John Carlo Sumabat
2018-03037
Lab 1

CS 145 Project Documentation

Parameter-Adaptive Reliable UDP-based Protocol

1 Implementation Level

The implementation level is **level 3**. Thus, the program is able to send Intent Message (to the receiver or test server), receive the Accept Message (from the receiver or test server), and send the payload completely in 120 seconds.

2 Documentation Video

The documentation video can be accessed by clicking this link.

3 Network Setup

For trace file generation, I used TShark on an Amazon EC2 t2.micro instance. The network interface used was `eth0`. Trace analysis was done on a Windows machine.

4 Introduction

4.1 Implementation Overview

On a high level, the program works as follows:

- From a terminal, the name of the program is entered, followed by 5 optional arguments which define the program constants. For each argument that is not specified, a default value is used. Included in these constants are the IP address and port of the server to transact with.
- The program creates an intent message and sends it to the specified server address. Upon success, it receives a transaction ID, indicating that a transaction has begun. In case a previous transaction has not yet terminated, it retries again in 10 seconds.
- Upon receiving a transaction ID, the program begins parameter estimation to determine the maximum payload length that the server can process.
- Lastly, the program sends the remaining payload in chunks equal to the maximum payload length. It sends the data in a stop-and-wait manner, such that an ACK from the server is required for a succeeding packet to be sent. If no ACK is received for a while, the program terminates.

4.2 Parameter Estimation Algorithm

The parameter that the program estimates is the maximum payload length that the server will acknowledge. It achieves this as follows:

- The program initializes the maximum payload length as one eighth the size of the original payload.
- A packet is sent to the server containing the payload with the initial length.
- If there is no ACK received within half a second, the maximum payload length is reduced by 5%, and a new packet is sent.
- The process of decreasing the payload length repeats until an ACK is received or the payload length is 1 byte.
- Once an ACK is received, its checksum is used to determine which packet was acknowledged. From the said packet, the final maximum payload length is obtained.

This is explored in more detail in Code Block 8 in Section 5.4.

5 Implementation Details

5.1 Python Standard Library Facilities Used

Code Block 1 shows the standard library modules, functions, and classes that the program imports. Their uses are shown in Table 1.

Facility	Purpose
`argparse.ArgumentParser`	Class responsible for handling logic of parsing command line arguments.
`hashlib.md5`	Function to compute hash value of packet
`time.sleep`	Function to make thread pause execution
`socket`	Module that provides low-level networking interface

Table 1. Descriptions of imported facilities

Code Block 1 `import` statements

```
from argparse import ArgumentParser
from hashlib import md5
from time import sleep
import socket
```

5.2 Program Constants

Code Block 2 shows the program constants[1]. Default values are used unless a corresponding command line argument is supplied. Descriptions for the constants are shown in Table 2.

Line	Description
7, 11	Client address; used to bind a UDP socket for listening
9, 10	Server address; used for packet sending
8	Pathname of payload to send
12	Unique ID assigned to student

Table 2. Descriptions of program constants

5.3 Helper Functions

To help with testing and repetitive tasks, some helper functions were created.

- The function `announce` (Code Block 3) accepts a string and prints it with a border. It is used to differentiate between different “stages” of the program.
- Upon receiving an ACK, `parse_ack` (Code Block 4) parses the packet and returns its contents via a dictionary.
- The function `make_msg` (Code Block 5) constructs a data packet given a unique ID, sequence number, transaction ID, `LAST` flag, and payload. It returns this message as a string.
- Every time a packet is sent, the function `compute_checksum` (Code Block 6) is called to compute the hash value of the packet. This is used to determine the client packet that the server acknowledges.

[1] Although there are no “official” constants in Python, these variables are functionally the same since they never get modified after being initialized either by default or via command line arguments.

Code Block 2 Program constants

```
CLIENT_HOSTNAME = socket.gethostbyname(socket.gethostname())
FILE = "2b97c5ee.txt"  # file to be sent by client
SERVER_HOSTNAME = "10.0.7.141"  # IP address of receiver
UDP_PORT_SEND = 9000  # sending port for UDP
UDP_PORT_LISTEN = 6750  # listening port for UDP
ID = "2b97c5ee"  # unique student ID
```

Code Block 3 `announce` function

```
def announce(msg):
  """
  Print a message surrounded by a border for emphasis

  :param msg: message to be printed
  """
  print("\n" + "=" * len(msg))
  print(msg)
  print("=" * len(msg))
```

Code Block 4 `parse_ack` function

```
def parse_ack(ack):
  """
  Extract the contents of an ack packet

  :param ack: ack packet received
  :returns: contents of packet stored in dictionary
  """
  sn = ack[3:10]
  txn = ack[13:20]
  chksum = ack[23:]

  print(f"Received ack: {ack}")

  return {"sn": sn, "txn": txn, "chksum": chksum}
```

Code Block 5 `make_msg` function

```
def make_msg(idd, sn, txn, last, payload):
  """
  Construct a data packet

  :param idd: unique student ID
  :param sn: sequence number
  :param txn: transaction ID
  :param last: denotes whether the packet is the final packet
  :param payload: data to be sent
  :returns: formatted packet
  """
  return f"ID{idd}SN{sn:>07d}TXN{txn:>07d}LAST{last}{payload}"
```

Code Block 6 `compute_checksum` function

```
def compute_checksum(packet):
  """
  Compute the hash value of a packet

  :param packet: packet whose hash is to be computed
  :returns: 128-bit hash value
  """
  return md5(packet.encode("utf-8")).hexdigest()
```

5.4 Main Functions

When the client wishes to send a payload, the program performs this in three stages:

1. First, `begin_transaction` (Code Block 7) is called. The function begins in line 166 by creating a socket named `udp_socket` via `socket.socket`. The program notifies the user that it is sending an intent message by calling `announce`. Using the constants `CLIENT_HOSTNAME` and `UDP_PORT_LISTEN`, the program calls `socket.bind` to bind the socket to an address for receiving packets in line 169. In line 170, the intent message is constructed using the constant `ID`.

   In lines 174–184, the function attempts to retrieve a transaction ID by sending the intent message to the server (whose address is stored in the constants `SERVER_HOSTNAME` and `UDP_PORT_SEND`). If there is a preexisting alive transaction, the server specifies this, and the client will retry in 10 seconds. Otherwise, the server sends a transaction ID, and the transaction is commenced. Afterwards, the transaction ID is printed in line 186. Finally, the function calls `get_max_payload_size` with the UDP socket and transaction ID as parameters.

Code Block 7 `begin_transaction` function

```
162 def begin_transaction():
163   """
164   Send an intent message and commence a transaction
165   """
166   with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as udp_socket:
167     announce("SENDING INTENT MESSAGE")
168
169     udp_socket.bind((CLIENT_HOSTNAME, UDP_PORT_LISTEN))
170     intent_msg = f"ID{ID}"
171
172     print(f"Intent message: {intent_msg}")
173
174     while True:
175       udp_socket.sendto(
176         intent_msg.encode(), (SERVER_HOSTNAME, UDP_PORT_SEND)
177       )
178       data, addr = udp_socket.recvfrom(2048)
179
180       if data.decode() == "Existing alive transaction":
181         print("Existing alive transaction found. Retrying in 10s.")
182         sleep(10)
183       else:
184         break
185
186     print(f"Transaction ID: {data.decode()}")
187
188     get_max_payload_size(udp_socket, txn_number=data.decode())
```

2. Upon receiving a transaction ID, `get_max_payload_size` (Code Block 8) performs parameter estimation.

   For lines 121–125: first, the function sets the sequence number and `LAST` flag to `0`, and saves the transaction ID. It informs the user via `announce` that it will begin estimating the maximum payload size that the server will ACK.

   For lines 127–132: the function opens the file that contains the payload, which is specified in the constant `FILE`. An empty dictionary `sent_packets` is initialized which would contain the checksum of each packet sent and its corresponding payload length. The initial size of the payload is set to 1/8th the size of the file and is assigned to the variable `payload_size`. The timeout duration for receiving ACKs is set to 0.5 seconds.

