

Department of Computer Science
College of Engineering
University of the Philippines Diliman

CS 145 Project 1 Documentation: Parameter-Adaptive Reliable UDP- based Protocol

In Partial Fulfillment of the Requirements for the Course
CS 145: Computer Networks

Submitted by:
Rule, Camille P. (2019-09414)

Table of Contents

1	Introduction	2
2	Declaration of Accomplished Levels.....	2
3	Sender implementation.....	2
3.1	Importing modules (Lines 17-21).....	2
3.2	Getting parameters (Lines 29-45).....	2
3.3	Initializing a socket for the client (Lines 47-50).....	4
3.4	Level 1 Implementation: Sending intent message and getting transaction ID (Lines 60-65)	5
3.5	Opening and reading payload file (Lines 71-74).....	6
3.6	Initialization of flags for packet transmission (Lines 76-98)	7
3.7	Packet transmission implementation (Lines 102-234)	8
3.7.1	Setting initial packet size and starting transmission timer (Lines 102-106).....	9
3.7.2	Data packet creation.....	9
3.7.3	Main Loop: Check for last packet (Lines 109-139).....	10
3.7.4	Main Loop: Packet sending (Lines 145-149)	11
3.7.5	ACK message.....	12
3.7.6	Getting the ACK	12
4	Testing.....	15
4.1	Transaction 1	15
4.2	Transaction 2	19
4.3	Transaction 3.....	22
4.4	Transaction 4	25
4.5	Transaction 5.....	28
5	Conclusion.....	30

1 Introduction

The project is an implementation of the sender program for a reliable UDP-based protocol given certain hidden parameters. The assigned `___` levels further explained in the following sections of this document were successfully accomplished. The project can be found in the following [GitHub repository](#) and the corresponding video documentation can found in this [link](#).

2 Declaration of Accomplished Levels

For this project, the first two (2) levels given in the project specifications were accomplished. The following section contains a line-by-line explanation of the sender code utilized to complete this project.

- **Level 1:** Able to send Intent Message (to the receiver or test server) and receive the Accept Message (from the receiver or test server).
- **Level 2:** Can do everything that Level 1 does, plus, able to send at least 50% of the payload to the receiver or test server

3 Sender implementation

To understand how the implementation of the sender works, we explain the code line by line. Note that the `sender.py` file also has corresponding comments that can help in understanding how the code works.

3.1 Importing modules (Lines 17-21)

Several libraries were used in the project shown in Figure 1:

- Line 17 imports the `socket` module which allows for the communication of the server and client by using socket endpoints
- Line 18 imports the `sys` module, which is mainly used to get the needed parameters from the command line
- Line 19 imports `hashlib` as instructed by the given `compute_checksum` function in the project specifications
- Line 20 imports `time` which is used to keep track of socket timeouts and how long transaction time is
- Finally, Line 21 imports the `ceil` function from the `math` module used in estimating and dividing payload size

```
17 import socket
18 import sys
19 import hashlib
20 import time
21 from math import ceil
```

Figure 1: sender.py 17:21

3.2 Getting parameters (Lines 29-45)

The program is run on the terminal and certain parameters are given in the command line. These parameters are:

- Filename prefixed by the -f flag which denotes the filename of the payload
- The IP address of the server prefixed by the -a flag.
- The Port used by the server prefixed by the -s flag.
- The assigned port to be used by the student (given as a project credential) prefixed by the -c flag.
- Lastly, the unique ID assigned as a project credential as well, prefixed by the -i flag.

Figure 2 shows the code snippet of parameter getting and initialization of the program.

```

28
29 #Initialize parameters from terminal command
30 #-f path/to/file.txt -a SERVER IP ADDR -s SERVER PORT -c STUDENT PORT -i STUDENT ID
31 comms = sys.argv[1:]
32 for comm in range(len(comms)):
33     if comms[comm] == '-f': #filename
34         fn = comms[comm+1]
35     elif comms[comm] == '-a': #IP address of server
36         UDP_IP_ADDRESS = comms[comm+1]
37     elif comms[comm] == '-s': #Server port
38         UDP_PORT_NO = int(comms[comm+1])
39     elif comms[comm] == '-c': #Client port
40         CLIENT_PORT = int(comms[comm+1])
41     elif comms[comm] == '-i': #Student ID
42         ID = comms[comm+1]
43
44 #Set server IP addr and port to a tuple
45 UDP_IP_PORT = (UDP_IP_ADDRESS, UDP_PORT_NO)
46

```

Figure 2: sender.py; 29:45

Line 30 shows an example of the command line instruction used to run the program. Line 31 gets the list of command line arguments by making use of the `sys.argv` function. The output of this is an array with the arguments as elements. We slice the list starting from its first element, because element 0 will always be the name of the .py file (in this case `casender.py`).

Line 32 iterates over the array of command line arguments. Since order is not ensured when executing the command, we first check for the prefix and get the value next to it once we have encountered a match. The prefix flags for each parameter were as discussed above and in the lab specifications.

Variable	Prefix	Parameter
Fn	-f	Filename of payload data to be sent
UDP_IP_ADDRESS	-a	Server IP address
UDP_PORT_NO	-s	Server Port Number
CLIENT_PORT	-c	Student assigned port number
ID	-i	Student assigned ID

Line 45 just creates a tuple out of the receiver's IP address and Port. When sanding packets, these two values are zipped in a tuple in the `socket.sendto()` function. For brevity, we create a tuple variable of these two values in this line already.

These lines constitute getting and initializing the parameters to be used in the implementation of sender program.

3.3 Initializing a socket for the client (Lines 47-50)

The main idea of using sockets for communication over a network is by using sockets to form connections between two nodes – a server and a client. For this program, we initialize a UDP socket instance for the client to be able to receive ACK packets from the client.

```
47 #Create socket for receiving packets from server
48 clientSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
49 clientSock.bind('', 6745) #bind at port 6745 (designated port for student)
50 clientSock.settimeout(10) #set timeout to default 10 seconds
51
```

Figure 3: sender.py; 47:50

The code snippet shows the creation of the socket instance and the assignment of a port to the created socket.

Line 48 assigns the created socket instance to the variable clientSock.

- The socket() function of the socket module creates a socket object or instance. It takes in two arguments which specifies the address family and the protocol that will be used to transport the messages.
- For this program, arguments passed to the socket() function are:
 - socket.AF_INET is the address family for IPv4 messages. This specifies that the created socket instance can only communicate with IPv4 addresses.
 - socket.SOCK_DGRAM specifies that the protocol that will be used to transport messages over this socket is through a datagram-based protocol or UDP.

Line 49 binds the socket instance to the unique designated port for the student. In my case, my assigned port number is 6745. Since an IPv4 address family was used, the bind() function expects a tuple containing the IP address and port number. However, since the IP address cannot always be sure for every runtime, this is left as an empty string to be identified on the server side. This socket instance will be used to listen to ACK messages from the server once we start sending packets.

- clientSock is binded to a two-tuple (host,port) = ('', 6745) where the empty string denotes the IP address of the client (to be identified by server) and 6745 is the unique designated port number for the student.

Line 50 sets a timeout value for messages sent from the client. Note that the project entails sending packets to a server with hidden parameters, hence, some packets may not get acknowledged. Without the timeout value set, the socket will continue to listen for an ACK message from the server (which may or may not come). Hence, we must set a specified time value to instruct the socket to give up waiting and retry sending a packet again once this value has been reached. Setting a timeout value too large gives us the risk of waiting for an ACK message for too long which will compromise the payload being sent. Setting a value too small

will bypass the server’s processing time and will just retry sending packets to the server, even if the server will send an ACK message back. Hence, we settle for 10 as the timeout value which is 8.33% of 120s – the specified time limit for Level 2 and 3. Note that this value was taken from trial-and-error executions.

We started with an initial value of 30 which is 25% of 120s. This value is too big. In the whole 120s transmission time, we would only be able to retry sending packets 4 times. We tried again using 15 which is 12.5% of 120s, again, payload could not be sent efficiently. Hence, we decrease the timeout value again and settle for 10 which is 6.33% of 120s.

3.4 Level 1 Implementation: Sending intent message and getting transaction ID (Lines 60-65)

Level 1 entails for being able to send an intent message to the server to establish a connection and get a transaction ID for consequent packet transmission. Figure 4 shows a code snippet of this in the sender.py file.

```

52  """
53  LEVEL 1 IMPLEMENTATION:
54  - append "ID" to student ID
55  - encode intent message and send to server using server IP and port
56  - server sends transaction ID back
57  - decode and store in trans_id variable
58  """
59
60  intent = "ID"+ID
61  clientSock.sendto(intent.encode(), UDP_IP_PORT)
62
63  trans_id, addr = clientSock.recvfrom(1024)
64  trans_id = trans_id.decode()
65  print(f"Trans ID: {trans_id}")
66

```

Figure 4: sender.py; 52:65

Line 60 prepares the intent message according to the specified format in the project specifications, where the first two characters are the letters “ID” followed by the 8-alphanumeric string specifying the unique student ID given as a project credential. In my case, this ID is 44919a94.

I	D	W	W	W	W	W	W	W	W
0	1	2	3	4	5	6	7	8	9

Table 1: Intent message format

Hence, line 61 concatenates the prefix “ID” to my assigned student ID. The intent message generated by my socket would look like this:

I	D	4	4	9	1	9	a	9	4
0	1	2	3	4	5	6	7	8	9

Table 2: Student intent message

Line 61 uses the `sendto()` function to send the intent message over UDP to the server. The function takes in two arguments, the first being the data to be sent, and the second being the two-tuple destination of the data (server IP address and port number). Note that we first encode the intent message to a byte string, by using the `str.encode()` method since the `sendto()` function does not accept strings.

