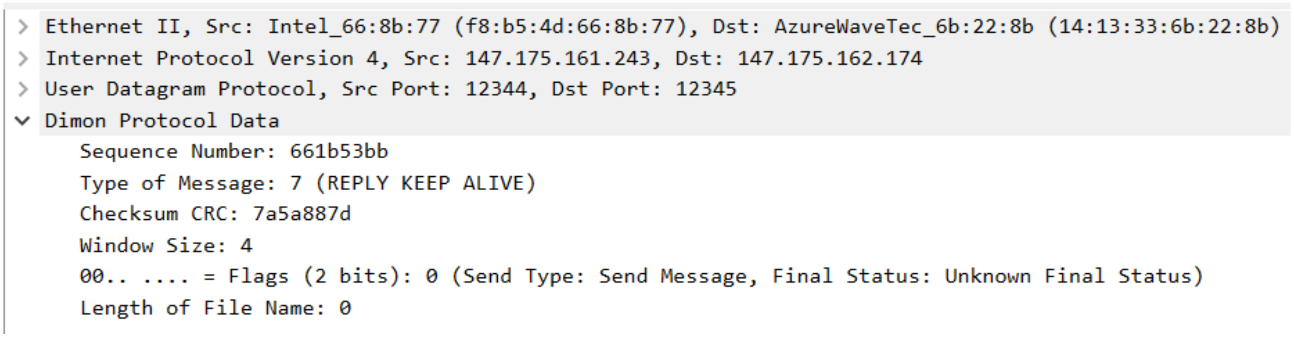


```
 154 4.313011   147.175.161.243   147.175.162.174   DimonProtocol    56 KEEP ALIVE
 155 4.314243   147.175.162.174   147.175.161.243   DimonProtocol    56 REPLY KEEP ALIVE
 634 13.511038  147.175.162.174   147.175.161.243   DimonProtocol    56 KEEP ALIVE
 635 13.546068  147.175.161.243   147.175.162.174   DimonProtocol    56 REPLY KEEP ALIVE
 717 19.354303  147.175.161.243   147.175.162.174   DimonProtocol    56 KEEP ALIVE
 718 19.355294  147.175.162.174   147.175.161.243   DimonProtocol    56 REPLY KEEP ALIVE
 768 28.527702  147.175.162.174   147.175.161.243   DimonProtocol    56 KEEP ALIVE
 769 28.557817  147.175.161.243   147.175.162.174   DimonProtocol    56 REPLY KEEP ALIVE
 813 34.417141  147.175.161.243   147.175.162.174   DimonProtocol    56 KEEP ALIVE
 814 34.419103  147.175.162.174   147.175.161.243   DimonProtocol    56 REPLY KEEP ALIVE
 857 43.558019  147.175.162.174   147.175.161.243   DimonProtocol    56 KEEP ALIVE
 858 43.561823  147.175.161.243   147.175.162.174   DimonProtocol    56 REPLY KEEP ALIVE
 894 49.470529  147.175.161.243   147.175.162.174   DimonProtocol    56 KEEP ALIVE
 925 53.978687  147.175.162.174   147.175.161.243   DimonProtocol    56 REPLY KEEP ALIVE
1120 59.384272  147.175.161.243   147.175.162.174   DimonProtocol    56 KEEP ALIVE
1121 59.384953  147.175.162.174   147.175.161.243   DimonProtocol    56 REPLY KEEP ALIVE
```



```
> Frame 154: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C6
> Ethernet II, Src: Intel_66:8b:77 (f8:b5:4d:66:8b:77), Dst: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b)
> Internet Protocol Version 4, Src: 147.175.161.243, Dst: 147.175.162.174
> User Datagram Protocol, Src Port: 12344, Dst Port: 12345
v Dimon Protocol Data
    Sequence Number: 00000000
    Type of Message: 6 (KEEP ALIVE)
    Checksum CRC: bab11c3c
    Window Size: 4
    Flags (2 bits): 0 (Send Message)
    Length of File Name: 0
```

On this screen, we can see a message like KEEP ALIVE. It is sent every 5 seconds to check the connection with the user.



```
> Ethernet II, Src: Intel_66:8b:77 (f8:b5:4d:66:8b:77), Dst: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b)
> Internet Protocol Version 4, Src: 147.175.161.243, Dst: 147.175.162.174
> User Datagram Protocol, Src Port: 12344, Dst Port: 12345
v Dimon Protocol Data
    Sequence Number: 661b53bb
    Type of Message: 7 (REPLY KEEP ALIVE)
    Checksum CRC: 7a5a887d
    Window Size: 4
    00.. .... = Flags (2 bits): 0 (Send Type: Send Message, Final Status: Unknown Final Status)
    Length of File Name: 0
```

In this screenshot, you can see a message like REPLY KEEP ALIVE. It is sent immediately after receiving KEEP ALIVE to let the other user know that the signal is still alive.

Description of the method to ensure reliable data transmission (ARQ)

In the Selective Repeat protocol, the retransmitted frame is received out of sequence. In selective repeat, the sender sends multiple frames defined by the window size, even without having to wait for a separate acknowledgement from the receiver. In the Selective Repeat protocol, the resent frame is received out of sequence. In the Selective Repeat ARQ protocol, only lost or erroneous frames are retransmitted, while correct frames are received and buffered. The receiver, keeping track of the sequence numbers, buffers the frames in memory and sends a NACK only for lost or damaged frames. The sender sends/retransmits the packet for which the NACK is received. I will implement NACK in such a way that it takes the sequence number and - 1, which will indicate to me that the file was lost.[7] My program will use the Selective Repeat ARQ method. Therefore, I have fields like Window in the protocol header, and to improve Selective Repeat, I will have another field called Window Scaling to increase the number of packets transmitted.

Correction:

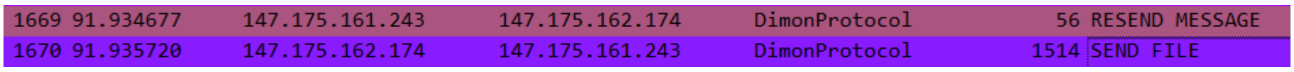
In the final version, my ARQ has a dynamically sized window. At the start, it is set that 4 fragments can be transmitted without acknowledgement, after the response about accepting the fragment comes, I use a formula to calculate whether the window size can be increased.[9]

$$\text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT}$$

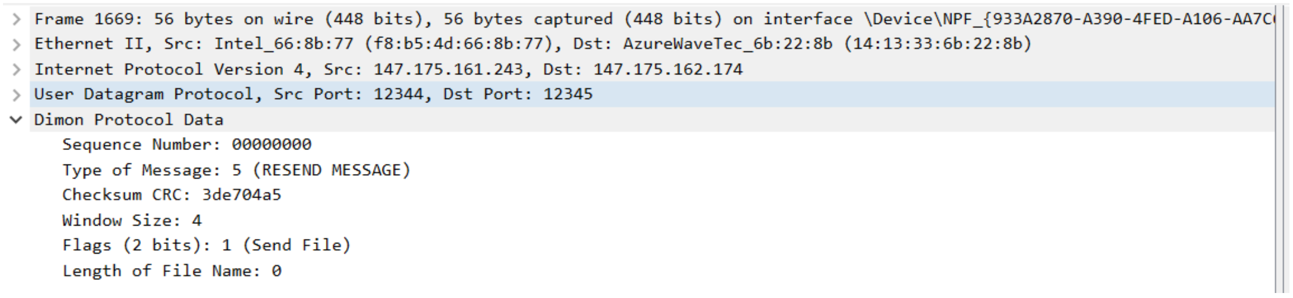
My alpha is 0.2 gives 20% weight to the latest RTT and 80% to the historical average, making RTT adjustments gradual. When I calculate the estimatedRTT, I look to see if it is less than 100(the most appropriate constant), if so, I increase the window size by 1, if it is less, I decrease it.

Below you can see how my selective arq works

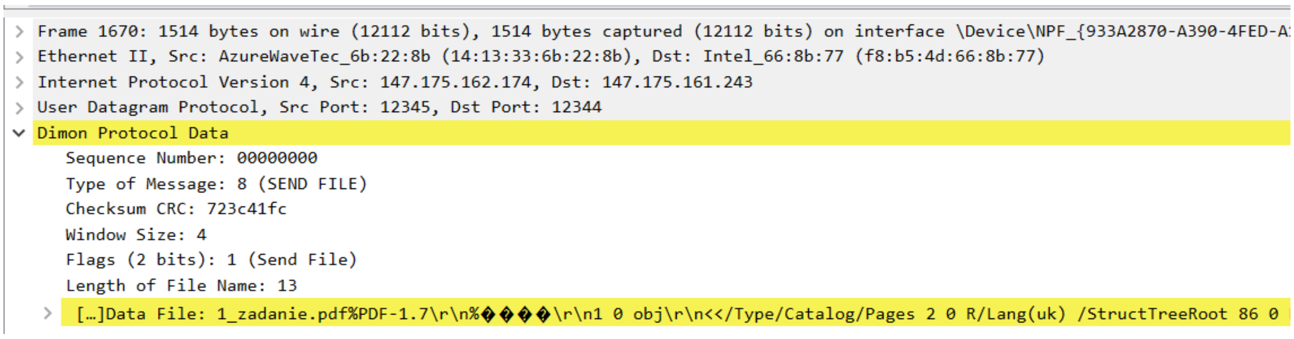
Wireshark



```
1669 91.934677    147.175.161.243    147.175.162.174    DimonProtocol      56 RESEND MESSAGE
1670 91.935720    147.175.162.174    147.175.161.243    DimonProtocol    1514 SEND FILE
```