   For lines 134–149 and 154–155: the client constructs a packet containing the contents of `FILE` from index `0` to `payload_size`. The checksum of this packet is obtained via `compute_checksum`. The client attempts to send this packet and waits 0.5 seconds for an ACK. If no ACK is received, then either the server dropped the packet or it accepted the packet but is currently processing it. In any case, the checksum and corresponding payload length of the packet are saved in `sent_packets`. The value of `payload_size` is then decreased by 5%, and the client tries again. However, if the value of `payload_size` is `1`, then the client will wait for an ACK indefinitely since the payload length cannot get any smaller.

   For lines 150–153: the `while` loop exits when the client receives **one** ACK from the server. The checksum of this ACK is used to find the corresponding payload size in `sent_packets`. The variable `payload_size` is then updated to represent the maximum payload length.

   After parameter estimation, `send_payload` is called (line 159).

3. Finally, the remaining payload is sent via `send_payload` (Code Block 9).

   For lines 74–78: the function begins by setting the `LAST` flag to `0` and the transaction ID as specified in the parameters. Since the client has received one ACK at this point, the sequence number `SN` is set to `1`. It calls `announce` to inform the user that the remaining payload will be transmitted to the server.

   For lines 80–86: the function also takes in a parameter `offset` which should be equal to the maximum payload size. This variable represents the location in the file where a payload begins. Before transmission, `FILE` is reopened and the timeout duration is set to 30 seconds. Since offset is updated for every ACKed packet, `payload_size` is set to the initial offset value.

   For lines 88–100: the client sends the remainder of the payload via a `while` loop. For every iteration, the client constructs a packet containing the contents of `FILE`. The content of the payload portion of the packet depends on the amount of remaining data to transmit. In case the contents of `FILE` are all exhausted, `LAST` is set to `1`. For each packet constructed, the checksum is also computed.

   For lines 103–111: the client waits for an ACK for each packet sent. If an ACK is received within 30 seconds, then the sequence number is incremented and `offset` is updated. Otherwise, the server has terminated the transaction. In this case (or when all of `FILE` is successfully sent), the program terminates. The user is informed of this via a final `announce` call.

Code Block 8 `get_max_payload_size` function

```
def get_max_payload_size(udp_socket, txn_number=0):
  """
  Determine maximum payload size

  :param udp_socket: active socket
  :param txn_number: transaction ID
  """
  SN = 0
  TXN = int(txn_number)
  LAST = 0

  announce("DETERMINING MAXIMUM PAYLOAD SIZE")

  with open(FILE) as file:
    f = file.read()

    sent_packets = {}
    payload_size = int(len(f) * 0.125)  # initial payload size: 1/8 of file
    udp_socket.settimeout(0.5)          # waiting time for ack: 0.5 seconds

    while True:
      PAYLOAD = f[0:payload_size]

      msg = make_msg(ID, SN, TXN, LAST, PAYLOAD)
      chksum = compute_checksum(msg)
      udp_socket.sendto(msg.encode(), (SERVER_HOSTNAME, UDP_PORT_SEND))

      sent_packets[chksum] = payload_size

      print(f"\nSent message: {msg}")
      print(f"Checksum: {chksum}")

      # decrease payload size by 5% for every timeout
      try:
        if payload_size == 1:
          udp_socket.settimeout(None)
        data, addr = udp_socket.recvfrom(2048)
        ack = parse_ack(data.decode())
        payload_size = sent_packets[ack["chksum"]]
        break
      except socket.timeout:
        payload_size = int(payload_size * 0.95)

    print(f"\nMax payload size = {payload_size} characters")

  send_payload(udp_socket, txn_number=txn_number, offset=payload_size)
```

Code Block 9 `send_payload` function

```
def send_payload(udp_socket, txn_number=0, offset=0):
  """
  Send the entire payload

  :param udp_socket: active socket
  :param txn_number: transaction ID
  :param offset: which section of the file to begin sending
  """
  SN = 1
  TXN = int(txn_number)
  LAST = 0

  announce("SENDING PAYLOAD")

  with open(FILE) as file:
    f = file.read()
    maxlen = len(f)

    payload_size = offset

    udp_socket.settimeout(30)

    while offset < maxlen:
      if offset + payload_size < maxlen:
        PAYLOAD = f[offset : offset + payload_size]
      else:
        PAYLOAD = f[offset:]
        LAST = 1

      msg = make_msg(ID, SN, TXN, LAST, PAYLOAD)
      chksum = compute_checksum(msg)
      udp_socket.sendto(msg.encode(), (SERVER_HOSTNAME, UDP_PORT_SEND))

      print(f"\nSent message: {msg} ({offset} / {len(f)})")
      print(f"Checksum:{' '*28}{chksum}")

      # end transaction if no longer receiving acks (120 seconds exceeded)
      try:
        data, addr = udp_socket.recvfrom(2048)
        ack = parse_ack(data.decode())
        SN += 1
        offset += payload_size
      except socket.timeout:
        break

  announce("TRANSACTION TERMINATED")
```

5.5 Program Entry Point

Code Block 10 shows where program execution begins. Upon launch, an `ArgumentParser` object is set up in lines 193–199 to handle command line arguments. The constants discussed in Section 5.2 are initialized in lines 204–208 and are revealed to the user in lines 210–216. Afterwards, `begin_transaction` is called to attempt initiating a transaction with the server.

Code Block 10 Program entry point

```
191 if __name__ == "__main__":
192   # command line argument parser
193   parser = ArgumentParser()
194 
195   parser.add_argument("-f", default=FILE)
196   parser.add_argument("-a", default=SERVER_HOSTNAME)
197   parser.add_argument("-s", type=int, default=UDP_PORT_SEND)
198   parser.add_argument("-c", type=int, default=UDP_PORT_LISTEN)
199   parser.add_argument("-i", default=ID)
200 
201   args = parser.parse_args()
202 
203   # initialize program constants
204   FILE = args.f
205   SERVER_HOSTNAME = args.a
206   UDP_PORT_SEND = args.s
207   UDP_PORT_LISTEN = args.c
208   ID = args.i
209 
210   announce("TRANSACTION DETAILS")
211 
212   print(f"File to send: {FILE}")
213   print(f"IP address of server: {SERVER_HOSTNAME}")
214   print(f"Port used by server: {UDP_PORT_SEND}")
215   print(f"Port used by client: {UDP_PORT_LISTEN}")
216   print(f"Unique ID: {ID}")
217 
218   begin_transaction()
```

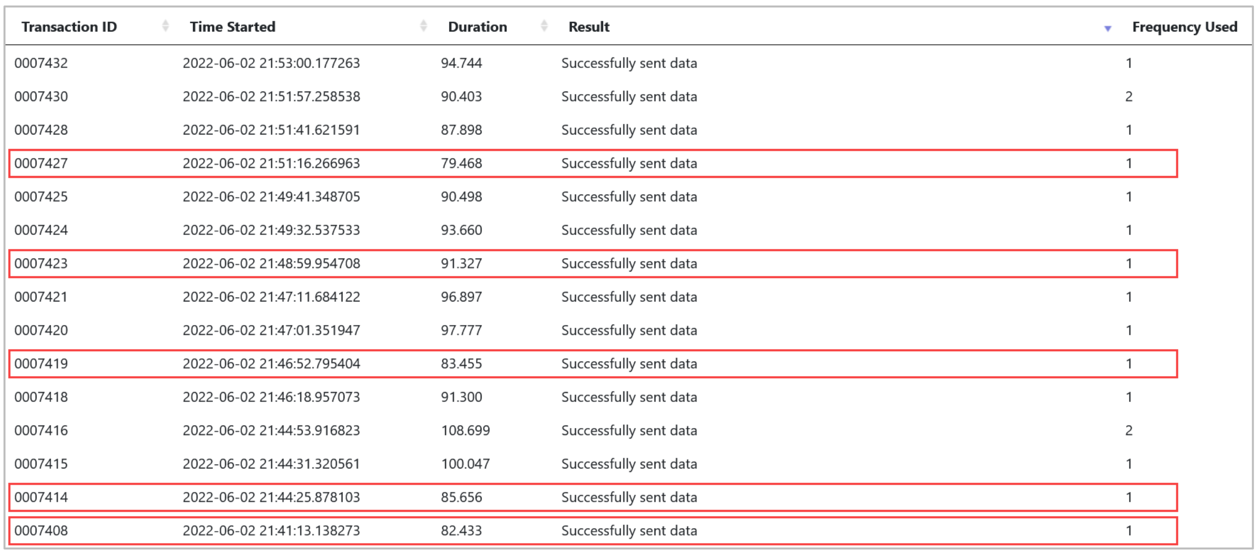
6 Testing the Program

To test whether the implementation works, 5 transactions were initiated with the server. Table 3 shows the command line arguments supplied when testing the program.