- `clientSock.sendto(intent.encode(), UDP_IP_PORT)` is used to send intent message to server, where `intent` is the message containing student unique student ID to request to establish connection, we use `intent.encode()` to convert the string into bytestring format, and `UDP_IP_PORT` is a two-tuple variable containing the server's IP address and port number.

Lines 63 and 65 is responsible for listening to and retrieving the accept message to be sent by the server. The socket module's `recvfrom()` method takes in one required parameter which denotes the number of bytes to be read from the socket. This method returns a two-tuple value, the first one being the byte object read from the UDP socket, and the second element being the address of the sender (in this case, the server).

- `clientSock.recvfrom(1024)` receives 1024 bytes of data from the server
- `trans_id, addr = clientSock.recvfrom(1024)` assigns the two-tuple value from `recvfrom()` to the variables `trans_id` and `address`. Note that `trans_id` refers to the 7-digit integer denoting the transaction ID.

We are interested in the `trans_id` variable since this will be used in the generation of our packets and subsequent packet transmissions. Note that this is received in byte string format, hence we use the `str.decode()` method in line 64 to convert this into a string format which we can use. Finally, line 65 prints the transaction ID for identification and testing later on.

These lines constitute the Level 1 implementation of the project.

3.5 Opening and reading payload file (Lines 71-74)

Since the payload to be sent would be coming from a specified file, we must be able to open and read this file to get its contents. Note that the payload file is downloaded using the `wget` command.

```
70
71  #Open payload file and parse contents
72  f = open(fn,"rb")
73  dp = f.read()
74  dp = dp.decode()
75
```

Figure 5: *sender.py*; lines 71:74

- Line 72 uses Python's `open()` function to open the file specified from its filename, with `rb` as its permissions. This opens the payload file in binary mode for reading.
- Line 73 uses Python's `read()` function to read all contents of the file into a string.
- Line 74 decodes the binary form of the data into a string version to be used in our program.

These lines constitute the opening and parsing of the payload data to be sent to the server.

3.6 Initialization of flags for packet transmission (Lines 76-98)

After receiving the transaction ID from the server, we can now begin to transmit packets. However, we must first initialize flags that will help us in our packet transmission. According to the project specifications, the format of the data packet is as follows:

I	D	W	W	W	W	W	W	W	W
0	1	2	3	4	5	6	7	8	9
S	N	X	X	X	X	X	X	X	T
10	11	12	13	14	15	16	17	18	19
X	N	Y	Y	Y	Y	Y	Y	Y	L
20	21	22	23	24	25	26	27	28	29
A	S	T	Z	PAYLOAD					
30	31	32	33						

Table 3: Data packet format

Where:

- IDWWWWWWWW denotes the unique student ID assigned prefixed by “ID”
- SNXXXXXXX denotes the sequence number of packet prefixed by “SN”
- TXNYYYYYYY denotes the transaction id received from server prefixed by “TXN”
- Z denotes the last packet in transmission
- Lastly, the payload will be at the end of the data packet.

For these reasons, we must initial values for the variables Z, sequence number, as well as other variables that will help us in our packet transmission.

```

75
76 """
77 Set appropriate flags
78 Z variable is the Z flag which denotes last packet in transmission, initially 0
79 seq variable keeps track of the sequence number of packet, initially 0
80 charsent denotes how many characters of the payload has been sent thus far, initially 0
81
82 `accepted` flag if server accepts transmission of a packet, initially 0
83 (server has not accepted any packet yet)
84
85 `increment` flag to check if client can continue incrementing payload size being sent, initially 0
86 `maxed` flag to check if no more incrementation can be done - payload size has been exceeded, initially 0
87
88 empty string `sent` to keep track of all sent data from payload
89 (will be useful to check if correct packets are sent)
90 """
91 Z = seq = charsent = 0
92 accepted = increment = maxed = 0
93 dp_size = len(dp) #get length of payload for reference
94 sent = ''
95

```

Figure 6: sender.py; 76:94

Line 91 initializes variables Z, seq, and charsent to 0. The Z variable denotes the last packet of the transmission as per Table 3. The seq variable denotes the sequence number of the packet, and charsent denotes how many characters of the payload has been sent so far.

Other flags were also utilized in the program implementation which can be seen in line 92.

- The accepted variable is a flag which tells us if the initial packet we sent was accepted. This is initialized to 0.
- The increment flag tells us if we can continue incrementing the payload size or not. This is initialized to 0.
- The maxed flag tells us if we have maxed out the payload size of the transmission (i.e. we cannot increment the payload size of the packets anymore). This is initialized to 0.

We get the size of the payload to be sent in line 93 and put it in the variable dp_size which we will use as reference for our payload divisions. Lastly, we initialize an empty string variable sent which will hold all the character we have sent so far in our transmission. This is useful for error checking if the client is sending the correct part of the payload. After initializing these variables, we can move on to sending packets.

3.7 Packet transmission implementation (Lines 102-234)

For packet transmission, we devised a pseudocode as shown below:

Pseudocode for packet transmission

```

initialize best guess packet size for initial packet
packt_size = best guess
while charsent < dp_size:
    if packet is last to transmit:
        check remaining data to be sent
        Z = 1
        payload = dp[charsent:remaining]
    else:
        payload = dp[charsent:charsent+packt_size]
    compute for checksum
    send payload to server
    if socket has not timed out:
        get ACK response from server
        accepted = 1 #server has accepted initial payload size
        compute for checksum of ACK message from server
        bf checksums not equal:
            Break #there is an error
        charsent += packet_size
        seq += 1
        if can increment pkt_size:
            increment pkt_size
        if accepted = 1 and cant increment:
            maxed = 1 #maxed out payload size

    else: #socket has timed out, no ACK from server

```

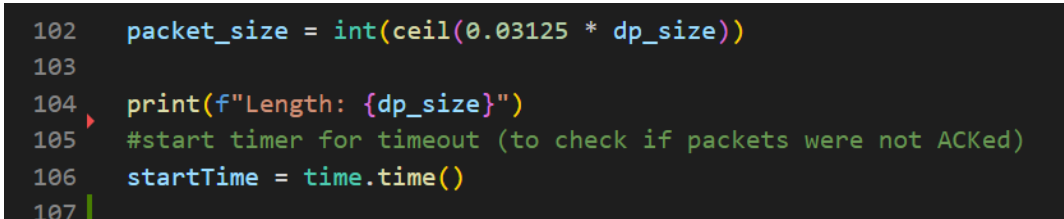
```

    if maxed: #skip iteration, we just need to resend packet
        continue
    if accepted = 0: #initial pkt_size not accepted
        decrement pkt_size
    else: #packet has been accepted but timed out because we incremented too big
        increment = 0 #can no longer increment
        decrement pkt_size
    if transmission time > 120:
        break

```

The next subsections will go over the parts of the packet transmission program and explain them in detail.

3.7.1 Setting initial packet size and starting transmission timer (Lines 102-106)



```

102     packet_size = int(ceil(0.03125 * dp_size))
103
104     print(f'Length: {dp_size}')
105     #start timer for timeout (to check if packets were not ACKed)
106     startTime = time.time()
107

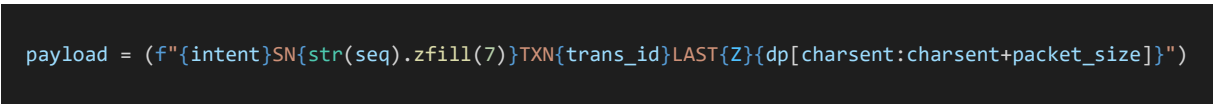
```

Figure 7: sender.py; 102-106

- Line 102 initializes the best guess packet size to be $\sim 3\%$ of the total data size. This value arose from trial-and-error executions.
 - Too big of a value, say 25%, would give too much of an overhead in terms of decrementing the packet size if it was too big.
 - Too small of a value, say 1%, would not let us deliver packets. We settle for $\sim 3\%$ of the data size instead.
- Line 106 starts the time for the transmission time. This is needed to make sure that the client stops trying to send packets even after the 120s allotted transmission time.

3.7.2 Data packet creation

According to the data packet format shown in Table 3 and as discussed in the project specifications document, there are certain parts of the data packet that we must make sure to follow, or the server will not acknowledge our packet. Standard creation of data packets in the implementation are given by:



```

payload = (f'{intent}SN{str(seq).zfill(7)}TXN{trans_id}LAST{Z}{dp[charsent:charsent+packet_size]}')

```

Figure 8: Packet creation

The payload variable holds the data packet to be sent. We use an f-string in Python to format the packet according to the specifications shown in Table 3.

- First part of the data packet is the intent message, which was formatted already (i.e. prefix “ID” added) when the transaction ID was requested from the server.
- Next, we have the sequence number prefixed with “SN” and followed by the value of the seq variable. Since the sequence number in the specifications called for a 7-digit integer, we pad our sequence number in the packet using the .zfill() function of Python.
- Next, we have the transaction ID, received from the server from the Level 1 implementation. This is prefixed with “TXN”.
- Next, we have the last packet flag Z, which is prefixed by “LAST”.
- Lastly, we have the payload to be sent, where we get the data in the data packet from the last character sent by the previous packet, up to the specified packet size.

3.7.3 Main Loop: Check for last packet (Lines 109-139)

To continuously transmit packets from the client to server, we use a while loop which keeps executing until the length of the data packet has been reached. Figure 11 shows the first part of the while loop, which constitutes the creation of the payload to be sent. Line 109 shows the while loop and its condition for execution which is if the current characters sent are less than the size of the total data packet to be sent. If there is still data to be sent to the server, we keep sending packets.

We first have a conditional statement to check if the packet to be sent is the last packet in the transmission or not. To determine if a packet is the last packet, we check the remaining data in the data packet by subtracting the number of sent characters from the total data packet size in line 114.