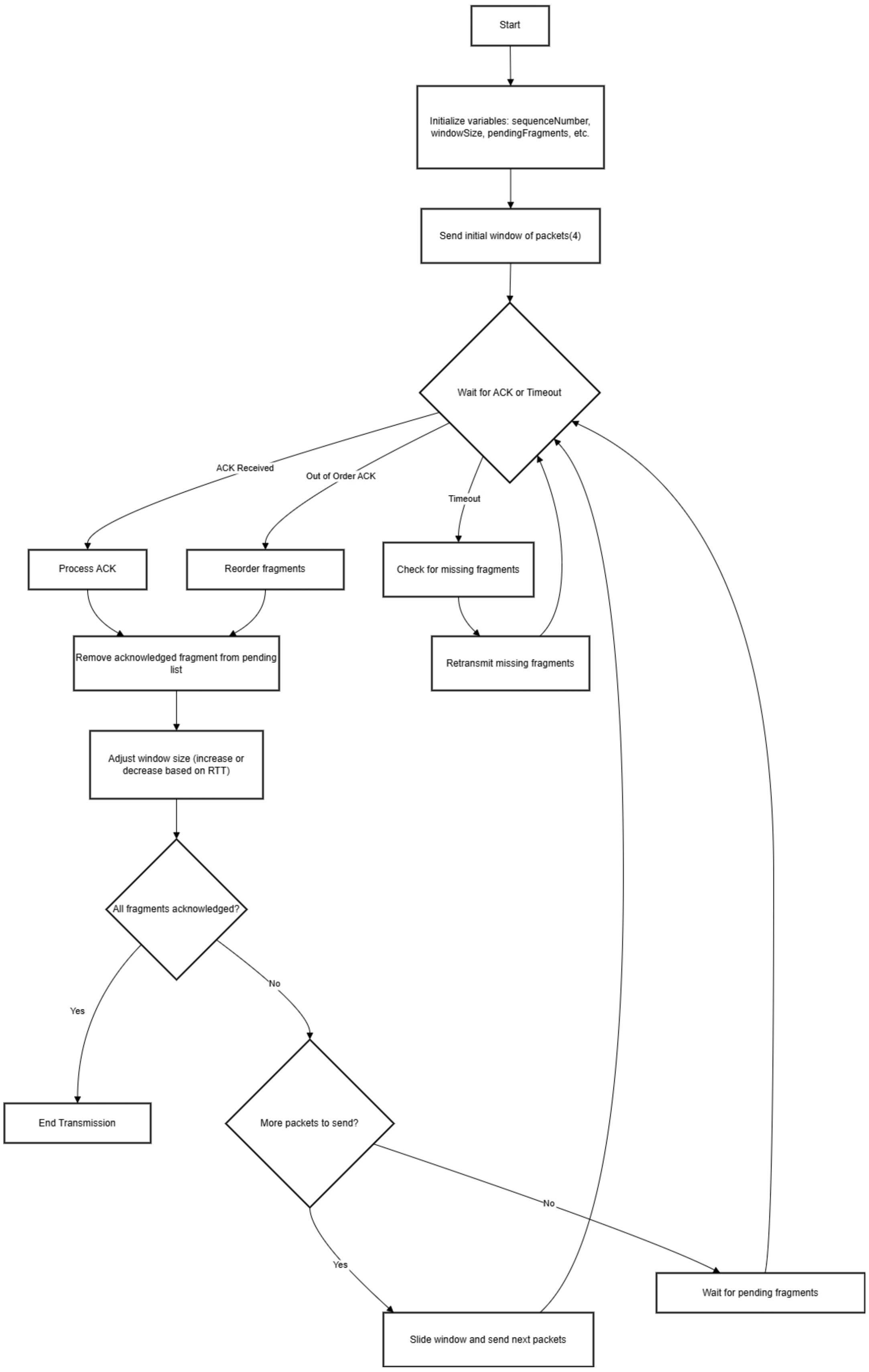


```
> Frame 1669: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: Intel_66:8b:77 (f8:b5:4d:66:8b:77), Dst: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b)
> Internet Protocol Version 4, Src: 147.175.161.243, Dst: 147.175.162.174
> User Datagram Protocol, Src Port: 12344, Dst Port: 12345
v Dimon Protocol Data
    Sequence Number: 00000000
    Type of Message: 5 (RESEND MESSAGE)
    Checksum CRC: 3de704a5
    Window Size: 4
    Flags (2 bits): 1 (Send File)
    Length of File Name: 0
```



```
> Frame 1670: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface \Device\NPF_{933A2870-A390-4FED-A
> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 00000000
    Type of Message: 8 (SEND FILE)
    Checksum CRC: 723c41fc
    Window Size: 4
    Flags (2 bits): 1 (Send File)
    Length of File Name: 13
  > […]Data File: 1_zadanie.pdf%PDF-1.7\r\n%����\r\n1 0 obj\r\n<</Type/Catalog/Pages 2 0 R/Lang(uk) /StructTreeRoot 86 0
```

In this screenshot, you can see a message of the RESEND MESSAGE type, followed by a message of the SEND FILE type. The other user has received a bad file, so he sends a signal that only this signal should be sent.He sends a RESEND MESSAGE with the sequence number of the same packet that was bad.


Start
Initialize variables: sequenceNumber, windowSize, pendingFragments, etc.
Send initial window of packets(4)
Wait for ACK or Timeout
ACK Received
Out of Order ACK
Timeout
Process ACK
Reorder fragments
Check for missing fragments
Remove acknowledged fragment from pending list
Retransmit missing fragments
Adjust window size (increase or decrease based on RTT)
All fragments acknowledged?
Yes
No
End Transmission
More packets to send?
No
Yes
Wait for pending fragments
Slide window and send next packets

Description of the method for checking the integrity of the transmitted message

My protocol header will include CRC32 to calculate the checksum, which has both advantages and disadvantages. In my case, CRC32 is preferable to CRC16 because I can use the Window Scaling field to increase the number of bytes transmitted. This allows me to transmit data up to 16,776,960 bytes, which is a significant amount of data. Given the larger data size, using CRC16 increases the risk of checksum collisions. CRC32 provides a much larger checksum space, which greatly reduces the chance of errors, which is why I chose it to ensure reliable data integrity during transmission.' But it also has the disadvantage of increasing the size of the field in the protocol header. [6]

How is it calculated?

1) Find the length of the divisor 'L'.
2) Append 'L-1' bits to the original message
3) Perform binary division operation.
4) Remainder of the division = CRC

My program will use the following equation to calculate CRC32:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

In the documentation, I will give an example with a smaller CRC, because CRC32 has the same progression as the others, but it has very large numbers.

Word: 1101101 [8]

Divisor: 10101

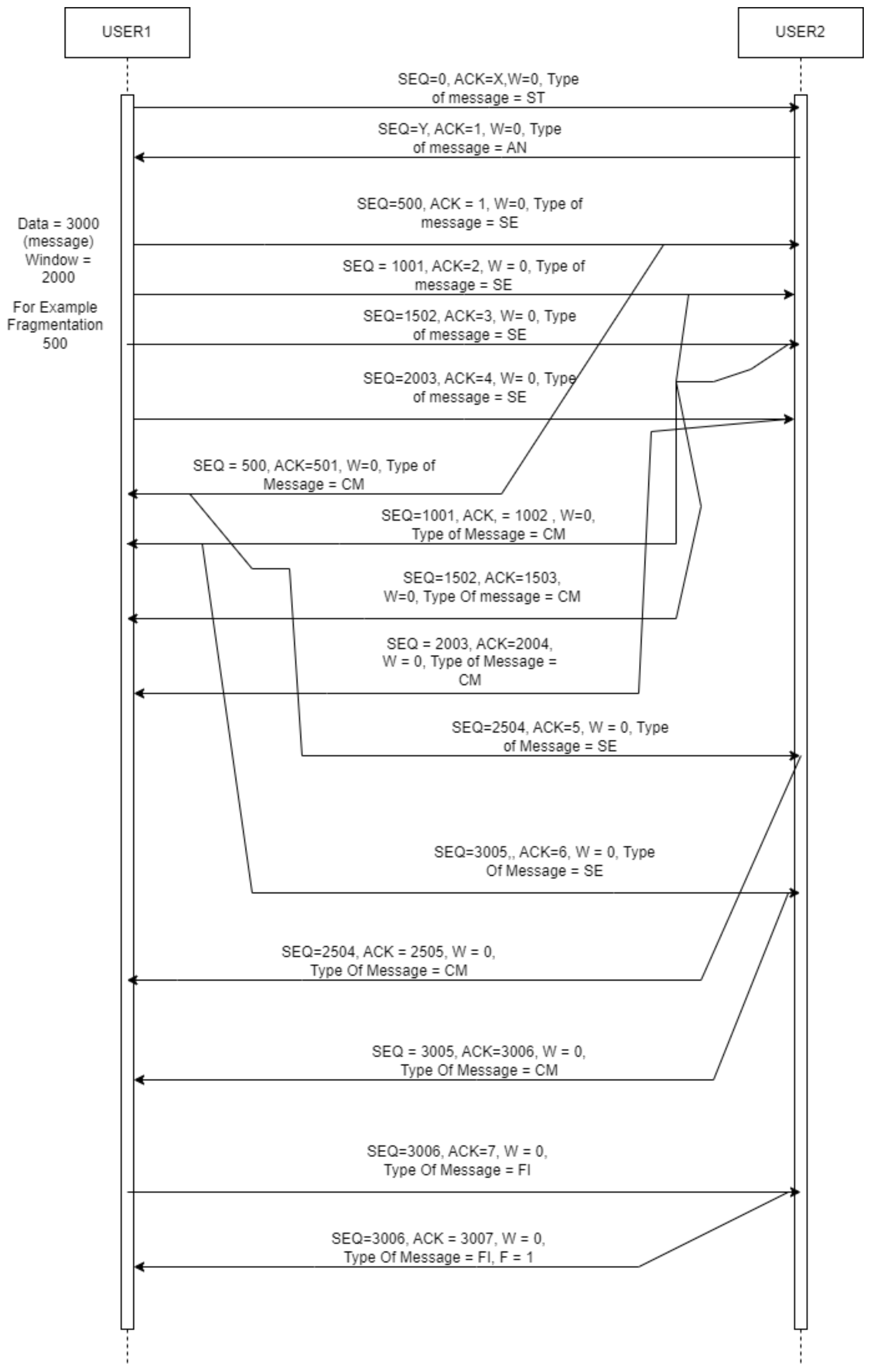
```
              1110111
      10101 | 11011010000
              10110
                ↓
              Ø11100
                10101
                  ↓
                Ø10011
                  10101
                    ↓
                  Ø01100
                    00000
                    ↓
                    Ø11000
                      10101
                ↓
                      Ø11010
                        10101
                  ↓
                        Ø11110
                          10101
                ↓
                       1011
```

The answer is 1011. In order to check the received data, I do the same thing and then check the given number with the one we received.