Flag	Supplied Option
-f	2b97c5ee.txt
-a	10.0.7.141
-s	9000
-c	6750
-i	2b97c5ee

Table 3. Command line arguments used for testing

For each test, a new payload is downloaded. Figure 1 highlights each of the 5 transactions and their respective IDs. As expected of a level 3 implementation, all tests have successfully sent the data in at most 120 seconds and have “Frequency Used: 1”.



Transaction ID	Time Started	Duration	Result	Frequency Used
0007432	2022-06-02 21:53:00.177263	94.744	Successfully sent data	1
0007430	2022-06-02 21:51:57.258538	90.403	Successfully sent data	2
0007428	2022-06-02 21:51:41.621591	87.898	Successfully sent data	1
0007427	2022-06-02 21:51:16.266963	79.468	Successfully sent data	1
0007425	2022-06-02 21:49:41.348705	90.498	Successfully sent data	1
0007424	2022-06-02 21:49:32.537533	93.660	Successfully sent data	1
0007423	2022-06-02 21:48:59.954708	91.327	Successfully sent data	1
0007421	2022-06-02 21:47:11.684122	96.897	Successfully sent data	1
0007420	2022-06-02 21:47:01.351947	97.777	Successfully sent data	1
0007419	2022-06-02 21:46:52.795404	83.455	Successfully sent data	1
0007418	2022-06-02 21:46:18.957073	91.300	Successfully sent data	1
0007416	2022-06-02 21:44:53.916823	108.699	Successfully sent data	2
0007415	2022-06-02 21:44:31.320561	100.047	Successfully sent data	1
0007414	2022-06-02 21:44:25.878103	85.656	Successfully sent data	1
0007408	2022-06-02 21:41:13.138273	82.433	Successfully sent data	1

Figure 1. Screenshot of Transaction Generation/Results Server

To explore the details of each transaction, we would show the following via screenshots of the terminal and trace files:

- The intent message sent and the transaction ID received
- The first packet to be acknowledged by the server and the corresponding ACK
- The last packet to be sent by the client and the last ACK sent by the server

For brevity, some screenshots show a combined view of the program's terminal output and the corresponding trace file, as demonstrated by Figure 2. Additionally, only UDP packets were captured, since all packets sent by the client and server are in UDP.

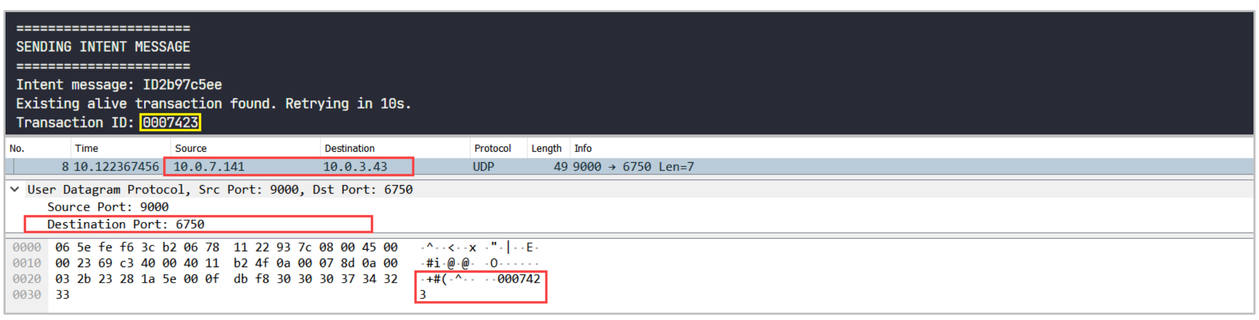


Figure 2. Sample of combined view of terminal output and Wireshark

6.1 First Test (Transaction ID: 0007408)

For this test, the corresponding trace file is `test1_transaction0007408.pcap`.

Figure 3 shows packet 5 which contains the intent message "`ID2b97c5ee`" sent to the IP address of the server via the supplied port `9000`. Figure 4 shows through the terminal and packet 6 that the transaction ID sent by the server is `0007408`. Note that the source and destination IP addresses in this packet are reversed, and that the server is addressing the supplied port `6750`.

As shown in packet 29 via Figure 5, the first ACK that the server sent corresponds to the packet with a payload size of 77 bytes. Included in the ACK is the checksum of the said packet. This checksum corresponds to packet 18 in Figure 6, as shown by the yellow rectangle.

Figure 7 shows the last packet sent by the client (with `LAST` flag set to `1`) and the corresponding last ACK sent by the server. Since the checksum of the packet and the ACK are equivalent, this is the correct pair of packets. Figure 8 shows via packet 56 that the last client packet was indeed sent to the correct IP address (`10.0.7.141`) and port (`9000`). In the same way, Figure 9 shows through packet 57 that the content of the last ACK matches with what is shown in the terminal and that the IP address and port (`6750`) belong to the client, as expected.