From here, there are two possible scenarios:

- The remaining data to be sent is exactly 0 (shown in line 116)
 - If the remaining data is exactly 0, then we set the last packet (Z) flag to 1 and get payload from the last character sent up to the last character in the data packet.

```
payload = (f"{intent}SN{str(seq).zfill(7)}TXN{trans_id}LAST{Z}{dp[charsent:dp_size]}")
```

Figure 9: Packet creation for last packet (rem = 0)

- The remaining data to be sent is not zero, but is less than the packet size (shown in line 131)
 - We still set the last packet (Z) flag to 1. This time, we get payload from the last character sent up to the remaining data in the data packet.

```
payload = (f"{intent}SN{str(seq).zfill(7)}TXN{trans_id}LAST{Z}{dp[charsent:charsent+rem]}")
```

Figure 10: Packet creation for last packet (rem < packet size)

```

108 #start sending
109 while charsent < dp_size:
110     #last packet rules
111     #if packet size and current size of data sent exceeds remaining payload size
112     if (charsent + packet_size >= dp_size):
113         #to get remaining data left, subtract payload size from sent characters so far
114         rem = dp_size - charsent
115         #if 0 is the answer, then there are no more data left to send
116         if rem == 0:
117             #Tell server this is the last packet, set Z to 1
118             Z = 1
119
120             """
121             Create payload using specifications:
122             ID = 44919a94 (given as project credential)
123             SN = sequence number padded with zeroes using .zfill
124             TXN = transaction id taken from server
125             PAYLOAD = take data from payload starting from the last sent character
126             to the length of the data packet size
127             """
128             payload = (f"{intent}SN{str(seq).zfill(7)}TXN{trans_id}LAST{Z}{dp[charsent:dp_size]}")
129
130         #remaining data may not be equal to 0 but is still less than packet size
131         elif rem > 0 and rem < packet_size:
132             #this is still the last packet to be sent, set Z to 1
133             Z = 1
134             #take payload to be from the last character sent up to the remaining data
135             payload = (f"{intent}SN{str(seq).zfill(7)}TXN{trans_id}LAST{Z}{dp[charsent:charsent+rem]}")
136         #if packet isn't last, packet is for transmission
137         #take payload to be from last character sent up to the size of packet
138         else:
139             payload = (f"{intent}SN{str(seq).zfill(7)}TXN{trans_id}LAST{Z}{dp[charsent:charsent+packet_size]}")
140

```

Figure 11: sender.py; 109:139

If the packet to be sent is not the last packet in the transmission, then we just follow the standard creation of packets discussed in Section 3.7.2.

3.7.4 Main Loop: Packet sending (Lines 145-149)

After creating the packet to be sent, we can now send them to the server. Figure 12 shows the code snippet responsible for this.

```

141     #print payload for checking/debugging
142     #print("Current payload", payload)
143
144     #use provided checksum function to generate checksum (client side)
145     checksum = compute_checksum(payload)
146
147     #send payload to server
148     clientSock.sendto(payload.encode(), UDP_IP_PORT)
149     print("Packet sending...")
150

```

Figure 12: Sending packet to server (sender.py; 141:149)

Line 145 computes for the checksum of the packet to be sent for error checking purposes. The helped function used for this is the `compute_checksum` function provided in the project specifications.

Afterwards, we send the packet to the server using the `sendto()` method shown in line 148. Again, this method takes in two arguments – the data to be sent, and where to send it to. In this case, the data to be sent is the packet (variable `payload`). But first, we must encode it into byte string format. The address and port of the server are again initialized in the `UDP_IP_PORT` variable.

3.7.5 ACK message

The server acknowledges the packet if it was sent successfully and will send an ACK packet back to the client in the following format:

A	C	K	X	X	X	X	X	X	X
0	1	2	3	4	5	6	7	8	9
T	X	N	Y	Y	Y	Y	Y	Y	Y
10	11	12	13	14	15	16	17	18	19
M	D	5	CHECKSUM						
20	21	22							

Table 4: ACK packet format

Where:

- ACKXXXXXXXX denotes the sequence number of the packet acknowledged prefixed by “ACK”
- TXNYYYYYYY denotes the transaction ID prefixed by “TXN”
- MD5CHECKSUM denotes the checksum of the packet from the receiver/server side.

This is the expected ACK message that will be received from the server.

3.7.6 Getting the ACK

Since we know what the ACK message from the server, there are two possible scenarios. First, if the server was able to send an ACK message, and if the server did not send an ACK message. We use a try-except block for this, since the socket will time out (i.e. the `settimeout()` value initialized) if it hasn’t received a response from the server. If the socket times out, it will throw an error. Thus, we use a try-except block to catch the timeout error to be thrown.

- If the server was able to ACK the packet that we sent, then we can either:
 - Continue sending packets AND increment packet size; OR
 - Continue sending packets ONLY (this means that packet size has been maxed)
- If the server was NOT able to send an ACK, then we can either:
 - Decrement packet size AND retry sending packet; OR
 - Retry sending packet ONLY.

The implementation of the first scenario is shown in Figure 13. For the try block of the code, this means that if the socket has not reached the specified timeout value yet, then we can get a response from the server. This is done line 158 using the `recvfrom()` method, taking in 1024 bytes of data from the server. The response of the server is then initialized into the `response` variable. Once a response has been received, then we know that the server has accepted our initial payload size. And so, we set the `accepted` flag to 1 in line 160. After this, we update the

sent variable to hold the sent data so far in line 162. We decode the server's response in line 168, and then get the checksum part (refer to Table 4) of the response in line 169.

We then check if the checksums of the client-sent packet and server-packet matches. If they do, then we know that the right packet has been sent and received. If they don't match, we break out of the loop since this means that there has been an error in packet transmission. These are shown in lines 172 to 176.

In line 179, we increment the `charsent` variable with the packet size. If the packet has been accepted, then the last character sent will also change depending on how big the packet was sent. We also increment the sequence number of the packet in line 181.

Lines 185 to 194 constitute the incrementation of the packet size. Remember that the increment flag tells us if the packet can be incremented or not. If this flag is 0, then this means that previous packets that has been sent were not rejected by the server, and next packet can be incremented. The formula follows that of the initial packet sizing in Section 3.7.1 but with a smaller percentage.

```
packet_size += int(ceil(0.02*(dp_size-packet_size)))
```

We increment our packet size only by a fraction of the data size when subtracted by the size of the current packet. We use the `ceil` function of Python's `math` module to round this value up and convert it into an integer. Again, this value was taken from a series of trial-and-error executions. Percentages ranging from 2 – 10% of the remaining data size were compared and the percentage that led to successful runs and manageable fluctuations of packet sizes was determined to be 2%. Hence, this value was used in our implementation.

If the server has already accepted packets, but the increment flag is set to 1, then we know that the payload size has been maxed out. And so, we set the maxed flag to 1 in line 201.

For the second scenario where the socket times out and the server fails to send an ACK reply to the packet sent, then we know that our packet has an incorrect payload size, and we need to decrement the current payload size that we have. Figure 14 shows the except block which handles this scenario.

Lines 214-215 checks if the maxed flag is set to 1. If it is, then we skip the iteration (i.e. just send the packet to the server). Line 219-220 checks if the accepted flag is set to 0. If it is, then we know that the initial packet has not been accepted by the server, and so we decrement the initial packet size by 2% of the data packet size subtracted from the packet size. Again, note that this value was taken from trial-and-error executions, testing different percentages, and choosing 2% as the most viable increment.

Lines 228-231 checks if the packet has been accepted. If the accepted flag is set to 1, then we know that previous packets has been accepted by the server already. However, the server may have not sent an ACK reply because the incrementation of the packet size was too big. So, we revert back to the previous packet size.

Finally, lines 234-235 checks if the transmission time has exceeded 120 seconds, if it has, packet transmission must be terminated and while loop must stop executing.

Lines 237-238 check if the payload that was sent matches the original payload for debugging and error checking purposes.

```
156 try:
157     #try getting ACK response from server
158     response, addr = clientSock.recvfrom(1024)
159     #if response was received, server was able to receive packet, set 'accepted' flag to 1
160     accepted = 1
161     #increment 'sent' variable to hold current sent data (for checking)
162     sent += dp[charsent:charsent+packet_size]
163
164     print("ACK received...")
165
166     #decode response from server to get checksum; checksum at index 23 onwards of ACK packet
167     #according to project specs
168     rescheck = response.decode()
169     rescheck = rescheck[23:]
170
171     #if checksums are not equal, break transmission
172     if rescheck != checksum:
173         print(f"Server checksum: {rescheck}")
174         print(f"Client checksum: {checksum}")
175         print("Checksum error")
176         break
177
178     #increment the last sent character to be packet size
179     charsent += packet_size
180     #increment sequence number of packet by 1
181     seq += 1
182
183     #increment flag used to check if payload can still be increased
184     #if this flag is 0, incrementation can still be done
185     if increment == 0:
186         """
187         increment packet size by subtracting current packet size from total
188         data packet size
189
190         get only 2 percent of size to add
191         this value was gotten from trial and error. implementation first tested by
192         getting 50% -> 25% -> 10% -> 5% -> 2%
193         """
194         packet_size += int(ceil(0.02*(dp_size-charsent)))
195
196     #accepted flags and increments flags will be set to 1 if packet was:
197     # 1) accepted by server
198     # 2) but can no longer be incremented
199     #this means that payload size has been maxed out and we set 'maxed' flag to 1
200     if accepted == 1 and increment == 1:
201         maxed = 1
202
203     print(f"Current packet size {packet_size}, remaining {dp_size-charsent}")
```