Correction:

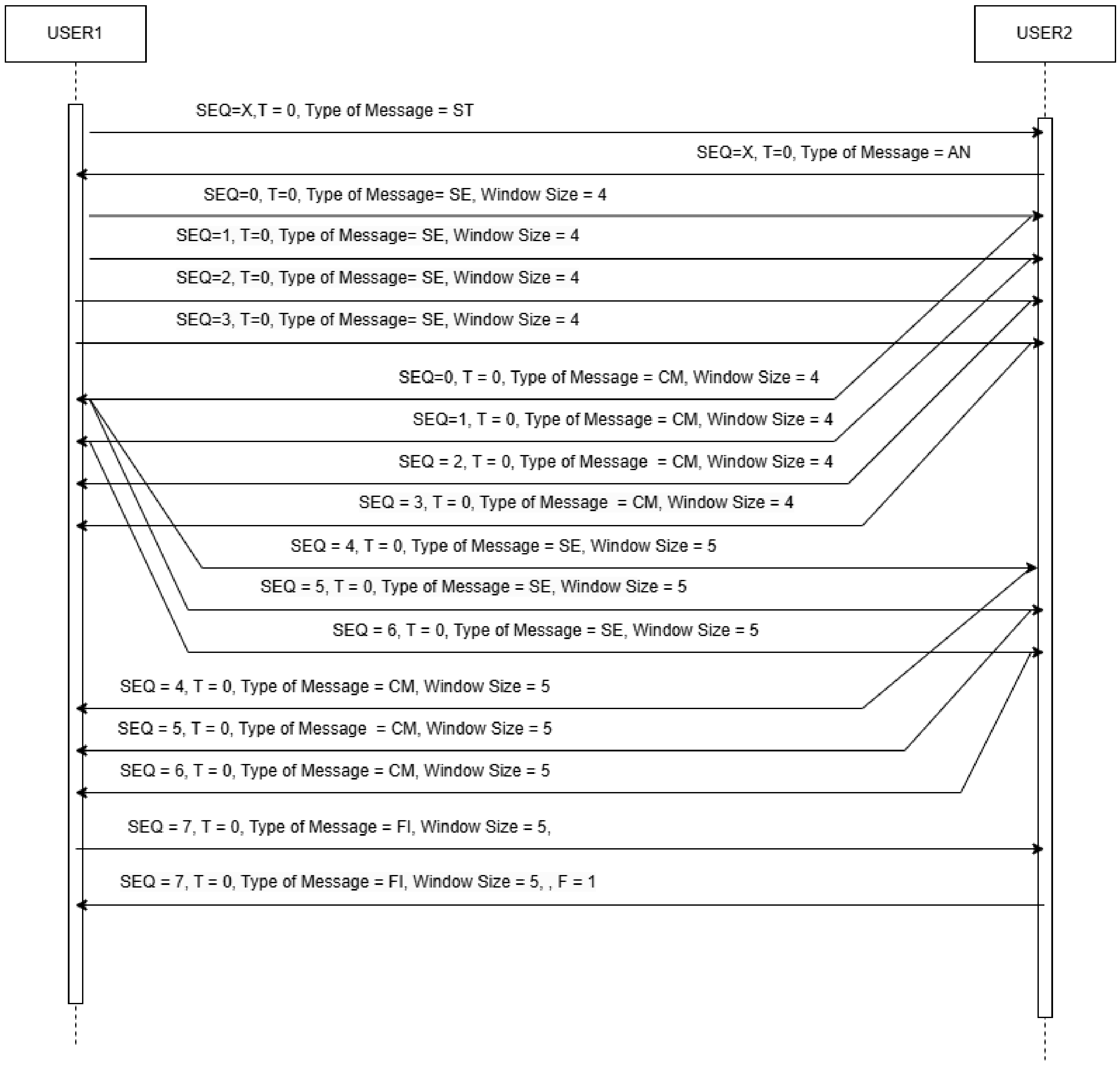
When the user receives a broken file, he sends a RESEND MESSAGE message to the sender with the sequence numberr of the packet that is broken. On the other side, this user receives this packet and finds the fragment whose sequence number is equal to the one that came to us from the RESEND MESSAGE.

A diagram describing the expected behavior of my nodes


USER1
USER2
SEQ=0, ACK=X,W=0, Type of message = ST
SEQ=Y, ACK=1, W=0, Type of message = AN
Data = 3000 (message) Window = 2000
For Example Fragmentation 500
SEQ=500, ACK = 1, W=0, Type of message = SE
SEQ = 1001, ACK=2, W = 0, Type of message = SE
SEQ=1502, ACK=3, W= 0, Type of message = SE
SEQ=2003, ACK=4, W= 0, Type of message = SE
SEQ = 500, ACK=501, W=0, Type of Message = CM
SEQ=1001, ACK, = 1002 , W=0, Type of Message = CM
SEQ=1502, ACK=1503, W=0, Type Of message = CM
SEQ = 2003, ACK=2004, W = 0, Type of Message = CM
SEQ=2504, ACK=5, W = 0, Type of Message = SE
SEQ=3005,, ACK=6, W = 0, Type Of Message = SE
SEQ=2504, ACK = 2505, W = 0, Type Of Message = CM
SEQ = 3005, ACK=3006, W = 0, Type Of Message = CM
SEQ=3006, ACK=7, W = 0, Type Of Message = FI
SEQ=3006, ACK = 3007, W = 0, Type Of Message = FI, F = 1

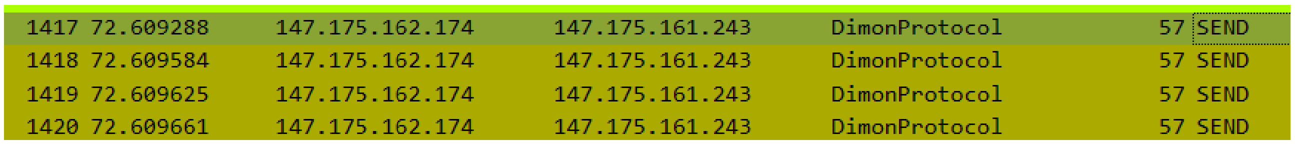
The data transfer programme starts with the user who wants to send data sending an initial packet to test the connection by sending the W flag, which is responsible for Window Scaling (if 1, it means that he would like to use this function), and with the message type ST (meaning that it is the beginning). The other user then sends a confirmation of receipt of the start packet, with the message type AN (responsible for confirming the start packet). After that, the basic data transfer process begins. For example, we have Data = 3000 bytes, at the start the window is set to 2000 by default, but usually in my program, it will change during the program if it sees that the load on the system of both users is not large. I send packets with the message type SE. After the number of bytes specified in the window has been sent, the other user will start confirming them. In this confirmation, the message type will be CM. This process will continue until we run out of data to send. After that, the user who sent the data sends a message with the FI type, indicating the end of the transmission. The other user receives this message and sends its own confirmation of completion (it will have message type FI and set the F 1 flag)

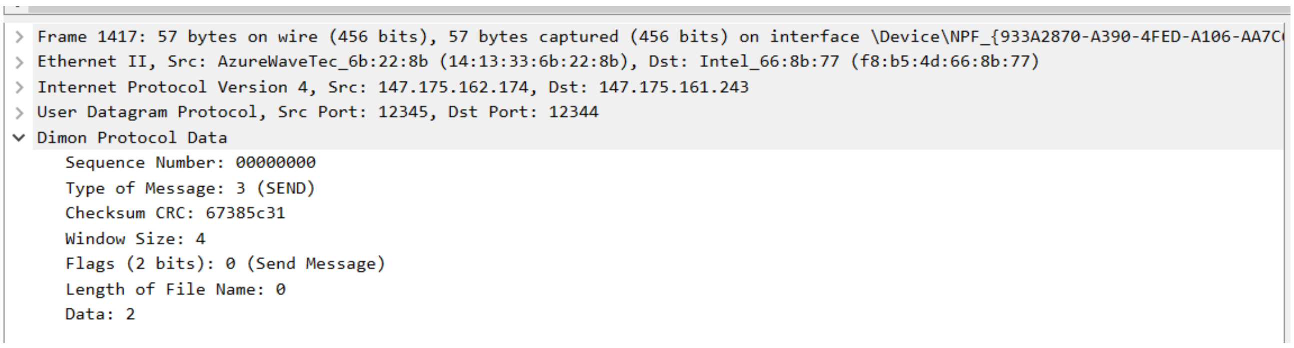
Correction:


USER1
USER2
SEQ=X,T = 0, Type of Message = ST
SEQ=X, T=0, Type of Message = AN
SEQ=0, T=0, Type of Message= SE, Window Size = 4
SEQ=1, T=0, Type of Message= SE, Window Size = 4
SEQ=2, T=0, Type of Message= SE, Window Size = 4
SEQ=3, T=0, Type of Message= SE, Window Size = 4
SEQ=0, T = 0, Type of Message = CM, Window Size = 4
SEQ=1, T = 0, Type of Message = CM, Window Size = 4
SEQ = 2, T = 0, Type of Message = CM, Window Size = 4
SEQ = 3, T = 0, Type of Message = CM, Window Size = 4
SEQ = 4, T = 0, Type of Message = SE, Window Size = 5
SEQ = 5, T = 0, Type of Message = SE, Window Size = 5
SEQ = 6, T = 0, Type of Message = SE, Window Size = 5
SEQ = 4, T = 0, Type of Message = CM, Window Size = 5
SEQ = 5, T = 0, Type of Message = CM, Window Size = 5
SEQ = 6, T = 0, Type of Message = CM, Window Size = 5
SEQ = 7, T = 0, Type of Message = FI, Window Size = 5,
SEQ = 7, T = 0, Type of Message = FI, Window Size = 5, , F = 1

In the final version, as I wrote earlier, seq now contains the fragment number, not the number of transmitted bits. Also, when I receive a message acknowledgement, I check if I can increase the window size, and if so, I increase it. In general, the entire transmission process can be said to have remained unchanged, except for Window and seq number.

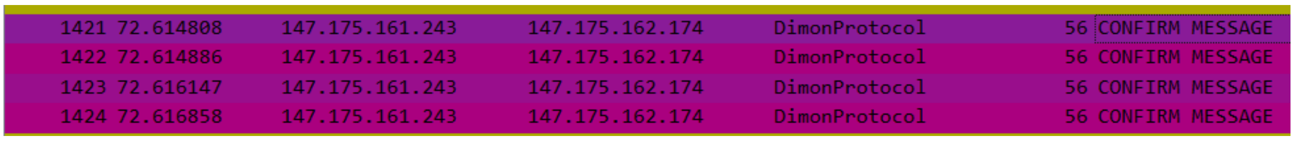
Wireshark


```
1417 72.609288    147.175.162.174    147.175.161.243    DimonProtocol       57 SEND
1418 72.609584    147.175.162.174    147.175.161.243    DimonProtocol       57 SEND
1419 72.609625    147.175.162.174    147.175.161.243    DimonProtocol       57 SEND
1420 72.609661    147.175.162.174    147.175.161.243    DimonProtocol       57 SEND
```

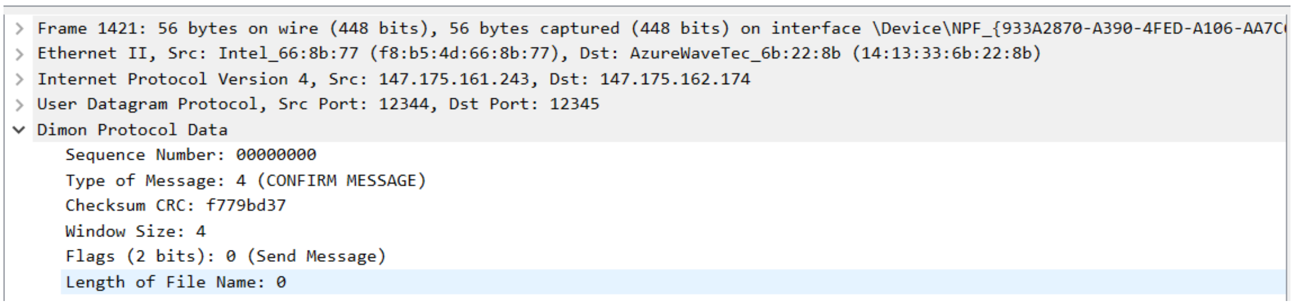


```
> Frame 1417: 57 bytes on wire (456 bits), 57 bytes captured (456 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 00000000
    Type of Message: 3 (SEND)
    Checksum CRC: 67385c31
    Window Size: 4
    Flags (2 bits): 0 (Send Message)
    Length of File Name: 0
    Data: 2
```

In this screenshot, we can see the SEND message, which is responsible for sending plain text. If we look at the details of what it contains, we can see that the first fragment is being sent, because the sequence number is 0, and we can also see that the initial window size is 4, but it will gradually increase or decrease.



```
 1421 72.614808   147.175.161.243   147.175.162.174   DimonProtocol       56 CONFIRM MESSAGE
 1422 72.614886   147.175.161.243   147.175.162.174   DimonProtocol       56 CONFIRM MESSAGE
 1423 72.616147   147.175.161.243   147.175.162.174   DimonProtocol       56 CONFIRM MESSAGE
 1424 72.616858   147.175.161.243   147.175.162.174   DimonProtocol       56 CONFIRM MESSAGE
```

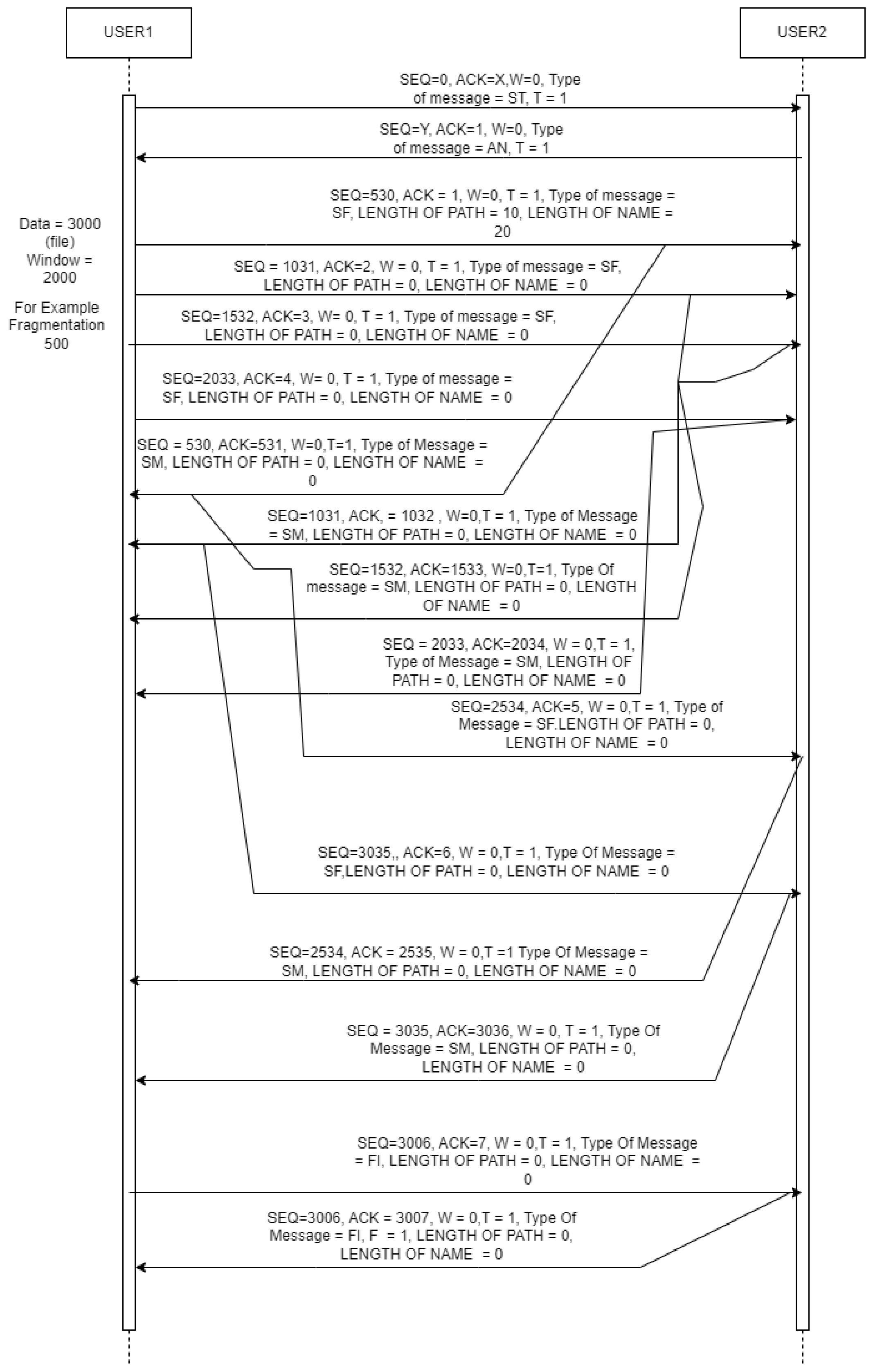


```
> Frame 1421: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: Intel_66:8b:77 (f8:b5:4d:66:8b:77), Dst: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b)
> Internet Protocol Version 4, Src: 147.175.161.243, Dst: 147.175.162.174
> User Datagram Protocol, Src Port: 12344, Dst Port: 12345
v Dimon Protocol Data
    Sequence Number: 00000000
    Type of Message: 4 (CONFIRM MESSAGE)
    Checksum CRC: f779bd37
    Window Size: 4
    Flags (2 bits): 0 (Send Message)
    Length of File Name: 0
```