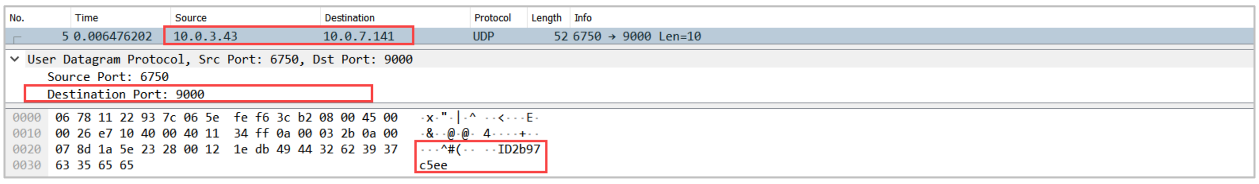


Figure 3. Intent message for the first transaction

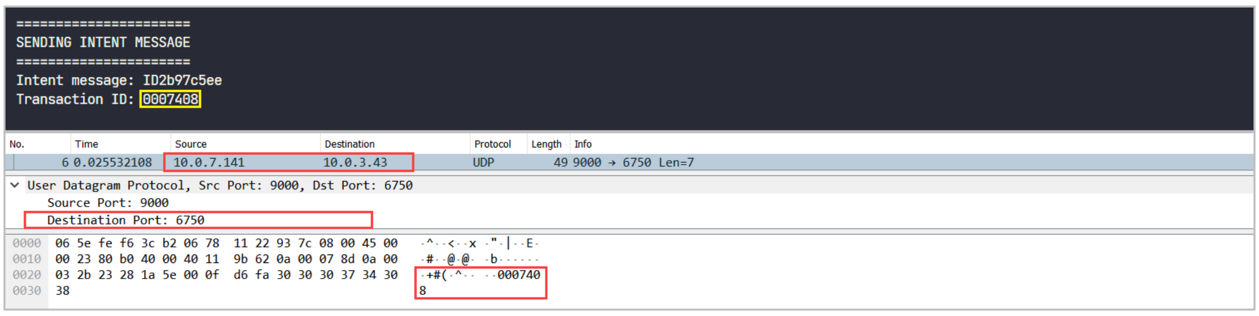


Figure 4. Transaction ID for the first transaction

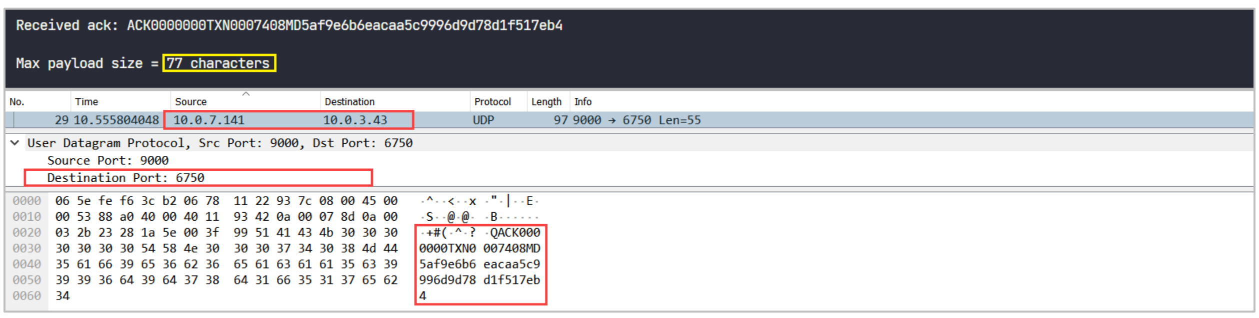


Figure 5. First acknowledgement received for the first transaction

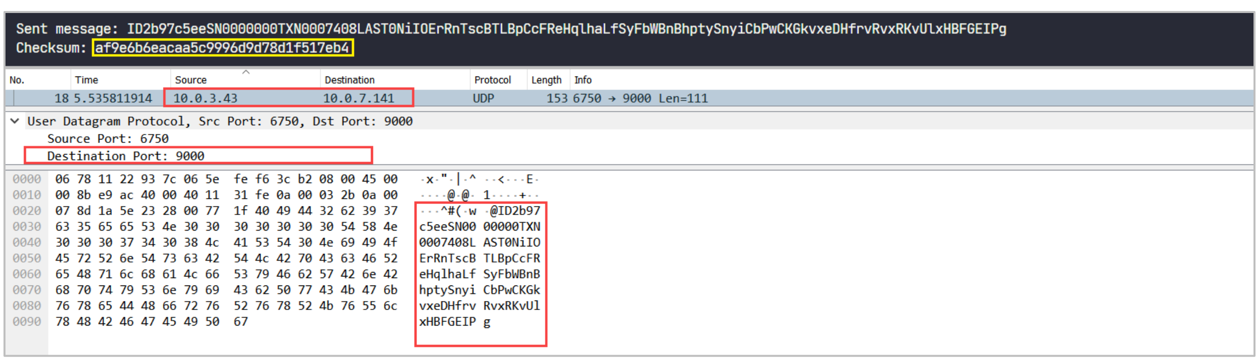


Figure 6. First packet that was acknowledged in the first transaction

Figure 7. Last packet sent and last ACK received in the first transaction

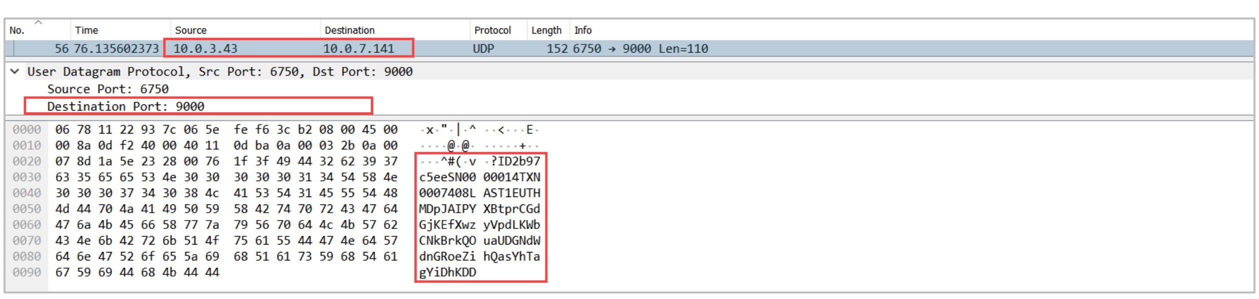


Figure 8. Wireshark view of last packet sent in first transaction

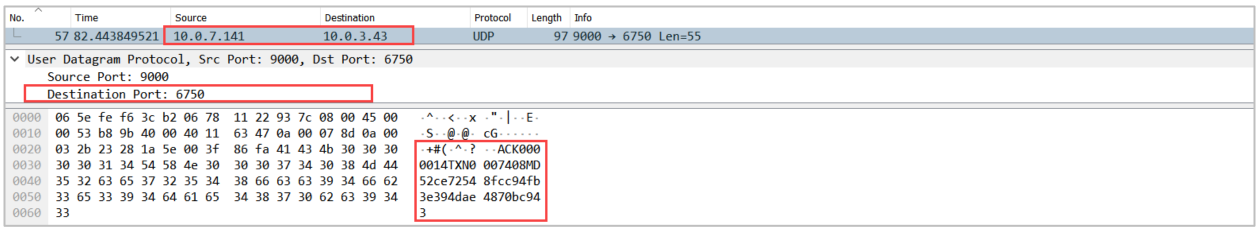


Figure 9. Wireshark view of last ACK received in first transaction

6.2 Second Test (Transaction ID: 0007414)

For this test, the corresponding trace file is `test2_transaction0007414.pcap`.

Figure 10 shows packet 5 which contains the intent message “`ID2b97c5ee`” sent to the IP address of the server via the supplied port `9000`. Figure 11 shows through the terminal and packet 6 that the transaction ID sent by the server is `0007414`. Note that the source and destination IP addresses in this packet are reversed, and that the server is addressing the supplied port `6750`.

As shown in packet 28 via Figure 12, the first ACK that the server sent corresponds to the packet with a payload size of 94 bytes. Included in the ACK is the checksum of the said packet. This checksum corresponds to packet 15 in Figure 13, as shown by the yellow rectangle.

Figure 14 shows the last packet sent by the client (with `LAST` flag set to `1`) and the corresponding last ACK sent by the server. Since the checksum of the packet and the ACK are equivalent, this is the correct pair of packets. Figure 15 shows via packet 51 that the last client packet was indeed sent to the correct IP address (`10.0.7.141`) and port (`9000`). In the same way, Figure 16 shows through packet 52 that the content of the last ACK matches with what is shown in the terminal and that the IP address and port (`6750`) belong to the client, as expected.