Figure 13: Server sends an ACK message (sender.py; 158:203)

```

210 except socket.timeout:
211     print("Server NACKed")
212     print(f"Packet not accepted because of payload size: {packet_size}")
213
214     if maxed == 1:
215         continue
216
217     #if accepted flag has not been set to 1, initial packet sent was not accepted, adjust payload size
218     #values same as used during incrementation of initial packet
219     if accepted == 0:
220         packet_size -= int(ceil(0.02*(dp_size-packet_size)))
221
222     #if accepted flag has been set to 1, transmission has already been accepted by server
223     #but socket has timed out either because of:
224     #1) payload size
225     #2) being in the queue
226     #in this case, we want to decrease payload size still and send again
227     #in general, increment bigger, decrement smaller
228     elif accepted == 1:
229         #set increment flag to 1 to signify that we can't increment payload size anymore
230         increment = 1
231         packet_size -= int(ceil(0.003*(dp_size-packet_size)))
232
233     #checker if 120 second time allotment has passes
234     if time.time() - startTime > 120:
235         break
236
237 print("Packet transmission finished.")
238 print(f"Sent data equal to payload? {sent == dp}")

```

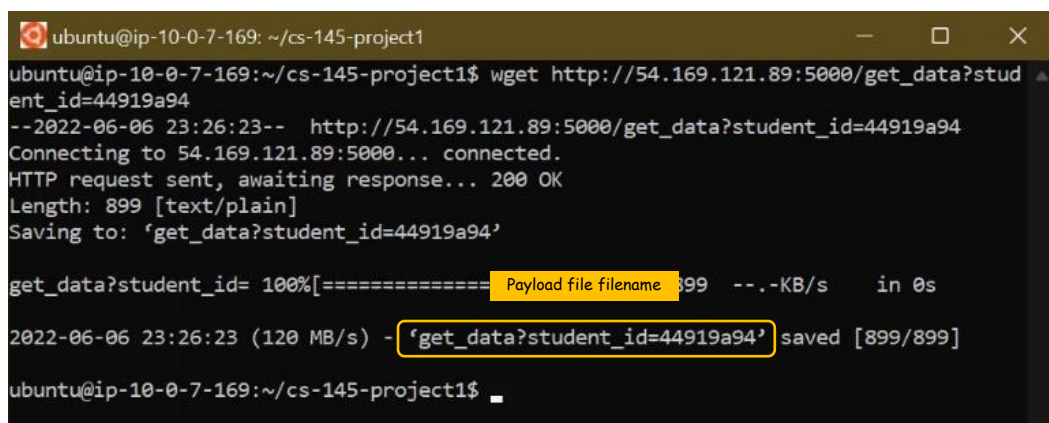
Figure 14: Server does not an ACK message (sender.py; 210:238)

4 Testing

To test the implementation of the sender program, five transactions were initiated.

4.1 Transaction 1

For the first transaction, we first download the following payload file:



```

ubuntu@ip-10-0-7-169: ~/cs-145-project1
ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?stud
ent_id=44919a94
--2022-06-06 23:26:23-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 899 [text/plain]
Saving to: 'get_data?student_id=44919a94'

get_data?student_id= 100%[===== Payload file filename 899 --.-KB/s in 0s

2022-06-06 23:26:23 (120 MB/s) - 'get_data?student_id=44919a94' saved [899/899]

ubuntu@ip-10-0-7-169:~/cs-145-project1$

```

Then, we start tshark before initiating a transaction with the server.


```

ubuntu@ip-10-0-7-169: ~/cs-145-project1
ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?student_id=44919a94
--2022-06-06 23:26:23-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 899 [text/plain]
Saving to: 'get_data?student_id=44919a94'

get_data?student_id= 100%[=====] 899 --.-KB/s in 0s

2022-06-06 23:26:23 (120 MB/s) - 'get_data?student_id=44919a94' saved [899/899]

ubuntu@ip-10-0-7-169:~/cs-145-project1$

Packet capture started

ubuntu@ip-10-0-7-169:~/cs-145-project1$ sudo tshark -i eth0 -w /home/tracefile.pcapng
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
** (tshark:1753) 23:26:26.701983 [Main MESSAGE] -- Capture started.
** (tshark:1753) 23:26:26.702387 [Main MESSAGE] -- File: "/home/tracefile.pcapng"

```

Once tshark has started capturing packets, we can now run our sender program via the command "python3 sender.py -f get_data?student_id=44919a94 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94."

cs145_212

54.169.121

Instances

+

Not secure

54.169.121.89

...

Transactions

Show

Search:

Transaction ID	Time Started	Duration	Result	Frequency Used
0001319	2022-06-06 23:26:31.474865	None	None	1
Transaction ID: 0001319	2022-06-06 23:23:40.580382	120.000	Failed to send data	1
0001317	2022-06-06 23:12:19.956418		Transaction alive	Frequency used of payload: 1
0001316	2022-06-06 23:09:30.935843	114.195	Successfully sent data	1
0001315	2022-06-06 23:06:48.580690	115.185	Successfully sent data	1
0001314	2022-06-06 23:04:15.901270	115.264	Successfully sent data	1
0001313	2022-06-06 23:01:45.647258	114.680	Successfully sent data	1

```

ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?student_id=44919a94
--2022-06-06 23:26:23-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 899 [text/plain]
Saving to: 'get_data?student_id=44919a94'

get_data?student_id= 100%[=====] 899 --.-KB/s in 0s

2022-06-06 23:26:23 (120 MB/s) - 'get_data?student_id=44919a94' saved [899/899]

ubuntu@ip-10-0-7-169:~/cs-145-project1$ python3 sender.py -f get_data?student_id=44919a94 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94
Trans ID: 0001319
Length: 899
Current payload ID44919a94SN000000TXN0001319LAST0JymFPAbbqIdQhpMbIkKwKViCVFpt
Packet sending...

ubuntu@ip-10-0-7-169:~/cs-145-project1$ sudo tshark -i eth0 -w /home/tracefile.pcapng
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
** (tshark:1753) 23:26:26.701983 [Main MESSAGE] -- Capture started.
** (tshark:1753) 23:26:26.702387 [Main MESSAGE] -- File: "/home/tracefile.pcapng"
92

```

Note that this transaction has a transaction ID of 0001319. After successfully sending the packets, the sender program finishes executing and the result in the TGRS is updated and is shown to have **successfully sent data**.

cs145_212

54.169.121

Instances

Not secure

54.169.121.8...

Transactions

Show entries

Search:

Transaction ID	Time Started	Duration	Result	Frequency Used
0001319	2022-06-06 23:26:31.474865	112.654	Successfully sent data	1
0001317	2022-06-06 23:12:19.95641	120.000	Failed to send data	1
0001316	2022-06-06 23:09:30.935843		Successfully sent data	1
0001315	2022-06-06 23:06:48.580690	115.185	Successfully sent data	1
0001314	2022-06-06 23:04:15.901270	115.264	Successfully sent data	1
	2022-06-06 23:01:45.647258	114.680	Successfully sent data	1

Transaction ID: 0001319

Time started: 23:26:31

Transaction result: Successfully sent data

Frequency used of payload: 1

Transaction time stamp matching in trace file and TGRS

Packet sending...

ACK received...

Current packet size 11, rem63

Current payload ID44919a94SN0000076TXN0001319LAST0jZyrmMVOiE

Packet sending...

ACK received...

Current packet size 11, rem52

Current payload ID44919a94SN0000077TXN0001319LAST0YdKLVLGVnn

Packet sending...

ACK received...

Current packet size 11, rem41

Current payload ID44919a94SN0000078TXN0001319LAST0dtKGIstQDe

Packet sending...

ACK received...

Current packet size 11, rem30

Current payload ID44919a94SN0000079TXN0001319LAST0ZfpDMjnkCaX

Packet sending...

ACK received...

Current packet size 11, rem19

Current payload ID44919a94SN0000080TXN0001319LAST1NbMtEobd

Packet sending...

ACK received...

Current packet size 11, rem8

Current payload ID44919a94SN0000081TXN0001319LAST1NbMtEobd

Packet sending...

ACK received...

Current packet size 11, rem-3

Sent data equal to payload? True

ubuntu@ip-10-0-7-169:~/cs-145-project1\$

Does original and sent payload match? Yes

Execution of sender finishes

Running as user "root" and group "root". This could be dangerous.

Capturing on 'eth0'

** (tshark:1753) 23:26:26.701983 [Main MESSAGE] -- Capture started.

** (tshark:1753) 23:26:26.702387 [Main MESSAGE] -- File: "/home/tracefile.pcapng"

891

[0] 0: bash*

"ip-10-0-7-169" 23:28 06-Jun-22

By checking the transaction in the tracefile generated, we can verify that the intent message was indeed sent as packet 53 in the tracefile by filtering out the UDP packets and checking their payload. The server's accept message can also be verified as packet 54 which contains the transaction ID.