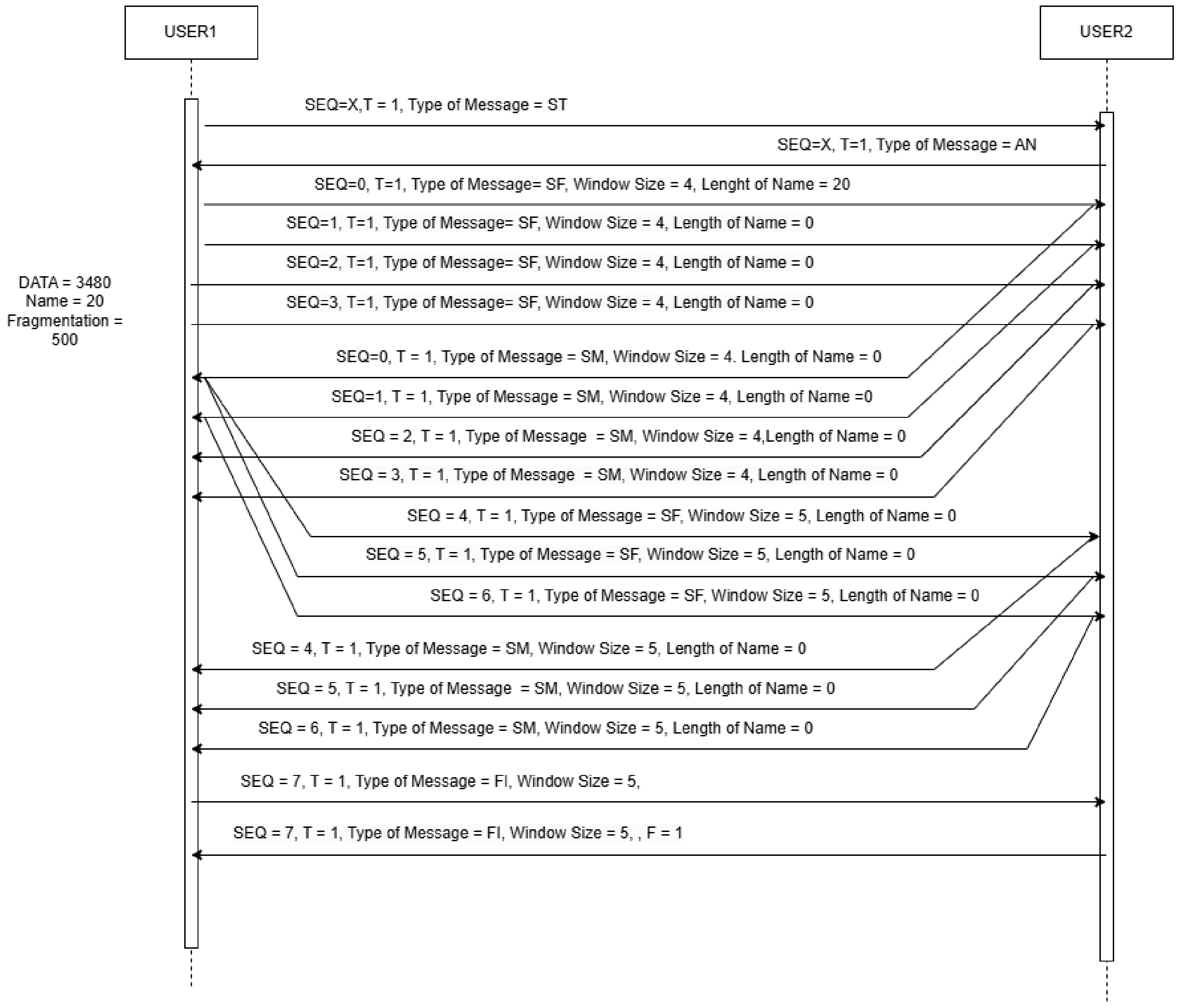
In this screenshot, we can see a message of the CONFIRM MESSAGE type, it is responsible for confirming the message. The sequence number indicates the sequence number of the received packet (which must be confirmed). After the user receives this packet, he can continue to send another fragment, plus he can adjust the window size by increasing or decreasing it.

Example with file transfer

In general, the process of transferring a file is almost identical to that of a string message, but there is a slight difference. First of all, I set the T flag to 1, which means that this session will transfer a file, not a message. Then, in the first packet I send, I will send the name of the file and the route where I should put it. I will also change the Type of message to SF when sending the file, and the Type of message to SM when confirming it. After sending the route and the file name, I set the Length of Name and Length of Path to 0, which indicates that only data is being sent.

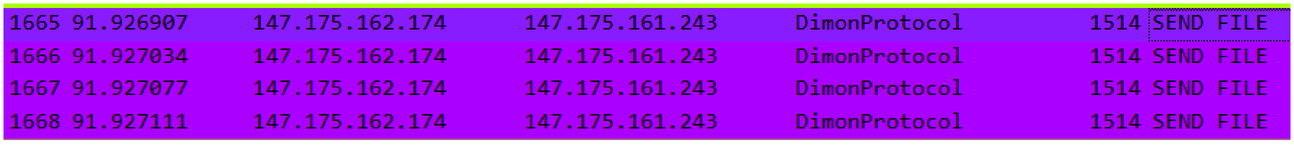

USER1
USER2
SEQ=0, ACK=X,W=0, Type of message = ST, T = 1
SEQ=Y, ACK=1, W=0, Type of message = AN, T = 1
SEQ=530, ACK = 1, W=0, T = 1, Type of message = SF, LENGTH OF PATH = 10, LENGTH OF NAME = 20
Data = 3000 (file) Window = 2000
For Example Fragmentation 500
SEQ = 1031, ACK=2, W = 0, T = 1, Type of message = SF, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=1532, ACK=3, W= 0, T = 1, Type of message = SF, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=2033, ACK=4, W= 0, T = 1, Type of message = SF, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ = 530, ACK=531, W=0,T=1, Type of Message = SM, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=1031, ACK, = 1032 , W=0,T = 1, Type of Message = SM, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=1532, ACK=1533, W=0,T=1, Type Of message = SM, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ = 2033, ACK=2034, W = 0,T = 1, Type of Message = SM, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=2534, ACK=5, W = 0,T = 1, Type of Message = SF.LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=3035,, ACK=6, W = 0,T = 1, Type Of Message = SF,LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=2534, ACK = 2535, W = 0,T =1 Type Of Message = SM, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ = 3035, ACK=3036, W = 0, T = 1, Type Of Message = SM, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=3006, ACK=7, W = 0,T = 1, Type Of Message = FI, LENGTH OF PATH = 0, LENGTH OF NAME = 0
SEQ=3006, ACK = 3007, W = 0,T = 1, Type Of Message = FI, F = 1, LENGTH OF PATH = 0, LENGTH OF NAME = 0

Correction:

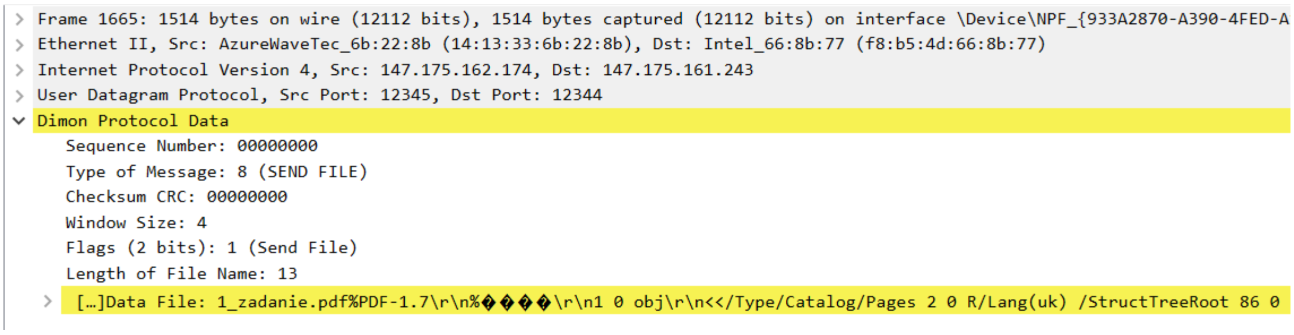

USER1
USER2
SEQ=X,T = 1, Type of Message = ST
SEQ=X, T=1, Type of Message = AN
SEQ=0, T=1, Type of Message= SF, Window Size = 4, Lenght of Name = 20
SEQ=1, T=1, Type of Message= SF, Window Size = 4, Length of Name = 0
SEQ=2, T=1, Type of Message= SF, Window Size = 4, Length of Name = 0
DATA = 3480
Name = 20
Fragmentation = 500
SEQ=3, T=1, Type of Message= SF, Window Size = 4, Length of Name = 0
SEQ=0, T = 1, Type of Message = SM, Window Size = 4. Length of Name = 0
SEQ=1, T = 1, Type of Message = SM, Window Size = 4, Length of Name =0
SEQ = 2, T = 1, Type of Message = SM, Window Size = 4,Length of Name = 0
SEQ = 3, T = 1, Type of Message = SM, Window Size = 4, Length of Name = 0
SEQ = 4, T = 1, Type of Message = SF, Window Size = 5, Length of Name = 0
SEQ = 5, T = 1, Type of Message = SF, Window Size = 5, Length of Name = 0
SEQ = 6, T = 1, Type of Message = SF, Window Size = 5, Length of Name = 0
SEQ = 4, T = 1, Type of Message = SM, Window Size = 5, Length of Name = 0
SEQ = 5, T = 1, Type of Message = SM, Window Size = 5, Length of Name = 0
SEQ = 6, T = 1, Type of Message = SM, Window Size = 5, Length of Name = 0
SEQ = 7, T = 1, Type of Message = FI, Window Size = 5,
SEQ = 7, T = 1, Type of Message = FI, Window Size = 5, , F = 1

In the final version, as I wrote earlier, seq now contains the fragment number, not the number of transmitted bits. Also, when I receive a message acknowledgement, I check if I can increase the window size, and if so, I increase it. We also changed the file transfer logic a bit. After I removed the Length of Path field, my transfer is done so that the name is first written to the initial fragment, after I write it down, I check if I can still fit the file itself, if so, the file content is transferred along with the name, if there is no space, the name itself is transferred, using the Length of Name field, I can find out where the file name is when I receive it. Also, if the file name is too large for one fragment, it will also be fragmented and sent in other packets, and the name length will be set to the part I am sending. In general, the entire transmission process can be said to have remained unchanged, except for Window and seq number.