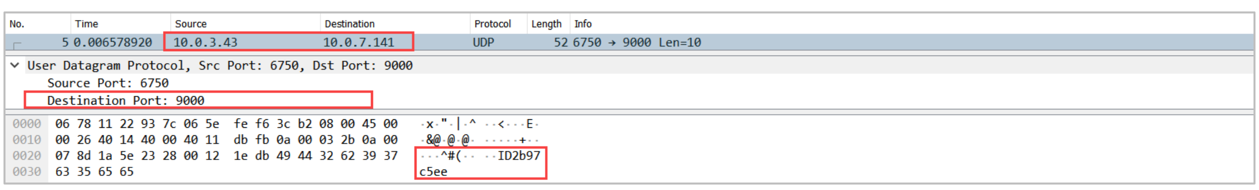


Figure 10. Intent message for the second transaction

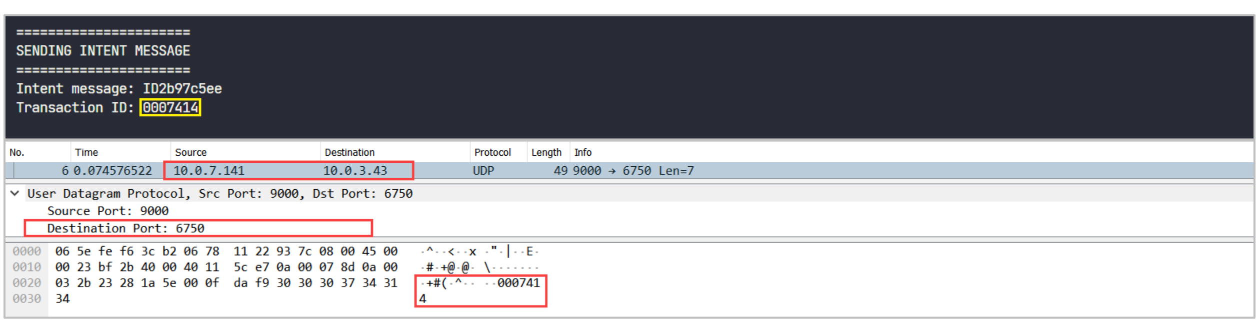


Figure 11. Transaction ID for the second transaction

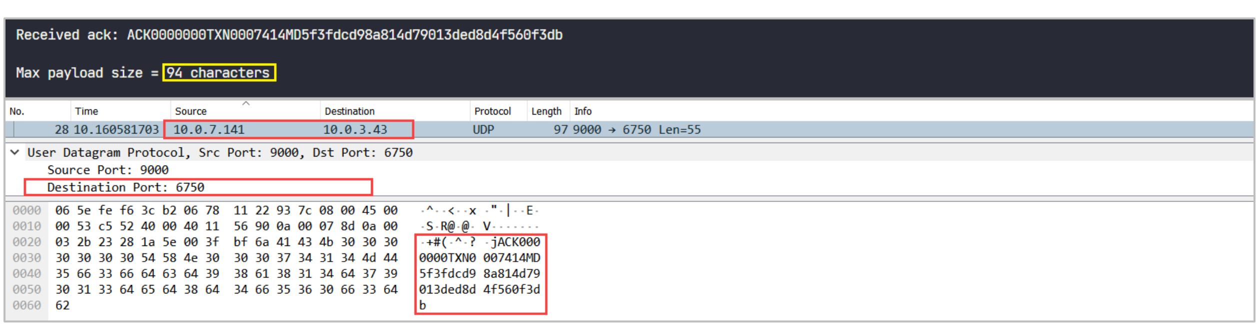


Figure 12. First acknowledgement received for the second transaction

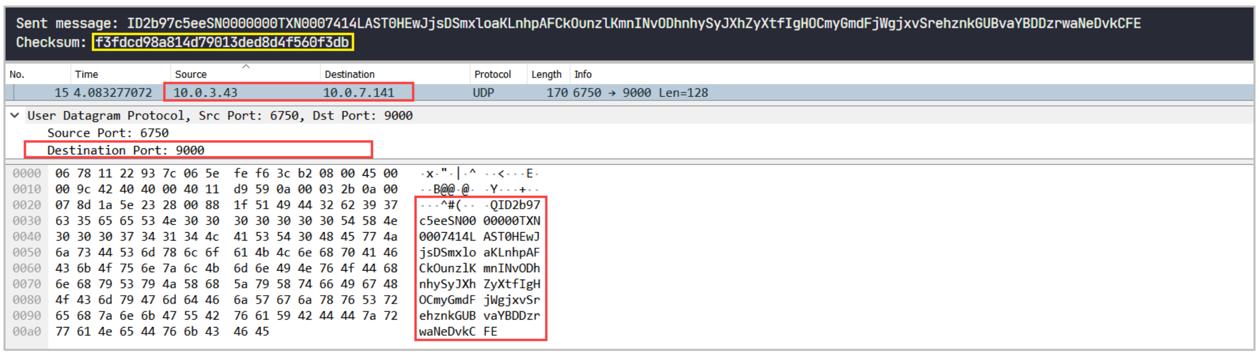


Figure 13. First packet that was acknowledged in the second transaction

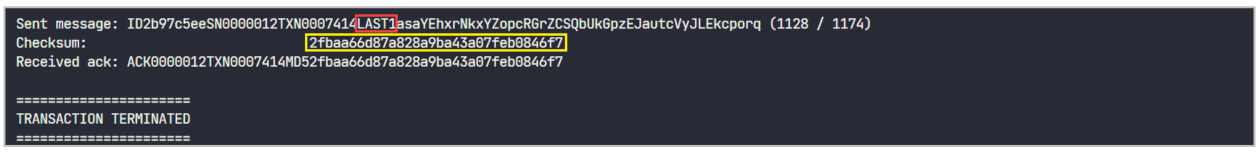


Figure 14. Last packet sent and last ACK received in the second transaction

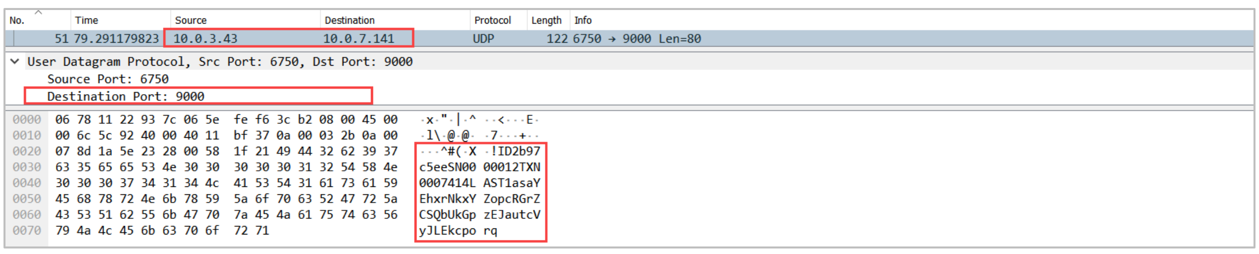


Figure 15. Wireshark view of last packet sent in second transaction

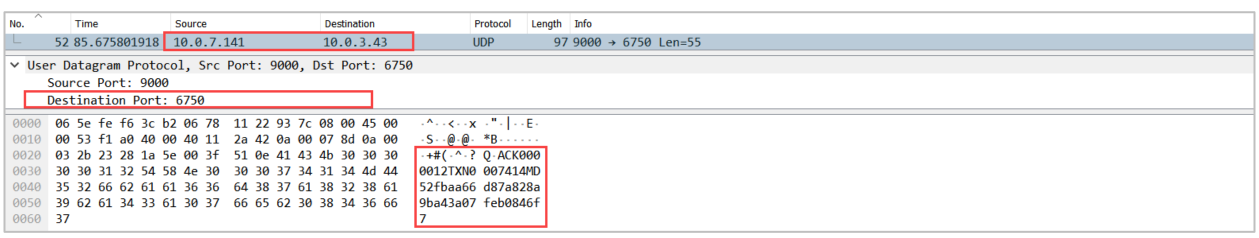


Figure 16. Wireshark view of last ACK received in second transaction

6.3 Third Test (Transaction ID: 0007419)

For this test, the corresponding trace file is `test3_transaction0007419.pcap`.

Figure 17 shows packet 5 which contains the intent message "`ID2b97c5ee`" sent to the IP address of the server via the supplied port `9000`. Figure 18 shows through the terminal and packet 6 that the transaction ID sent by the server is `0007419`. Note that the source and destination IP addresses in this packet are reversed, and that the server is addressing the supplied port `6750`.

As shown in packet 27 via Figure 19, the first ACK that the server sent corresponds to the packet with a payload size of 32 bytes. Included in the ACK is the checksum of the said packet. This checksum corresponds to packet 14 in Figure 20, as shown by the yellow rectangle.

Figure 21 shows the last packet sent by the client (with `LAST` flag set to `1`) and the corresponding last ACK sent by the server. Since the checksum of the packet and the ACK are equivalent, this is the correct pair of packets. Figure 22 shows via packet 50 that the last client packet was indeed sent to the correct IP address (`10.0.7.141`) and port (`9000`). In the same way, Figure 23 shows through packet 51 that the content of the last ACK matches with what is shown in the terminal and that the IP address and port (`6750`) belong to the client, as expected.