No.	Time	Source	Destination	Protocol	Length	Info
13	23:26:27.298053947	10.0.7.169	169.254.169.123	NTP	90	NTP Version 4, client
14	23:26:27.298504077	169.254.169.123	10.0.7.169	NTP	90	NTP Version 4, server
53	23:26:31.469855136	10.0.7.169	10.0.1.175	UDP	52	6745 → 9000 Len=10 [UDP CHECKSUM INCORRECT]
54	23:26:31.488600422	10.0.1.175	10.0.7.169	UDP	49	9000 → 6745 Len=7

Packet 53:

Intent message

52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface eth0, II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)

Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175

User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745

Destination Port: 9000

Length: 18

Checksum: 0x0000 (student assigned port)

Stream index: 1

[Timestamps]

UDP payload (10 bytes)

Data (10 bytes)

Data: 49443434393139613934

[Length: 10]

Destination port: 9000 (server assigned port)

Intent message containing student ID

53

23:26:31.469855136

10.0.7.169

10.0.1.175

UDP

52

6745 → 9000 Len=10 [UDP CHECKSUM INCORRECT]

0000

06 0e b4 c8 23 6e 06 37 3c 93 56 96 08 00 45 00

0010

00 26 16 c3 40 00 40 11 06 ad 0a 00 07 a9 0a 00

0020

01 af 1a 59 23 28 00 12 1d 7b 49 44 34 34 39 31

0030

39 61 39 34

9a94

ID4491

54

23:26:31.488600422

10.0.1.175

10.0.7.169

UDP

49

9000 → 6745 Len=7

0000

06 37 3c 93 56 96 06 0e b4 c8 23 6e 08 00 45 00

0010

00 23 21 d5 40 00 40 11 fb 9d 0a 00 01 af 0a 00

0020

07 a9 23 28 1a 59 00 0f d8 64 30 30 30 31 33 31

0030

39

9

d000131

Packet 54:

Accept message

49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface eth0, II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)

Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169

User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000

Destination Port: 6745

Length: 15

Checksum: 0xd864 (server assigned port)

Stream index: 1

[Timestamps]

UDP payload (7 bytes)

Data (7 bytes)

Data: 30303031333139

[Length: 7]

Source port: 6745 (student assigned port)

Server's accept message containing transaction ID

We can also verify that the last packet sent has the following payload:

```

ACK received...
Current packet size 11, rem19
Current payload ID44919a94SN0000080TXN0001319LAST0PT1LdMwKXGs
Packet sending...
ACK received...
Current packet size 11, rem8
Current payload ID44919a94SN0000081TXN0001319LAST1NbMtEobd
Packet sending...
ACK received...
Current packet size 11, rem-3
Sent data equal to payload? True
  
```

Z=1

**Transaction last payload
Z = 1**

If we analyze the tracefile, the last packet sent for this transaction from my machine to the server was packet number 857. We can verify the payload of the packet and check that it indeed corresponds to the payload printed by the sender program.

Packet 857: Last payload sent

857 23:28:23.122145338 10.0.7.169 10.0.1.175 UDP 84 6745 → 9000 Len=42 [UDP CHECKSUM INCORRECT]

84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface eth0

Ethernet II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)

Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175

User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745
Destination Port: 9000
Length: 50

Checksum: [Checksum offload]
[Checksum offload]
[Stream offload]
[Timestamps]
UDP payload (42 bytes)

Data (42 bytes)
Data: 49443434393139613934534e303030303030383154584e303030313331394c415354314e62...
[Length: 42]

0000 06 0e b4 c8 23 6e 06 37 3c 93 56 96 08 00 45 00 ...#n:7 <V...E
0010 00 46 4e 1d 40 00 40 11 cf 32 0a 00 07 a9 0a 00 ...FN:@:2.....
0020 01 af 1a 59 23 28 00 32 1d 9b 49 44 34 34 39 31 ...Y#(-2 ID4491
0030 39 61 39 34 53 4e 30 30 30 30 38 31 54 58 4e 9a94SN00 00081TXN
0040 30 30 30 31 33 31 39 4c 41 53 54 31 4e 62 4d 74 0001319L AST1NbMt
0050 45 6f 62 64 Eobd

Source port: 6745 (student assigned port)

Destination port: 9000 (server assigned port)

**Last payload sent
SQN: 0000081
TXN: 001319**

Correspondingly, the server has also sent the last ACK message for this payload in packet 864.

Packet 864: Last ACK sent

864 23:28:24.129315875 10.0.1.175 10.0.7.169 UDP 97 9000 → 6745 Len=55

97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface eth0

Ethernet II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)

Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169

User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000
Destination Port: 6745
Length: 63

Checksum: [Checksum offload]
[Checksum offload]
[Stream offload]
[Timestamps]
UDP payload (55 bytes)

Data (55 bytes)
Data: 41434b3030303030383154584e303030313331394d443563626265383562356639313262...
[Length: 55]

0000 06 37 3c 93 56 96 06 0e b4 c8 23 6e 08 00 45 00 ...7<V...#n:E
0010 00 53 5b 4e 40 00 40 11 c1 f4 0a 00 01 af 0a 00 ...S[N@:.....
0020 07 a9 23 28 1a 59 00 3f 91 90 41 43 4b 30 30 30 ...#(\Y? ACK000
0030 30 30 38 31 54 58 4e 30 30 30 31 33 31 39 4d 41 0081TXN0 001319MD
0040 35 63 62 62 65 38 35 62 35 66 39 31 32 62 37 32 5cbbe85b 5f912b72
0050 62 66 64 36 36 39 37 37 32 61 30 39 36 32 32 32 bfd66977 2a096222
0060 62 b

Source port: 9000 (server assigned port)

Destination port: 6745 (student assigned port)

**Last ACK sent from server to client
SQN: 0000081
TXN: 001319**

Thus, it can be shown that the sender implementation is indeed successful in sending data from the client to the server in this transaction.

4.2 Transaction 2

We do the same thing for another transaction and start by downloading a new payload file.

```
ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?student_id=44919a94
--2022-06-06 23:36:11-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 628 [text/plain]
Saving to: 'get_data?student_id=44919a94.1'

get_data?student_id= 100%[=====] 628 --.-KB/s in 0s
2022-06-06 23:36:11 (86.2 MB/s) - 'get_data?student_id=44919a94.1' saved [628/628]
```

Start tshark and use the new payload filename to execute the sender program via command line. We use the command "python3 sender.py -f get_data?student_id=44919a94.1 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94"

The screenshot displays the TGRS Transactions table and a terminal window. The table lists transactions with columns: Transaction ID, Time Started, Duration, Result, and Frequency Used. Transaction 0001320 is highlighted, showing it was successfully sent. The terminal shows the command to run the sender program and the tshark command to capture packets.

Transaction ID	Time Started	Duration	Result	Frequency Used
0001320	2022-06-06 23:36:37.021838	None	None	1
0001318	2022-06-06 23:23:40.580382	112.654	Successfully sent data	1
0001317	2022-06-06 23:12:19.956418	111.322	Successfully sent data	1
0001316	2022-06-06 23:09:30.935843	114.195	Successfully sent data	1
0001315	2022-06-06 23:06:48.580690	115.185	Successfully sent data	1
0001314	2022-06-06 23:04:15.901270	115.264	Successfully sent data	1

Transaction ID: 0001320

Transaction alive

Frequency used of payload: 1

Command runs on terminal

Tshark capturing packets

Note that this transaction has a transaction ID of 0001320. After successfully sending the packets, the sender program finishes executing and the result in the TGRS is updated and is shown to have **successfully sent data**.

cs145_212

54.169.121

Instances

+

Not secure

54.169.121.8...

54.169.121.8...

54.169.121.8...

54.169.121.8...

Transactions

Show

entries

Search:

Transaction ID	Time Started	Duration	Result	Frequency Used
0001320	2022-06-06 23:36:37.021838	119.221	Successfully sent data	1
0001318	2022-06-06 23:23:40.580382	112.654	Failed to send data	1
0001317	2022-06-06 23:12:19.956418		Successfully sent data	1
0001316	2022-06-06 23:09:30.935843	114.195	Successfully sent data	1
0001315	2022-06-06 23:06:48.580690	115.185	Successfully sent data	1
	2022-06-06 23:04:15.901270	115.264	Successfully sent data	1

Transaction ID: 0001320

Time started: 23:36:37

Transaction result: Successfully sent data

Frequency used of payload: 1

Does original and sent payload match? Yes

Transaction time stamp matching in trace file and TGRS

Packet sending...

ACK received...

Current packet size 57, rem245

Current payload ID44919a94SN0000008TXN0001320LAST0ubGwebZxmArifPsAYKASvBdqdnNuPYGGowpfCGwRL1ZNET1AibPbCkgEZK

Packet sending...

ACK received...

Current packet size 57, rem188

Current payload ID44919a94SN0000009TXN0001320LAST0fyXRhNnyUcxTyUnIFPIYnbUntuwiHguEdeXyShtMDMgJcKTTdSIzaWIC

Packet sending...

ACK received...

Current packet size 57, rem131

Current payload ID44919a94SN0000010TXN0001320LAST0gNncudGndQcKsexukzRtpmdigJgHmPbqoYpUVKYCxSiwZhmBcteSuFXmh

Packet sending...

ACK received...

Current packet size 57, rem74

Current payload ID44919a94SN0000011TXN0001320LAST0YCXhzWqTAKbjEodmFLxxSszrwhQHgpenZqcjeiQpiDAOdHLaGgGutnpvD

Packet sending...

ACK received...

Current packet size 57, rem17

Current payload ID44919a94SN0000012TXN0001320LAST1NYRuHkgqZUuK1TfJL

Packet sending...

ACK received...

Current packet size 57, rem40

Sent data equal to payload? True

ubuntu@ip-10-0-7-169:~/cs-145-project1\$

To https://github.com/camillerule/cs-145-project1

ubuntu@ip-10-0-7-169:~/cs-145-project1\$ sudo tshark -i eth0 -w /home/tracefile1.pcapng

Running as user "root" and group "root". This could be dangerous.

Capturing on 'eth0'

** (tshark:1822) 23:36:30.846712 [Main MESSAGE] -- Capture started.

** (tshark:1822) 23:36:30.847069 [Main MESSAGE] -- File: "/home/tracefile1.pcapng"

1133

ip-10-0-7-169 23:38 06-Jun-22

By checking the transaction in the tracefile generated, we can verify that the intent message was indeed sent as packet 171 in the tracefile by filtering out the UDP packets and checking their payload. The server's accept message can also be verified as packet 173 which contains the transaction ID.

No.