Wireshark

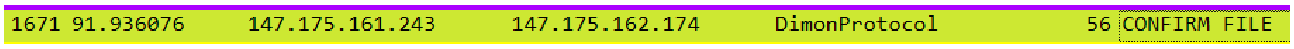


```
1665 91.926907    147.175.162.174    147.175.161.243    DimonProtocol    1514 SEND FILE
1666 91.927034    147.175.162.174    147.175.161.243    DimonProtocol    1514 SEND FILE
1667 91.927077    147.175.162.174    147.175.161.243    DimonProtocol    1514 SEND FILE
1668 91.927111    147.175.162.174    147.175.161.243    DimonProtocol    1514 SEND FILE
```

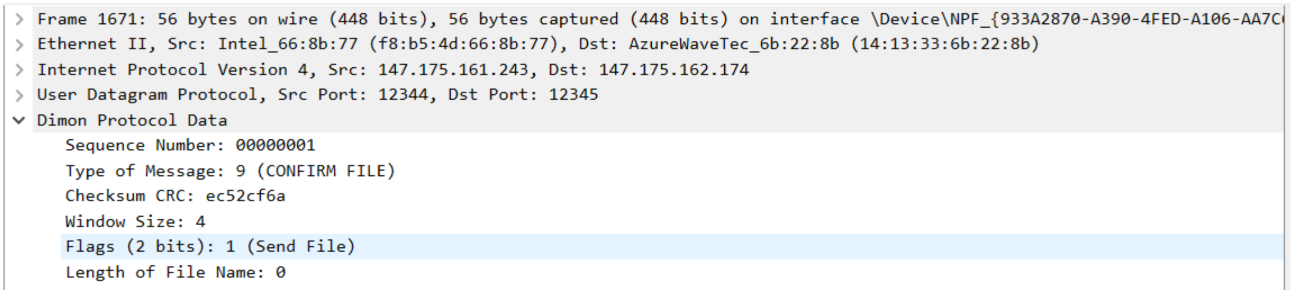


```
> Frame 1665: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface \Device\NPF_{933A2870-A390-4FED-A
> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 00000000
    Type of Message: 8 (SEND FILE)
    Checksum CRC: 00000000
    Window Size: 4
    Flags (2 bits): 1 (Send File)
    Length of File Name: 13
  > […]Data File: 1_zadanie.pdf%PDF-1.7\r\n%����\r\n1 0 obj\r\n<</Type/Catalog/Pages 2 0 R/Lang(uk) /StructTreeRoot 86 0
```

In this screenshot, we can see the SEND FILE message, which is responsible for sending plain text. If we look at the details of what it contains, we can see that the first chunk is being sent because the sequence number is 0, and we can also see that the initial window size is 4, but it will gradually increase or decrease. We can also see that length of name = 13, which means that the first 13 bytes are the name of the file, and the rest is the file itself. Also, flag is set to 1, which indicates that we are passing a flag.

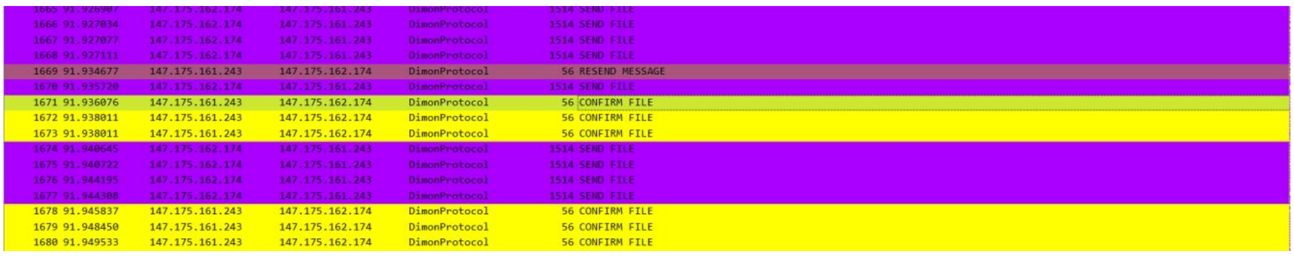


```
1671 91.936076    147.175.161.243    147.175.162.174    DimonProtocol    56 CONFIRM FILE
```



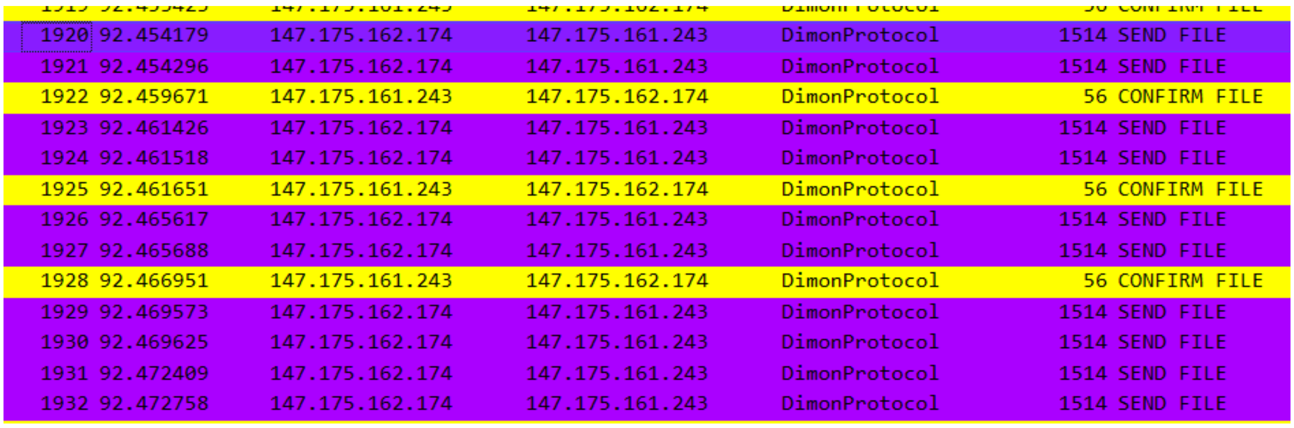
```
> Frame 1671: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: Intel_66:8b:77 (f8:b5:4d:66:8b:77), Dst: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b)
> Internet Protocol Version 4, Src: 147.175.161.243, Dst: 147.175.162.174
> User Datagram Protocol, Src Port: 12344, Dst Port: 12345
v Dimon Protocol Data
    Sequence Number: 00000001
    Type of Message: 9 (CONFIRM FILE)
    Checksum CRC: ec52cf6a
    Window Size: 4
    Flags (2 bits): 1 (Send File)
    Length of File Name: 0
```

In this screenshot, we see a message of the CONFIRM FILE type, it is responsible for confirming a message of the file type. The sequence number indicates the sequence number of the received packet (which must be confirmed). After the user receives this packet, he can continue to send the next fragment and also adjust the size of the window by increasing or decreasing it. Also, the flag is set to 1, which indicates that the file is being confirmed.

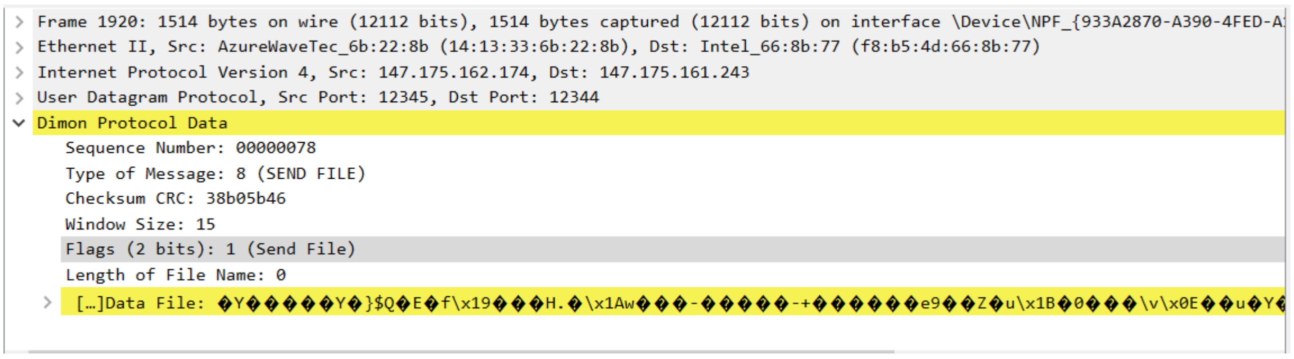


```
1665 91.926907   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1666 91.927034   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1667 91.927077   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1668 91.927111   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1669 91.934677   147.175.161.243   147.175.162.174   DimonProtocol     56 RESEND MESSAGE
1670 91.935720   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1671 91.936076   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1672 91.938011   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1673 91.938011   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1674 91.940645   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1675 91.940722   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1676 91.944195   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1677 91.944308   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1678 91.945837   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1679 91.948450   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1680 91.949533   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
```

This screenshot shows how selective repeat works.