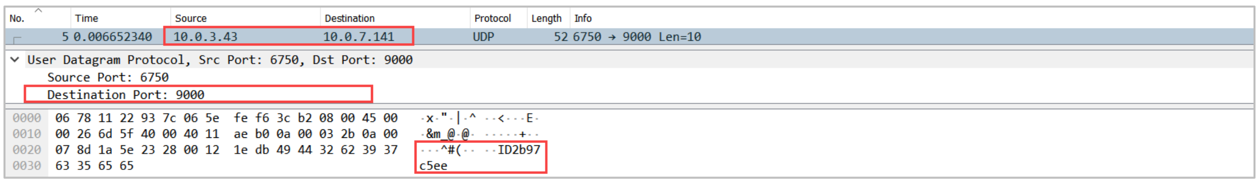


Figure 17. Intent message for the third transaction

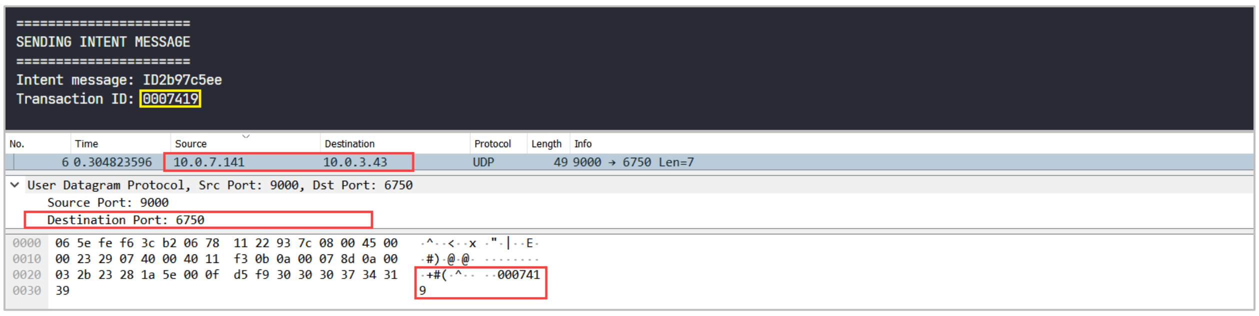


Figure 18. Transaction ID for the third transaction

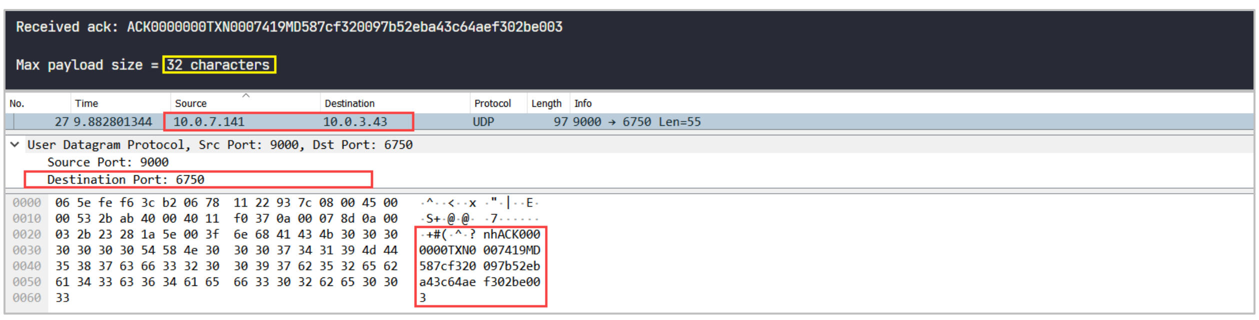


Figure 19. First acknowledgement received for the third transaction

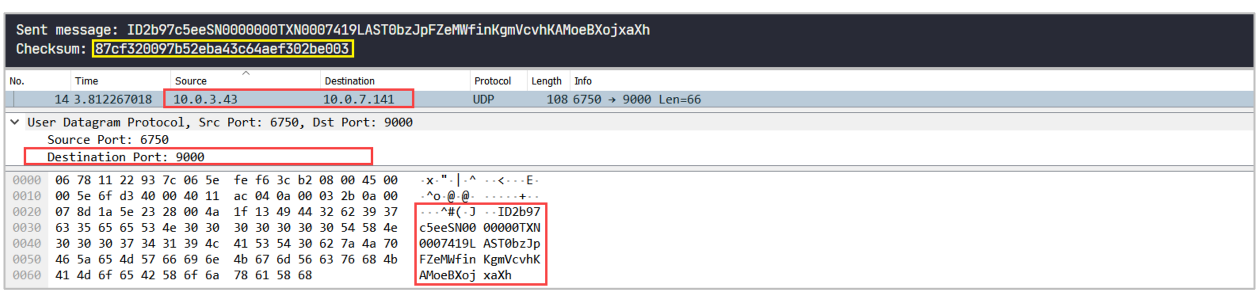


Figure 20. First packet that was acknowledged in the third transaction

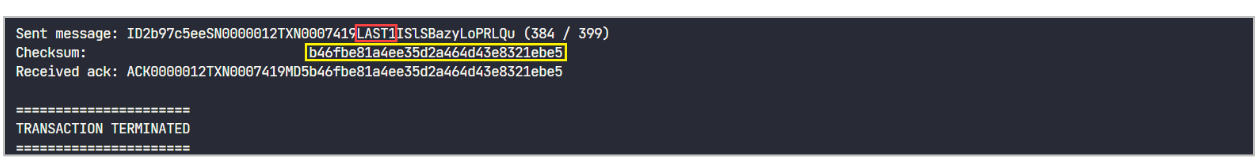


Figure 21. Last packet sent and last ACK received in the third transaction

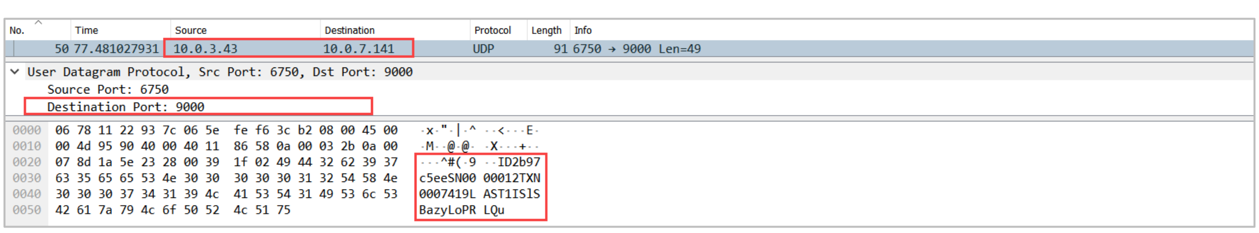


Figure 22. Wireshark view of last packet sent in third transaction

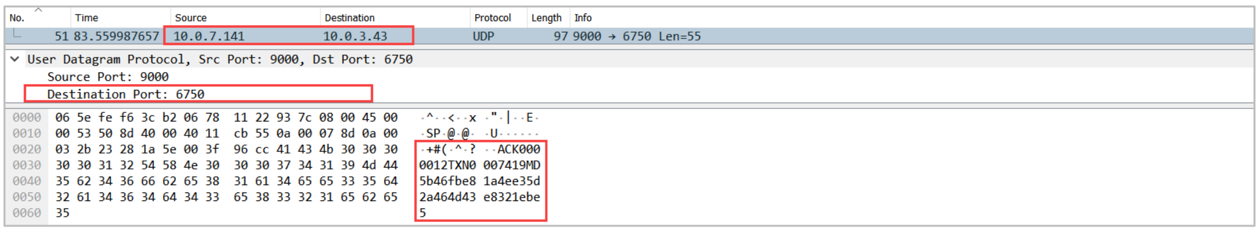


Figure 23. Wireshark view of last ACK received in third transaction

6.4 Fourth Test (Transaction ID: 0007423)

For this test, the corresponding trace file is `test4_transaction0007423.pcap`.

Figure 24 shows packet 7 which contains the intent message "`ID2b97c5ee`" sent to the IP address of the server via the supplied port `9000`. Figure 25 shows through the terminal and packet 8 that the transaction ID sent by the server is `0007423`. Note that the source and destination IP addresses in this packet are reversed, and that the server is addressing the supplied port `6750`.

As shown in packet 34 via Figure 26, the first ACK that the server sent corresponds to the packet with a payload size of 21 bytes. Included in the ACK is the checksum of the said packet. This checksum corresponds to packet 27 in Figure 27, as shown by the yellow rectangle.

Figure 28 shows the last packet sent by the client (with `LAST` flag set to `1`) and the corresponding last ACK sent by the server. Since the checksum of the packet and the ACK are equivalent, this is the correct pair of packets. Figure 29 shows via packet 85 that the last client packet was indeed sent to the correct IP address (`10.0.7.141`) and port (`9000`). In the same way, Figure 30 shows through packet 86 that the content of the last ACK matches with what is shown in the terminal and that the IP address and port (`6750`) belong to the client, as expected.