Time

Source

Destination

Protocol

Length

Info

171	23:36:37.017858351	10.0.7.169	10.0.1.175	UDP	52	6745 → 9000 Len=10 [UDP CHECKSUM INCORRECT]
173	23:36:37.035980907	10.0.1.175	10.0.7.169	UDP	49	9000 → 6745 Len=7
180	23:36:37.036800041	10.0.7.169	10.0.1.175	UDP	96	6745 → 9000 Len=54 [UDP CHECKSUM INCORRECT]

Frame 171: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface eth0

Ethernet II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)

User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745

Destination Port: 9000

Length: 18

Checksum: 0 (student assigned port)

Destination port: 9000 (server assigned port)

Intent message containing student ID

Frame 173: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface eth0

Ethernet II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)

User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000

Destination Port: 6745

Length: 15

Checksum: 0 (server assigned port)

Destination port: 6745 (student assigned port)

Server's accept message containing transaction ID TXN: 0001320

We can also verify that the last packet sent has the following payload:

```

ACK received...
Current packet size 57, rem17
Current payload ID44919a94SN0000012TXN0001320LAST1NYRuHkgqZUuK1TfjL
Packet sending...
ACK received...
Current packet size 57, rem-40
Sent data equal to payload? True
ubuntu@ip-10-0-7-169:~/cs-145-project1$

```

Z=1

Transaction last payload
Z = 1

If we analyze the tracefile, the last packet sent for this transaction from my machine to the server was packet number 1055. We can verify the payload of the packet and check that it indeed corresponds to the payload printed by the sender program.

1055	23:38:28.228507276	10.0.7.169	10.0.1.175	UDP	93 6745 → 9000 Len=51 [UDP CHECKSUM INCORRECT]
1086	23:38:35.232737373	10.0.7.169	169.254.169.123	NTP	90 NTP Version 4, client

Frame 1055: 93 bytes on wire (744 bits), 93 bytes captured (744 bits) on interface eth0

Ethernet II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)

Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175

User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Packet 1055: Last payload sent

Destination Port: 9000

Length: 59

Checksum: 0x1d41 (incorrect, should be 0x1d41)

[Checksum Status: [Checksum Status: [Stream index: 0]

Source port: 6745 (student assigned port)

Destination port: 9000 (server assigned port)

UDP checksum offload

0000 06 0e b4 c8 23 6e 06 37 3c 93 56 96 08 00 45 00 ...#n·7 <·V···E·

0010 00 4f 93 bf 40 00 40 11 89 87 0a 00 07 a9 0a 00 ...O··@·@· ······

0020 01 af 1a 59 23 28 00 3b 1d a4 49 44 34 34 39 31 ...·Y#(·;· ·ID4491

0030 39 61 39 34 53 4e 30 30 30 30 31 32 54 58 4e ...9a94SN00 00012TXN

0040 30 30 30 31 33 32 30 4c 41 53 54 31 4e 59 52 75 ...0001320L AST1NYRu

0050 48 6b 67 71 5a 55 75 4b 6c 54 66 6a 4c ...HkgqZUuK 1TfjL

Last payload sent
SEQ: 0000012
TXN: 001320

1092	23:38:36.242710485	10.0.1.175	10.0.7.169	UDP	97 9000 → 6745 Len=55
------	--------------------	------------	------------	-----	-----------------------

Frame 1092: 97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface eth0

Ethernet II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)

Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169

User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Packet 1092: Last ACK message

Source Port: 9000

Destination Port: 6745

Length: 55

Checksum: 0x1d41 (incorrect, should be 0x1d41)

[Checksum Status: [Checksum Status: [Stream index: 0]

Source port: 9000 (server assigned port)

Destination port: 6745 (student assigned port)

UDP packet

0000 06 37 3c 93 56 96 06 0e b4 c8 23 6e 08 00 45 00 ...<·V··· ·#n·E·

0010 00 53 6c 4b 40 00 40 11 b0 f7 0a 00 01 af 0a 00 ...·SlK@·@· ······

0020 07 a9 23 28 1a 59 00 3f f4 5d 41 43 4b 30 30 30 ...·#(·Y·? ·]ACK000

0030 30 30 31 32 54 58 4e 30 30 30 31 33 32 30 4d 31 ...0012TXN0 001320MD

0040 35 65 38 63 34 66 39 36 34 65 32 38 35 33 38 32 ...5e8c4f96 4e285382

0050 33 37 61 61 62 61 38 66 31 39 30 36 38 30 30 30 ...8f 190688ec

0060 31

Server's last ACK
for:
SEQ: 0000012
TXN: 001320

The server also sends its last ACK message for the corresponding packet shown as packet number 1092 in the trace file. Thus, it can be shown that the sender implementation is indeed successful in sending data from the client to the server in this transaction as well.

4.3 Transaction 3

We do the same thing for another transaction and start by downloading a new payload file.

```
ubuntu@ip-10-0-7-169: ~/cs-145-project1
ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?student_id=44919a94
--2022-06-06 23:38:55-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 583 [text/plain]
Saving to: 'get_data?student_id=44919a94.2'

get_data?student_id= 100%[=====] Payload file filename for transaction 3 --.-KB/s in 0s
2022-06-06 23:38:55 (72.5 MB/s) 'get_data?student_id=44919a94.2' saved [583/583]
```

Start tshark and use the new payload filename to execute the sender program via command line. We use the command "python3 sender.py -f get_data?student_id=44919a94.2 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94"

Transactions

Show entries

Search:

Transaction ID	Time Started	Duration	Result	Frequency Used
0001321	2022-06-06 23:39:20.983634	None	None	1
0001319	2022-06-06 23:26:31.474865	112.654	Successful sent data	1
0001318	2022-06-06 23:23:40.580382	120.000	Failed to send data	1
0001317	2022-06-06 23:12:19.956418	111.322	Successfully sent data	1
0001316	2022-06-06 23:09:30.935843	114.195	Successfully sent data	1
0001315	2022-06-06 23:09:30.935843	115.185	Successfully sent data	1

```
ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?student_id=44919a94
--2022-06-06 23:38:55-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 583 [text/plain]
Saving to: 'get_data?student_id=44919a94.2'

get_data?student_id= 100%[=====] 583 --.-KB/s in 0s
2022-06-06 23:38:55 (72.5 MB/s) - 'get_data?student_id=44919a94.2' saved [583/583]

Command runs on terminal

ubuntu@ip-10-0-7-169:~/cs-145-project1$ python3 sender.py -f get_data?student_id=44919a94.2 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94
Trans ID: 0001321
Length: 583
Current payload ID44919a94SN000000TXN0001321LAST0ZDDLOVbjTOEIAHKvJTz
Packet sending...

Transaction ID: 0001321

ubuntu@ip-10-0-7-169:~/cs-145-project1$ sudo tshark -i eth0 -w /home/tracefile2.pcapng
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
** (tshark:1881) 23:39:14.360396 [Main MESSAGE] -- Capture started.
** (tshark:1881) 23:39:14.360826 [Main MESSAGE] -- File: "/home/tracefile2.pcapng"
184

Tshark capturing packets
```

Note that this transaction has a transaction ID of 0001321. After successfully sending the packets, the sender program finishes executing and the result in the TGRS is updated and is shown to have **successfully sent data**.

Transactions

Show entries

Search:

Transaction ID	Time Started	Duration	Result	Frequency Used
0001321	2022-06-06 23:39:20.983634	114.196	Successfully sent data	1
0001319	2022-06-06 23:39:20.983634	119.221	Successfully sent data	1
0001318	2022-06-06 23:23:40.580382	119.221	Failed to send data	1
0001317	2022-06-06 23:12:19.956418	111.322	Successfully sent data	1
0001316	2022-06-06 23:09:30.935843	114.195	Successfully sent data	1
0001315	2022-06-06 23:06:48.580690	115.185	Successfully sent data	1

```

ACK received...
Current packet size 65, rem306
Current payload ID44919a94SN0000007TXN0001321LAST0TskwmdynioSWFRxikKdLCLVbFVPPMvRsr
b6jWgAYNQYLKcHcHHYYINGlY1AcUJXA
Packet sending...
ACK received...
Current packet size 65, rem241
Current payload ID44919a94SN0000007TXN0001321LAST0iaJLMDJVuG1QkQAIYQsoGGPZuCchVcTvRN
WbqYjPDLEmUuWtTudZgZcCSKvCNADkBN
Packet sending...
ACK received...
Current packet size 65, rem176
Current payload ID44919a94SN0000008TXN0001321LAST0uFeqUYMxvddonLSyDZpWQwqAOYyhVwUKFb
MzwzZVxPakfoENXOFicZegSsRBHbSac
Packet sending...
ACK received...
Current packet size 65, rem111
Current payload ID44919a94SN0000009TXN0001321LAST0zZIHUMcWFHEzMCOKeOXufpWanraOhiUUEcc
KjYUOcpGkxxnFeUWmtCuJJSGMnYpBwm
Packet sending...
ACK received...
Current packet size 65, rem46
Current payload ID44919a94SN0000010TXN0001321LAST1JvVxeDRIsNqZ2P1erFOHYEXioxmJj3kmJ
ezbnaUTFNkLG
Packet sending...
ACK received...
Current packet size 65, rem-19
Sent data equal to payload? True
ubuntu@ip-10-0-7-169:~/cs-145-project1$
ubuntu@ip-10-0-7-169:~/cs-145-project1$ sudo tshark -i eth0 -w /home/tracefile2.pcapng
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
** (tshark:1881) 23:39:14.360396 [Main MESSAGE] -- Capture started.
** (tshark:1881) 23:39:14.360826 [Main MESSAGE] -- File: "/home/tracefile2.pcapng"
849
[0] sudo: "ip-10-0-7-169" 23:41:06-Jun-22

```

Transaction ID: 0001321
Time started: 23:39:20

Transaction result: Successfully sent data

Frequency used of payload: 1

Does original and sent payload match? Yes

Execution of sender finishes

Transaction time stamp matching in trace file and TGRS

By checking the transaction in the tracefile generated, we can verify that the intent message was indeed sent as packet 150 in the tracefile by filtering out the UDP packets and checking their payload. The server's accept message can also be verified as packet 151 which contains the transaction ID.