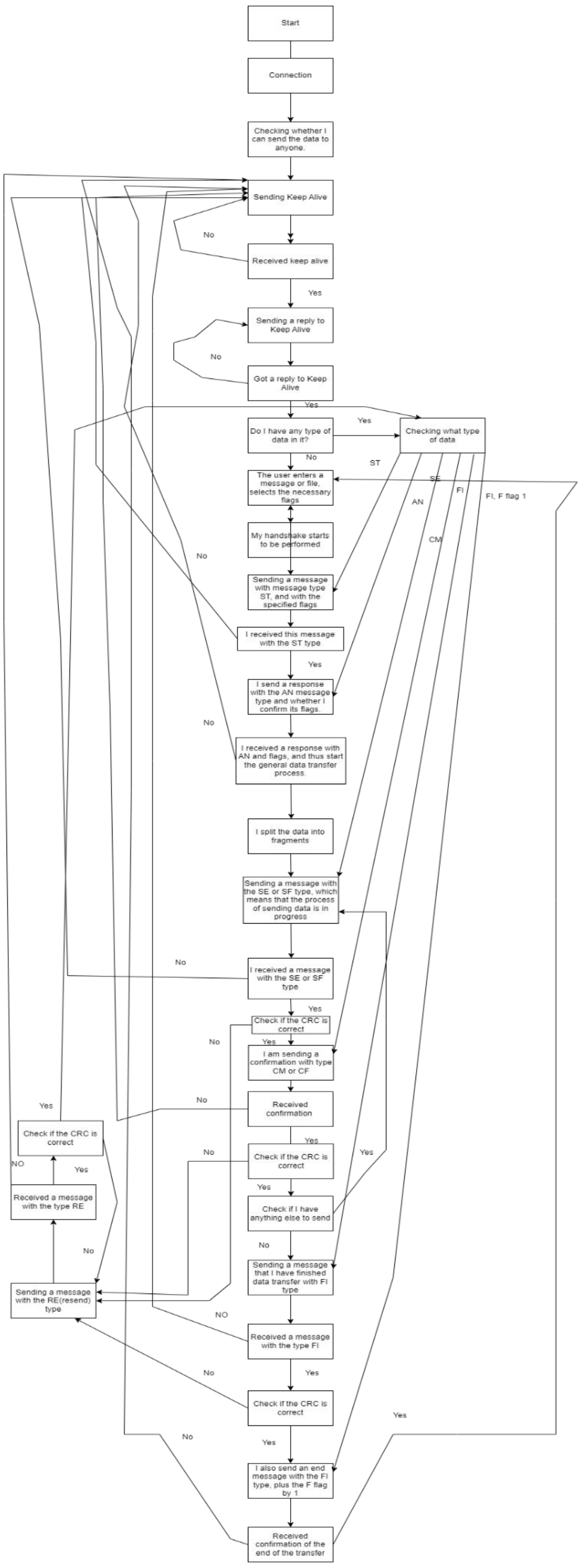
```
1920 92.454179   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1921 92.454296   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1922 92.459671   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1923 92.461426   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1924 92.461518   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1925 92.461651   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1926 92.465617   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1927 92.465688   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1928 92.466951   147.175.161.243   147.175.162.174   DimonProtocol     56 CONFIRM FILE
1929 92.469573   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1930 92.469625   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1931 92.472409   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
1932 92.472758   147.175.162.174   147.175.161.243   DimonProtocol   1514 SEND FILE
```



```
> Frame 1920: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface \Device\NPF_{933A2870-A390-4FED-A
> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 00000078
    Type of Message: 8 (SEND FILE)
    Checksum CRC: 38b05b46
    Window Size: 15
    Flags (2 bits): 1 (Send File)
    Length of File Name: 0
  > […]Data File: �Y�����Y�}$Q�E�f\x19���H.�\x1Aw���-�����-+������e9��Z�u\x1B�0���\v\x0E��u�Y�
```

In this screenshot, we can see that the window size is set to 15, which indicates that we are transferring fragments quickly.

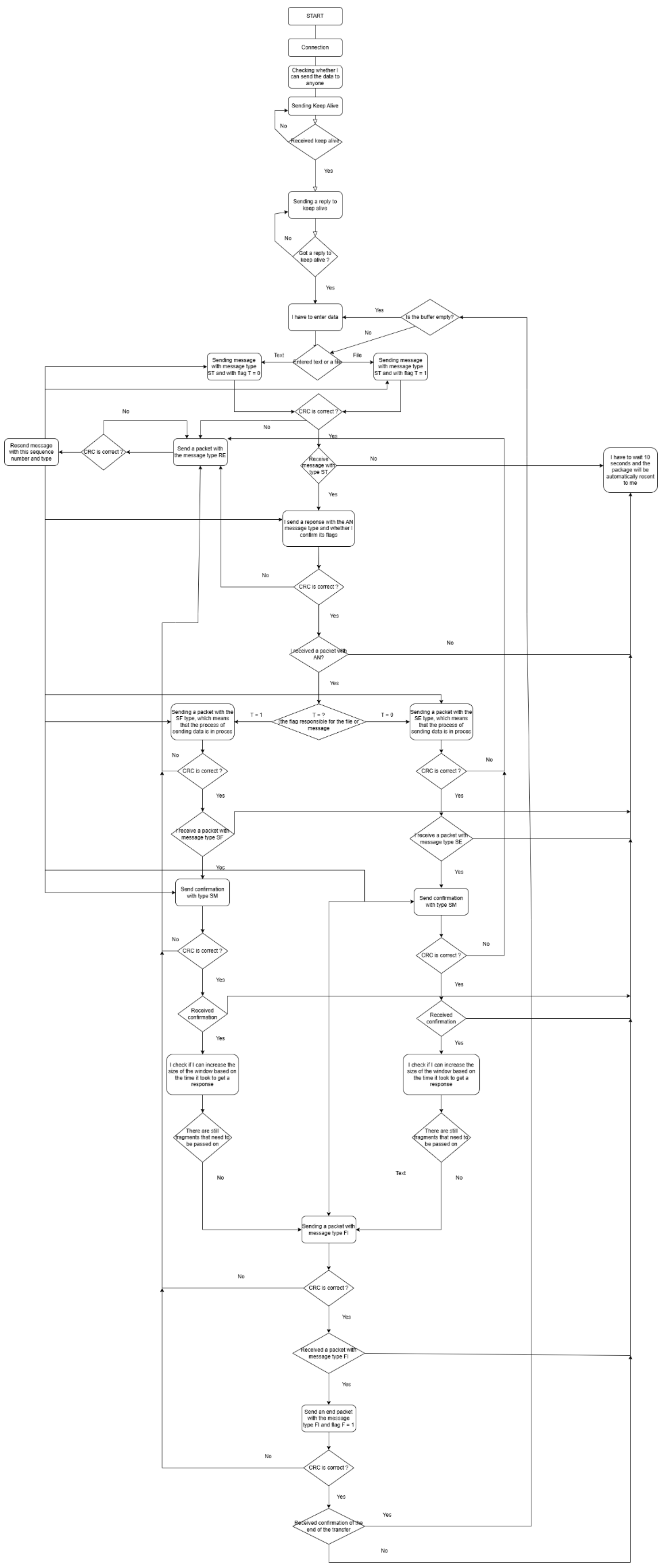
State diagram


Start
Connection
Checking whether I can send the data to anyone.
Sending Keep Alive
No
Received keep alive
Yes
Sending a reply to Keep Alive
No
Got a reply to Keep Alive
Yes
Do I have any type of data in it?
Yes
Checking what type of data
No
ST
SE
FI
FI, F flag 1
AN
CM
The user enters a message or file, selects the necessary flags
My handshake starts to be performed
No
Sending a message with message type ST, and with the specified flags
I received this message with the ST type
Yes
I send a response with the AN message type and whether I confirm its flags.
No
I received a response with AN and flags, and thus start the general data transfer process.
I split the data into fragments
Sending a message with the SE or SF type, which means that the process of sending data is in progress
No
I received a message with the SE or SF type
Yes
Check if the CRC is correct
No
Yes
I am sending a confirmation with type CM or CF
No
Received confirmation
Yes
Check if the CRC is correct
Yes
NO
Yes
No
Check if the CRC is correct
Yes
Yes
Received a message with the type RE
Check if I have anything else to send
No
No
Sending a message with the RE(resend) type
Sending a message that I have finished data transfer with FI type
NO
Received a message with the type FI
No
Yes
Check if the CRC is correct
Yes
No
Yes
I also send an end message with the FI type, plus the F flag by 1
Received confirmation of the end of the transfer