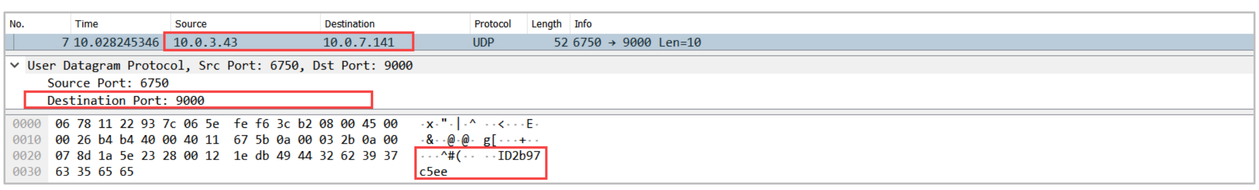


Figure 24. Intent message for the fourth transaction

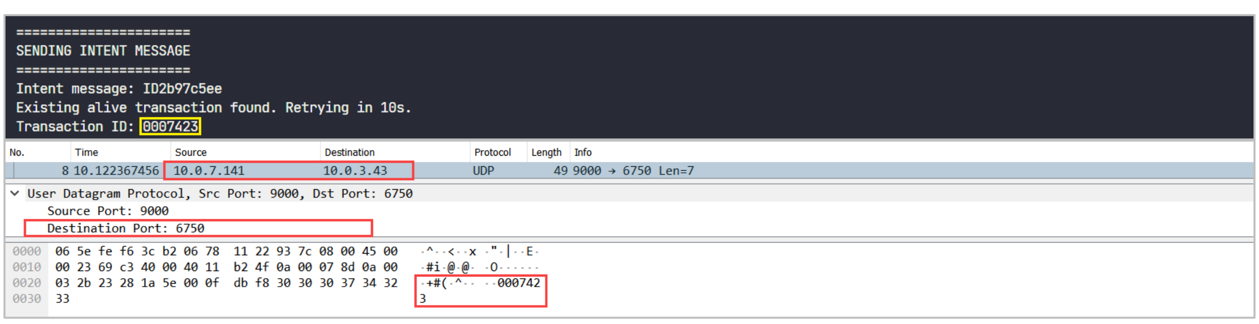


Figure 25. Transaction ID for the fourth transaction

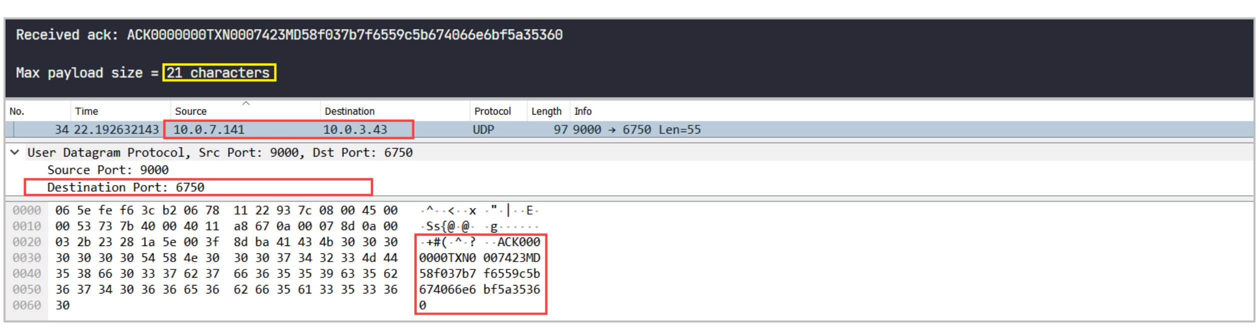


Figure 26. First acknowledgement received for the fourth transaction

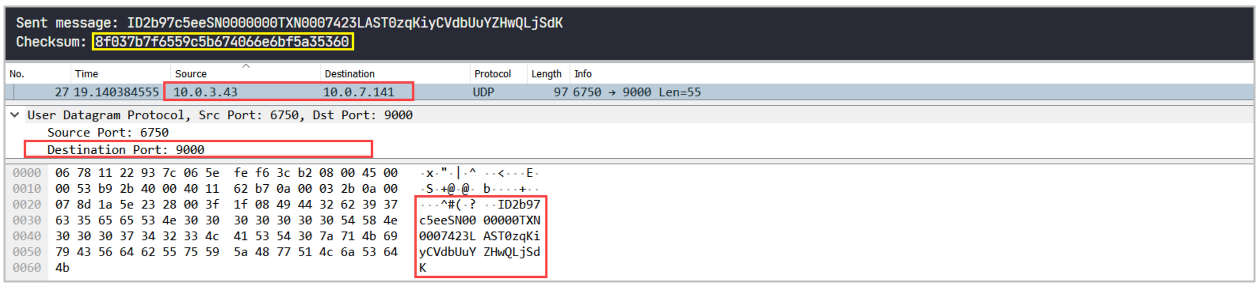


Figure 27. First packet that was acknowledged in the fourth transaction

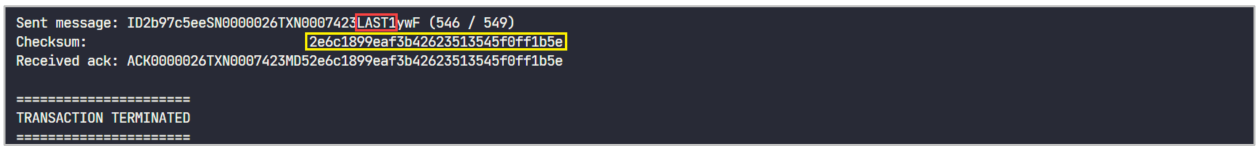


Figure 28. Last packet sent and last ACK received in the fourth transaction

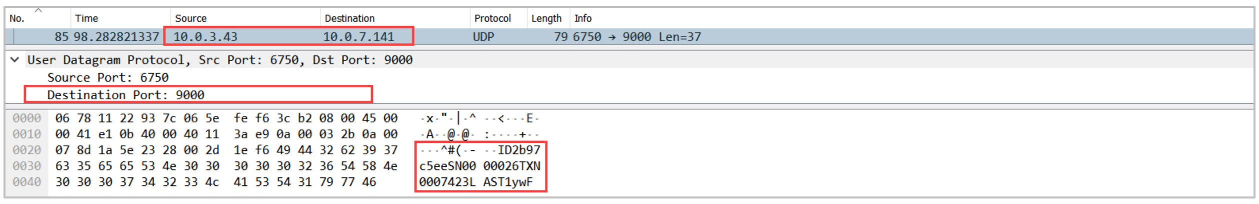


Figure 29. Wireshark view of last packet sent in fourth transaction

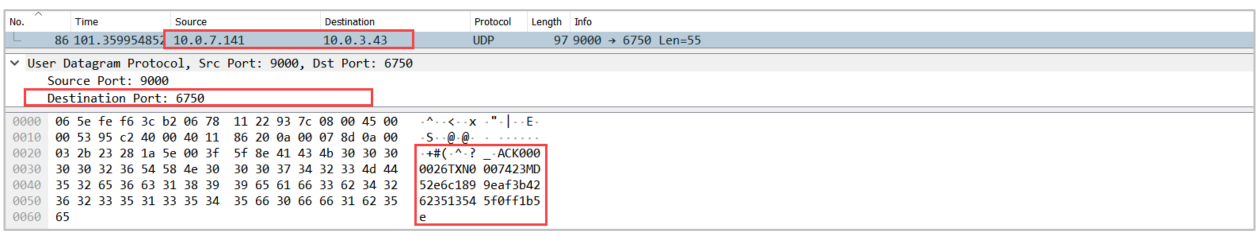


Figure 30. Wireshark view of last ACK received in fourth transaction

6.5 Fifth Test (Transaction ID: 0007427)

For this test, the corresponding trace file is `test5_transaction0007427.pcap`.

Figure 31 shows packet 5 which contains the intent message "`ID2b97c5ee`" sent to the IP address of the server via the supplied port `9000`. Figure 32 shows through the terminal and packet 6 that the transaction ID sent by the server is `0007427`. Note that the source and destination IP addresses in this packet are reversed, and that the server is addressing the supplied port `6750`.

As shown in packet 25 via Figure 33, the first ACK that the server sent corresponds to the packet with a payload size of 38 bytes. Included in the ACK is the checksum of the said packet. This checksum corresponds to packet 11 in Figure 34, as shown by the yellow rectangle.

Figure 35 shows the last packet sent by the client (with `LAST` flag set to `1`) and the corresponding last ACK sent by the server. Since the checksum of the packet and the ACK are equivalent, this is the correct pair of packets. Figure 36 shows via packet 44 that the last client packet was indeed sent to the correct IP address (`10.0.7.141`) and port (`9000`). In the same way, Figure 37 shows through packet 45 that the content of the last ACK matches with what is shown in the terminal and that the IP address and port (`6750`) belong to the client, as expected.