No.	Time	Source	Destination	Protocol	Length	Info
150	23:39:20.979619654	10.0.7.169	10.0.1.175	UDP	52	6745 → 9000 Len=10 [UDP CHECKSUM INCORRECT]
151	23:39:20.999826362	10.0.1.175	10.0.7.169	UDP	49	9000 → 6745 Len=7

Packet 150: Intent message

Frame 150: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface eth0

Ethernet II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)

Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175

User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745
Destination Port: 9000
Length: 18

Source port: 6745 (student assigned port)

Destination port: 9000 (server assigned port)

Intent message containing student ID

Packet 151: Accept message

Frame 151: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface eth0

Ethernet II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)

Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169

User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000
Destination Port: 6745
Length: 15

Checksum: 0xe063 [correct]

Source port: 9000 (server assigned port)

Destination port: 6745 (student assigned port)

Server's accept message containing transaction ID TXN: 0001321

We can also verify that the last packet sent has the following payload:

```

ACK received...
Current packet size 65, rem46
Current payload ID44919a94SN0000010TXN0001321LAST1JvVxeDRlsNqZTPZlerFOHYEXioxmJjJkmJ
ezbnauTFNkLG
Packet sending...
ACK received...
Current packet size 65, rem-19
  
```

Transaction last payload
Z = 1

If we analyze the tracefile, the last packet sent for this transaction from my machine to the server was packet number 758. We can verify the payload of the packet and check that it indeed corresponds to the payload printed by the sender program.

Packet 758: Last payload sent

Frame 758: 122 bytes on wire (976 bits), 122 bytes captured (976 bits) on interface 0
 Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)
 Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175
 User Datagram Protocol, Src Port: 6745, Dst Port: 9000
 Source Port: 6745
 Destination Port: 9000
 Length: 88
 Destination port: 9000 (server assigned port)
 Source port: 6745 (student assigned port)
 Checksum: 0x9273 (maybe caused by "L")

Packet 799: Last ACK message

Frame 799: 97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface 0
 Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)
 Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169
 User Datagram Protocol, Src Port: 9000, Dst Port: 6745
 Source Port: 9000
 Destination Port: 6745
 Length: 63
 Checksum: 0x34b6 [corrected]
 Source port: 9000 (server assigned port)
 Destination port: 6745 (student assigned port)

Transaction details:

- Transaction last payload: Z = 1
- Last payload sent: SQN: 0000010, TXN: 001321
- Server's last ACK for: SEQ: 0000010, TXN: 001321

The server also sends its last ACK message for the corresponding packet shown as packet number 799 in the trace file. Thus, it can be shown that the sender implementation is indeed successful in sending data from the client to the server in this transaction as well.

4.4 Transaction 4

We do the same thing for another transaction and start by downloading a new payload file.

```
ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?student_id=44919a94
--2022-06-06 23:41:33-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 82 [text/plain]
Saving to: 'get_data?student_id=44919a94.3'

get_data?student_id= 100%[=====] 82 --.-KB/s in 0s
2022-06-06 23:41:33 (10.5 MB/s) - 'get_data?student_id=44919a94.3' saved [82/82]
```

Start tshark and use the new payload filename to execute the sender program via command line. We use the command "python3 sender.py -f get_data?student_id=44919a94.3 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94"

The screenshot displays the TGRS Transactions table and a terminal window. The table lists transactions with columns: Transaction ID, Time Started, Duration, Result, and Frequency Used. Transaction 0001322 is highlighted, showing a successful result and a frequency of 1. The terminal shows the command to run the sender program and the tshark command to capture packets on eth0.

Transaction ID	Time Started	Duration	Result	Frequency Used
0001322	2022-06-06 23:41:55.652875	None	None	1
0001320	2022-06-06 23:36:37.021838	119.221	Successfully sent data	1
0001319	2022-06-06 23:26:31.474865	112.654	Successfully sent data	1
0001318	2022-06-06 23:23:40.580382	120.000	Failed to send data	1
0001317	2022-06-06 23:12:19.956418	111.322	Successfully sent data	1
0001316	2022-06-06 23:09:30.935843	114.195	Successfully sent data	1

Transaction ID: 0001322

Transaction alive

Frequency used of payload: 1

Command runs on terminal

Tshark capturing packets

Note that this transaction has a transaction ID of 0001322. After successfully sending the packets, the sender program finishes executing and the result in the TGRS is updated and is shown to have **successfully sent data**.

Transactions

Show entries

Search:

Transaction ID	Time Started	Duration	Result	Frequency Used
0001322	2022-06-06 23:41:55.652875	115.186	Successfully sent data	1
0001320	2022-06-06 23:36:37.02183	114.196	Successfully sent data	1
0001319	2022-06-06 23:26:31.474865	115.186	Successfully sent data	1
0001318	2022-06-06 23:23:40.580382	120.000	Failed to send data	1
0001317	2022-06-06 23:12:19.956418	111.322	Successfully sent data	1
0001316	2022-06-06 23:09:30.935843	114.195	Successfully sent data	1

Transaction ID: 0001322
Time started: 23:41:55

Transaction result: Successfully sent data

Frequency used of payload: 1

Does original and sent payload match? Yes

Transaction time stamp matching in trace file and TGRS

```

Packet sending...
ACK received...
Current packet size 13, rem47
Current payload ID44919a94SN0000005TXN0001322LAST0CP1jLKsJCEXFA
Packet sending...
Server NACKed
Size not accepted: 13
Current payload ID44919a94SN0000005TXN0001322LAST0CP1jLKsJCEX
Packet sending...
ACK received...
Current packet size 11, rem36
Current payload ID44919a94SN0000006TXN0001322LAST0FAekyxUMhuZ
Packet sending...
ACK received...
Current packet size 11, rem25
Current payload ID44919a94SN0000007TXN0001322LAST0mVDwaTyyxVvk
Packet sending...
ACK received...
Current packet size 11, rem14
Current payload ID44919a94SN0000008TXN0001322LAST0nrFLdHrXwWd
Packet sending...
ACK received...
Current packet size 11, rem3
Current payload ID44919a94SN0000009TXN0001322LAST1abf
Packet sending...
ACK received...
Current packet size 11, rem-8
Send data equal to payload? True
ubuntu@ip-10-0-7-169:~/cs-145-project1$
  
```

Execution of sender finishes

By checking the transaction in the tracefile generated, we can verify that the intent message was indeed sent as packet 27 in the tracefile by filtering out the UDP packets and checking their payload. The server's accept message can also be verified as packet 28 which contains the transaction ID.

Packet 27: Intent message

Frame 27: 52 bytes on wire (416 bits), 52 bytes captured (416 bits)

Ethernet II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)

Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175

User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745
Destination Port: 9000
Length: 18

Source port: 6745 (student assigned port)
Destination port: 9000 (server assigned port)

Intent message containing student ID

Packet 28: Accept message

Frame 28: 49 bytes on wire (392 bits), 49 bytes captured (392 bits)

Ethernet II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)

Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169

User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000
Destination Port: 6745
Length: 15
Checksum: 0xdf63 [correct]
Destination port: 6745 (student assigned port)

Source port: 9000 (server assigned port)

Server's accept message containing transaction ID TXN: 0001322

We can also verify that the last packet sent has the following payload:

```

Packet sending...
ACK received...
Current packet size 11, rem3
Current payload ID44919a94SN0000009TXN0001322LAST1abf
Packet sending...
ACK received...
Current packet size 11, rem-8
Sent data equal to payload? True
ubuntu@ip-10-0-7-169:~/cs-145-project1$

```

Z=1

**Transaction last payload
Z = 1**

If we analyze the tracefile, the last packet sent for this transaction from my machine to the server was packet number 638. We can verify the payload of the packet and check that it indeed corresponds to the payload printed by the sender program.

Packet 638: Last payload sent

Frame 638: 79 bytes on wire (632 bits), 79 bytes captured (632 bits) on interface 0
 Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)
 Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175
 User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745
Destination Port: 9000
 Length: 45
 Checksum: 0x1d96 [incorrect] (maybe caused by "L")
 Destination port: 9000 (server assigned port)

Source port: 6745 (student assigned port)

**Last payload sent
SQN: 000009
TXN: 001322**

Packet 683: Last ACK message

Frame 683: 97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface 0
 Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)
 Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169
 User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000
Destination Port: 6745
 Length: 63
 Checksum: 0x0089 [correct]
 Source port: 9000 (server assigned port)

Destination port: 6745 (student assigned port)

**Server's last ACK for:
SEQ: 000009
TXN: 001322**

The server also sends its last ACK message for the corresponding packet shown as packet number 683 in the trace file. Thus, it can be shown that the sender implementation is indeed successful in sending data from the client to the server in this transaction as well.

4.5 Transaction 5

We do the same thing for another transaction and start by downloading a new payload file.

```
ubuntu@ip-10-0-7-169:~/cs-145-project1$ wget http://54.169.121.89:5000/get_data?student_id=44919a94
--2022-06-06 23:44:06-- http://54.169.121.89:5000/get_data?student_id=44919a94
Connecting to 54.169.121.89:5000... connected.
HTTP request sent, awaiting response... 200 OK
Length: 277 [text/plain]
Saving to: 'get_data?student_id=44919a94.4'

get_data?student_id= 100%[=====] Payload file filename for transaction 5 --.-KB/s in 0s
2022-06-06 23:44:06 (36.8 MB/s) - 'get_data?student_id=44919a94.4' saved [277/277]
```

Start tshark and use the new payload filename to execute the sender program via command line. We use the command "python3 sender.py -f get_data?student_id=44919a94.4 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94"

The screenshot displays the TGRS Transactions table and a terminal window. The table shows transaction 0001323 as 'Transaction alive' with a frequency of 1. The terminal shows the command to run the sender program and tshark capturing packets.