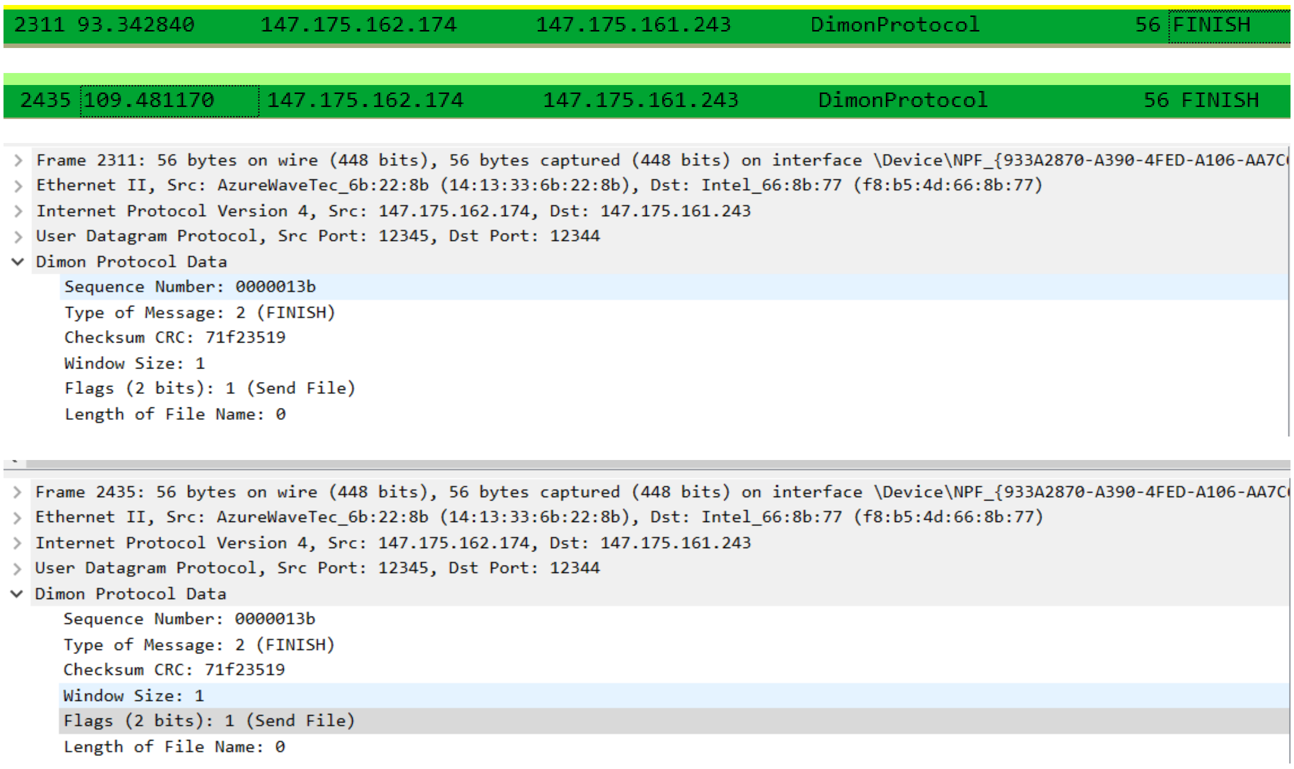
My protocol with UDP will work as follows: at the start, when connecting to the socket, I send a keep-alive signal to check if there is another user to send data to. The signal will be sent every 5 seconds until there is another user. When another user connects, he will receive this signal and send a response. When the first one receives it, he can send a message or a file (at the moment, only message transmission is ready). A user who wants to write something, he sends a message, and with what size he wants to send fragments. After that, I fragment the message, and the transmission of these fragments starts (currently, only the Stop And Wait mechanism is implemented, but Selective Repeat ARQ is planned). When a fragment is sent, it is first checked for CRC, if the CRC does not match, a response is sent that it is not the correct fragment, and then resend occurs. When a successful response is sent with the acceptance of the file, I first check the CRC and then the acknowledgment number to see how many have arrived. When all the fragments are sent, the user is checked to see if he has any more, and if not, a final request is sent. The other user receives it, and he sends a response, but with the F=1 flag, which means it's over. I also have a process with inactivity. If the user does not receive a response within 15 seconds, a keep-alive signal will be sent, which will be sent 5 times within 5 seconds. If the user does not receive a response within this time, the connection is closed.

Correction:

In the final version, I have already implemented Selective ARQ with a dynamically sliding window and new functionality. Now I check every time whether the packet has been sent and confirmed, and after 10 seconds, if it is still in the table that contains it, we will forward it. A dynamically sliding window has also been implemented to increase the number of transmitted fragments without acknowledgement. To do this, it uses the RTT formula.


START
Connection
Checking whether I can send the data to anyone
Sending Keep Alive
Received keep alive
No
Yes
Sending a reply to keep alive
Got a reply to keep alive ?
No
Yes
I have to enter data
Is the buffer empty?
Yes
No
Entered text or a file
Text
File
Sending message with message type ST and with flag T = 0
Sending message with message type ST and with flag T = 1
CRC is correct ?
No
No
Yes
Send a packet with the message type RE
CRC is correct ?
Resend message with this sequence number and type
Receive message with type ST
No
I have to wait 10 seconds and the package will be automatically resent to me
Yes
I send a reponse with the AN message type and whether I confirm its flags
CRC is correct ?
No
Yes
I received a packet with AN?
No
Yes
T = ?
The flag responsible for the file or message
T = 1
T = 0
Sending a packet with the SF type, which means that the process of sending data is in proces
Sending a packet with the SE type, which means that the process of sending data is in proces
CRC is correct ?
No
CRC is correct ?
No
Yes
Yes
I receive a packet with message type SF
I receive a packet with message type SE
Yes
Yes
Send confirmation with type SM
Send confirmation with type SM
CRC is correct ?
No
CRC is correct ?
No
Yes
Yes
Received confirmation
Received confirmation
Yes
Yes
I check if I can increase the size of the window based on the time it took to get a response
I check if I can increase the size of the window based on the time it took to get a response
There are still fragments that need to be passed on
There are still fragments that need to be passed on
No
Text
No
Sending a packet with message type FI
CRC is correct ?
No
Yes
Received a packet with message type FI
Yes
Send an end packet with the message type FI and flag F = 1
CRC is correct ?
No
Yes
Received confirmation of the end of the transfer
Yes
No

Wireshark



```
2311 93.342840      147.175.162.174      147.175.161.243      DimonProtocol      56 FINISH
```

```
2435 109.481170     147.175.162.174      147.175.161.243      DimonProtocol      56 FINISH
```

```
> Frame 2311: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 0000013b
    Type of Message: 2 (FINISH)
    Checksum CRC: 71f23519
    Window Size: 1
    Flags (2 bits): 1 (Send File)
    Length of File Name: 0
```

```
> Frame 2435: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface \Device\NPF_{933A2870-A390-4FED-A106-AA7C
> Ethernet II, Src: AzureWaveTec_6b:22:8b (14:13:33:6b:22:8b), Dst: Intel_66:8b:77 (f8:b5:4d:66:8b:77)
> Internet Protocol Version 4, Src: 147.175.162.174, Dst: 147.175.161.243
> User Datagram Protocol, Src Port: 12345, Dst Port: 12344
v Dimon Protocol Data
    Sequence Number: 0000013b
    Type of Message: 2 (FINISH)
    Checksum CRC: 71f23519
    Window Size: 1
    Flags (2 bits): 1 (Send File)
    Length of File Name: 0
```

In these screenshots, we can see that the file transfer is ending because the message type is FINISH and the flag is set to 1. The first screenshot shows the packet transmitted by the user who sent the file, and the second one shows the user who confirmed it.

Conclusion

My main task was to create a protocol header above UDP. It included Sequence number (a sequence number is a counter used to keep track of each byte sent out by the host), Acknowledgment number (an acknowledgement number is sent by the TCP server to indicate that it has received the accumulated data and is ready for the next segment), Message type (what message the user sent and what was the response from the person who received it), CRC32 (used to check data integrity), Window (A window is the amount of data (in bytes) that can be buffered during a connection, that can be buffered during a connection), Window Scale (Scaling the window effectively increases the capacity of my Window field from 16 to about 24 bits), Flag (The first is N (next chunk), W (to confirm that the window will be used), T (file or message), F (final or not)), Path length (This field will be responsible for the size in bytes of the path the file should follow), Filename length (This field, like the Path length field, will be responsible for the length, but now not of the path, but of the filename itself). The application will implement Selective Repeat ARQ, and already uses a 2nd source handshake.

Correction:
My main task was to create a protocol header over UDP. In the final version of my protocol, it contains such fields as: sequence number, message type, crc32, window, FL(flags) and file name length. Now window doesn't tell you how many bytes the user can send without acknowledgement, but how many fragments he can send without acknowledgement. The number of flags has been reduced to 2: T (whether a file or a regular message), F (final message). 2 fields have been removed: Window scaling, Length of Path. Also, the W flag, which was responsible for Window scaling, has been removed and the size of the message type field has been reduced to 1 byte. The program implements a selective arq with a dynamically sliding window, which uses the RTT method for its operation.

Sources

1) https://books.google.sk/books?hl=uk&lr=&id=X-l9NX3iemAC&oi=fnd&pg=PR9&dq=TCP+&ots=Z5ekk4LU5U&sig=KWAEFO26kehPWsJxQdQp9g4BudU&redir_esc=y#v=onepage&q=TCP&f=false
2) https://github.com/fiit-ba/PKS-course-2425/blob/main/materials/analyze_frames.pdf
3) https://books.google.sk/books?hl=uk&lr=&id=wqabAgAAQBAJ&oi=fnd&pg=PR7&dq=TCP&ots=zTRZwRulP8&sig=4L9DI3DjJkwMn8YdQe3CEpe9wHM&redir_esc=y#v=onepage&q&f=false
4) https://www.google.sk/books/edition/TCP_IP_Illustrated/X-l9NX3iemAC?hl=uk&gbpv=1
5) https://dl.acm.org/doi/epdf/10.1145/505696.505703
6) https://ieeexplore.ieee.org/abstract/document/1287910
7) https://www.tutorialspoint.com/what-is-selective-repeat-arq-in-computer-networks
8) https://www.youtube.com/watch?v=ZJH0KT6c0B0&ab_channel=TheBootStrappers
9) https://tcpcc.systemsapproach.org/algorithm.html