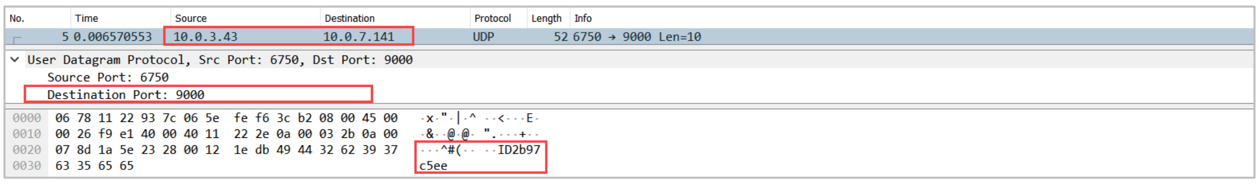


Figure 31. Intent message for the fifth transaction

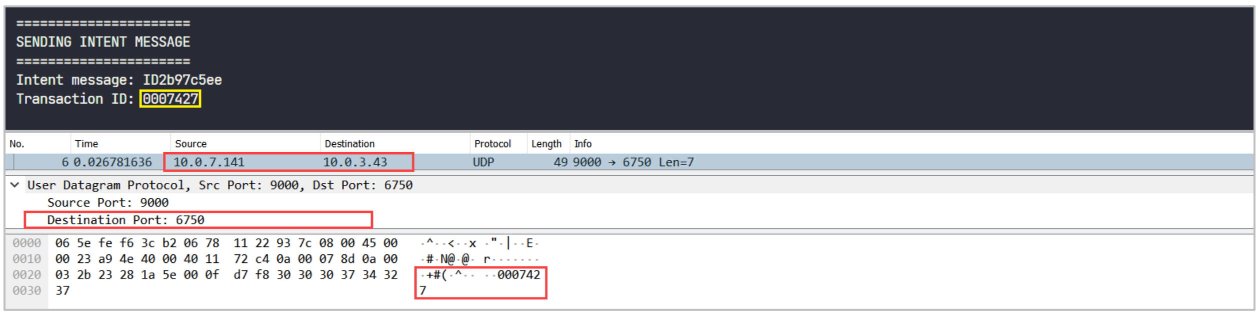


Figure 32. Transaction ID for the fifth transaction

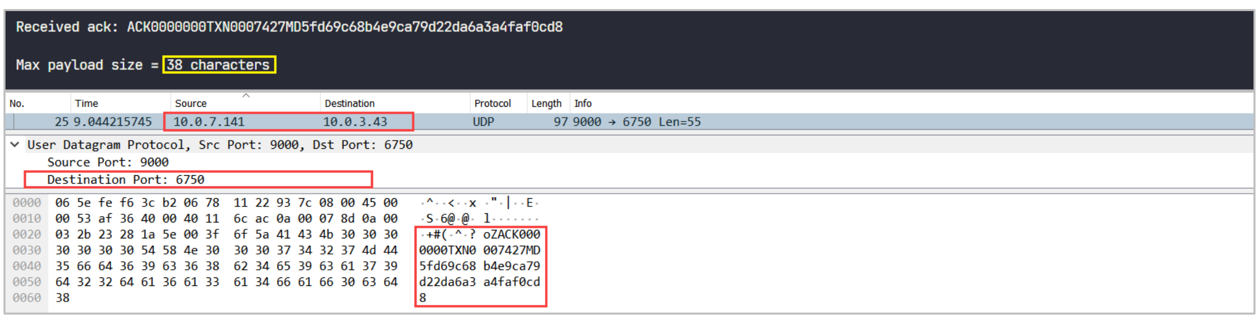


Figure 33. First acknowledgement received for the fifth transaction

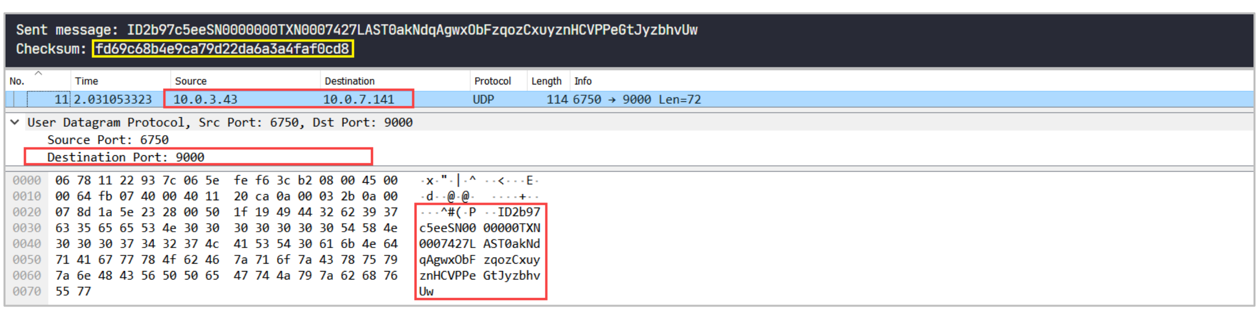


Figure 34. First packet that was acknowledged in the fifth transaction

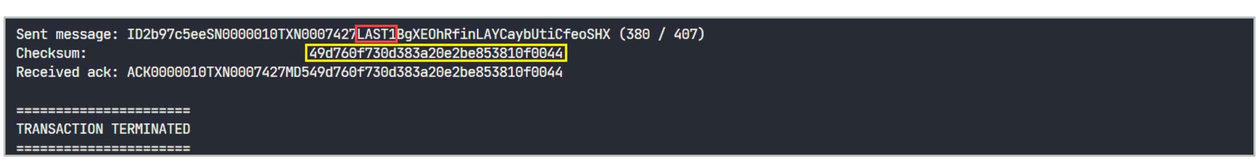


Figure 35. Last packet sent and last ACK received in the fifth transaction

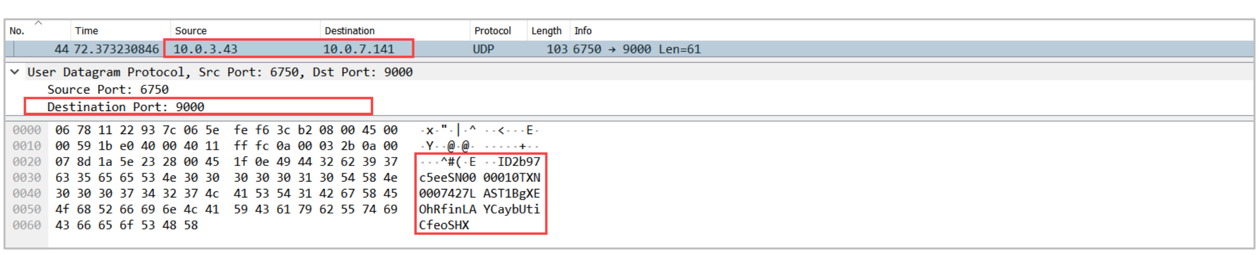


Figure 36. Wireshark view of last packet sent in fifth transaction

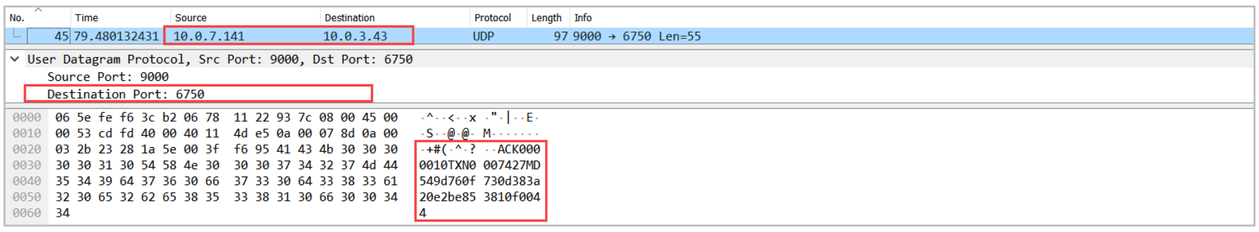


Figure 37. Wireshark view of last ACK received in fifth transaction