Transaction ID	Time Started	Duration	Result	Frequency Used
0001323	2022-06-06 23:44:28.614218	None	None	1
0001321	2022-06-06 23:39:20.983634	114.196	Successfully sent data	1
0001320	2022-06-06 23:36:37.021838	119.221	Successfully sent data	1
0001319	2022-06-06 23:26:31.474865	112.654	Successfully sent data	1
0001318	2022-06-06 23:23:40.580382	120.000	Failed to send data	1
0001317	2022-06-06 23:12:19.956418	111.322	Successfully sent data	1

Transaction ID: 0001323

Transaction alive

Frequency used of payload: 1

Command runs on terminal

```
ubuntu@ip-10-0-7-169:~/cs-145-project1$ python3 sender.py -f get_data?student_id=44919a94.4 -a 10.0.1.175 -s 9000 -c 6745 -i 44919a94
Trans ID: 0001323
Length: 277
Current payload ID44919a94SN000000TXN0001323LAST0pvOiuwqap
Packet sending...
ACK received...
Current packet size 15, rem268
Current payload ID44919a94SN000000TXN0001323LAST0PzuyPqrHnbXoGsK
Packet size 15, rem268
Transaction ID: 0001323
```

Tshark capturing packets

```
ubuntu@ip-10-0-7-169:~/cs-145-project1$ sudo tshark -i eth0 -w /home/tracefile4.pcapng
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
** (tshark:1957) 23:44:21.635482 [Main MESSAGE] -- Capture started.
** (tshark:1957) 23:44:21.635851 [Main MESSAGE] -- File: "/home/tracefile4.pcapng"
224
```

Note that this transaction has a transaction ID of 0001323. After successfully sending the packets, the sender program finishes executing and the result in the TGRS is updated and is shown to have **successfully sent data**.

Transactions

Show entries

Search:

Transaction ID	Time Started	Duration	Result	Frequency Used
0001323	2022-06-06 23:44:28.614218	115.178	Successfully sent data	1
0001321	2022-06-06 23:39:20.98363	115.186	Successfully sent data	1
0001320	2022-06-06 23:36:37.021838		Successfully sent data	1
0001319	2022-06-06 23:26:31.474865	112.654	Successfully sent data	1
0001318	2022-06-06 23:23:40.580382	120.000	Failed data	
	2022-06-06 23:12:19.956418	111.322	Successfully sent data	1

Transaction ID: 0001323
Time started: 23:44:28

Transaction result: Successfully sent data

Frequency used of payload: 1

Does original and sent payload match? Yes

Transaction time stamp matching in trace file and TGRS

```

Packet sending...
ACK received...
Current packet size 42, rem136
Current payload ID44919a94SN0000006TXN0001323LAST0KkIVBRzSeqpBtvbClQmFhATjexpeMeQom
OMttMmEW
Packet sending...
Server NACKed
Size not accepted: 42
Current payload ID44919a94SN0000006TXN0001323LAST0KkIVBRzSeqpBtvbClQmFhATjexpeMeQom
OMt
Packet sending...
ACK received...
Current packet size 37, rem99
Current payload ID44919a94SN0000009TXN0001323LAST0tMmEkWQZzYYCpOvWmResdXsUEJObckl
KaA
Packet sending...
ACK received...
Current packet size 37, rem62
Current payload ID44919a94SN0000008TXN0001323LAST0dVxEeDdPQbDRPexvsbuofdGTAYdhGpZrZq
SVP
Packet sending...
ACK received...
Current packet size 37, rem25
Current payload ID44919a94SN0000009TXN0001323LAST1irKWJyIEFAJGBxTNsdIVNxIq
Packet sending...
ACK received...
Current packet size 37, rem-12
Sent data equal to payload? True
ubuntu@ip-10-0-7-169:~/cs-145-project1$
ubuntu@ip-10-0-7-169:~/cs-145-project1$ sudo tshark -i eth0 -w /home/tracefile4.pcapng
Running as user "root" and group "root". This could be dangerous.
Capturing on 'eth0'
** (tshark:1957) 23:44:21.635482 [Main MESSAGE] -- Capture started.
** (tshark:1957) 23:44:21.635851 [Main MESSAGE] -- File: "/home/tracefile4.pcapng"
724
[0] 0:~bash$
  
```

Execution of sender finishes

By checking the transaction in the tracefile generated, we can verify that the intent message was indeed sent as packet 152 in the tracefile by filtering out the UDP packets and checking their payload. The server's accept message can also be verified as packet 153 which contains the transaction ID.

Packet 152: Intent message

Frame 152: 52 bytes on wire (416 bits), 52 bytes captured (416 bits) on interface 0

Ethernet II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e)

Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175

User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745 (student assigned port)

Destination Port: 9000 (server assigned port)

Length: 18

Checksum: 0x7bb2 (maybe caused by "l")

Packet 153: Accept message

Frame 153: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface 0

Ethernet II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)

Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169

User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000 (server assigned port)

Destination Port: 6745 (student assigned port)

Length: 15

Checksum: 0xde63 [correct]

Transaction ID: 0001323

We can also verify that the last packet sent has the following payload:

```
SVP
Packet sending...
ACK received...
Current packet size 37, rem25
Current payload ID44919a94SN0000009TXN0001323LAST1irKWJoyIEFAJGBxTNsdIVNxIq
Packet sending...
ACK received...
Current packet size 37, rem-12
Sent data equal to payload? True
ubuntu@ip-10-0-7-169:~/cs-145-project1$
```

Transaction last payload
Z = 1

If we analyze the tracefile, the last packet sent for this transaction from my machine to the server was packet number 643. We can verify the payload of the packet and check that it indeed corresponds to the payload printed by the sender program.

Packet 643: Last payload sent

643	23:46:13.776074747	10.0.7.169	10.0.1.175	UDP	101	6745 → 9000	Len=59	[UDP CHECKSUM INCORRECT]
687	23:46:23.792336174	10.0.1.175	10.0.7.169	UDP	97	9000 → 6745	Len=55	

Frame 643: 101 bytes on wire (808 bits), 101 bytes captured (808 bits) on interface 0
 Ethernet II, Src: 06:37:3c:93:56:96 (06:37:3c:93:56:96), Dst: 06:0e:6d:8c:45:00 (06:0e:6d:8c:45:00)
 Internet Protocol Version 4, Src: 10.0.7.169, Dst: 10.0.1.175
 User Datagram Protocol, Src Port: 6745, Dst Port: 9000

Source Port: 6745
 Destination Port: 9000
 Length: 67
 Checksum: 0x1dac (student assigned port)
 Destination port: 9000 (server assigned port)

0000 06 0e b4 c8 23 6e 06 37 3c 93 56 96 08 00 45 00 ...#n·7 <·V···E·
 0010 00 57 74 42 40 00 40 11 a8 fc 0a 00 07 a9 0a 00 ·WtB@·@· ······
 0020 01 af 1a 59 23 28 00 43 1d ac 49 44 34 34 39 31 ···Y#(·C ··ID4491
 0030 39 61 39 34 53 4e 30 30 30 30 30 39 54 58 4e 9a94SN00 00009TXN
 0040 30 30 30 31 33 32 33 4c 41 53 54 31 69 72 4b 57 0001323L AST1irKW
 0050 4a 6f 79 49 45 46 41 4a 47 42 78 54 4e 73 64 49 JoyIEFAJ GBxTNsdI
 0060 56 4e 78 49 71 VNxIq

Last payload sent
 SQN: 000009
 TXN: 001323

687: 97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface 0
 Ethernet II, Src: 06:0e:b4:c8:23:6e (06:0e:b4:c8:23:6e), Dst: 06:37:3c:93:56:96 (06:37:3c:93:56:96)
 Internet Protocol Version 4, Src: 10.0.1.175, Dst: 10.0.7.169
 User Datagram Protocol, Src Port: 9000, Dst Port: 6745

Source Port: 9000
 Destination Port: 6745
 Length: 63
 Checksum: 0x97b5 (server assigned port)
 Destination port: 6745 (student assigned port)

0000 06 37 3c 93 56 96 06 0e b4 c8 23 6e 08 00 45 00 ·7<·V··· ··#n··E·
 0010 00 53 5e fc 40 00 40 11 be 46 0a 00 01 af 0a 00 ·S·@·@· ·F·····
 0020 07 a9 23 28 1a 59 00 3f 97 b5 41 43 4b 30 30 30 ··#(·Y·? ··ACK000
 0030 30 30 30 39 54 58 4e 30 30 30 31 33 32 33 4d 44 0009TXN0 001323MD
 0040 35 31 31 36 64 65 61 36 34 38 37 37 64 38 32 64 5116dea6 4877d82d
 0050 32 33 65 39 39 31 32 34 39 63 31 66 61 65 38 61 23e99124 9c1fae8a
 0060 37

Server's last ACK for:
 SEQ: 000009
 TXN: 001323

The server also sends its last ACK message for the corresponding packet shown as packet number 687 in the trace file. Thus, it can be shown that the sender implementation is indeed successful in sending data from the client to the server in this transaction as well.

5 Conclusion

Although all testing transactions were successful in sending data, the implementation proved to be inconsistent especially for larger sets of payload (i.e. 4k+). Hence, only level 2 was declared to be implemented.