

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

Department of Computer Science
College of Engineering
University of Philippines Diliman

1 Introduction

1.1 Project Specification

The given project requires us to implement a working sender program under a custom UDP-based protocol. The program should be at minimum capable of initiating a transaction with a receiver. The grading depends on the level of functionality of the sender program. These functionalities are grouped in levels, with each increase in level corresponding to a higher capability of the sender program.

1.2 Declaration of Submitted Tasks

This implementation of the project delivers only the bare minimum of the sender program at Level 1 grading. That is, The program should be at minimum capable of sending **Intent Messages** to a receiver and receiving the **Accept Message** reply. There is an included functionality of a basic sender for testing purposes, but it is not good enough for the Level 2 grading as it is very inefficient, static and incapable of error handling.

1.3 Documentation

The video documentation for this project is provided by the link below:

<https://drive.google.com/file/d/1sIBwk0a-3Ave28UJdv9hPi17k5AsSoqr/view?usp=sharing>

The github repository for this project is provided by the link below:

<https://github.com/KadinaruDess/CS145Proj>

2 Implementation

2.1 Proof of Correctness

Before diving deep into the sender program implementation code, we must first ensure that the implementation works as intended. The testing is performed through the provided **Amazon EC2 instance**, with the sender program interacting with the provided test server as the receiver.

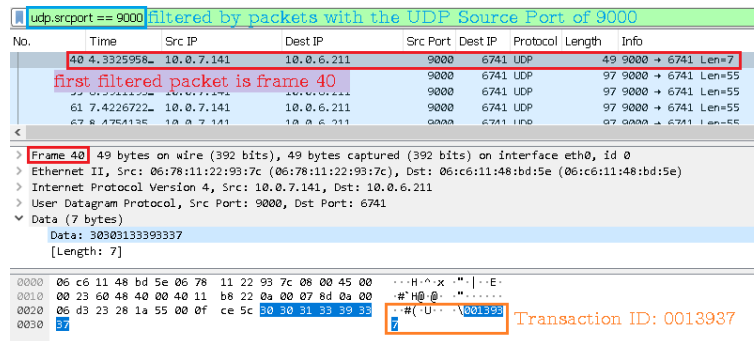
This testing will be proven by showing 5 experiments of unique transactions generated through the provided **Transaction Generation/Results Server (TGRS)**¹ link unique to each student, and matching the logged information of the experiments with the data in the provided **Transaction Logs**².

The information on the testing is provided through the trace files, which shows the packets sent and received by the **EC2 instance** for the duration of the experiments. These trace files were generated through the **tshark** functionality of **Ubuntu**.

Generated Trace Files

First thing to do is to determine the transaction ID's of each experiment. We can easily determine each trace file's transaction ID by looking at the **Accept Message** packet, as each packet informs the sender side of the transaction ID of the experiment corresponding to the trace file. While the all the generated trace files are heavily populated by packets, we can simply filter the trace files with the UDP packets with source port of 9000 (which was the provided port of the test server), and checking the payload of the first among the filtered packets as the **Accept Message** packet is also the first message sent by the receiver to the sender.

Images 2.1.1 to 2.1.5 below provides the screenshots of the **Accept Message** packets for each of the experiment, and from the **ASCII** representations of they payloads we can get that the transaction IDs of the 5 experiments are 13937, 13994, 14007, 14023 and 14032.



2.1.1: Accept Message packet of Experiment 1 in capture1.pcap tracefile

- 1 http://3.0.248.41:5000/get_data?student_id=03383a1b, **03383a1b** being my unique ID
- 2 <http://3.0.248.41:5000/transactions>

udp.srcport == 9000 filtered by packets with the UDP Source Port of 9000

No.	Time	Src IP	Dest IP	Src Port	Dest IP	Protocol	Length	Info
42	6.6757073	10.0.7.141	10.0.6.211	9000	6741	UDP	49	9000 → 6741 Len=7
first filtered packet is frame 42								
95	18.766087	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55
115	22.706044	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55

> Frame 42: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface eth0, id 0
 > Ethernet II, Src: 06:78:11:22:93:7c (06:78:11:22:93:7c), Dst: 06:c6:11:48:bd:5e (06:c6:11:48:bd:5e)
 > Internet Protocol Version 4, Src: 10.0.7.141, Dst: 10.0.6.211
 > User Datagram Protocol, Src Port: 9000, Dst Port: 6741
 > Data (7 bytes)
 Data: 30303133393934
 [Length: 7]

```

0000 06 c6 11 48 bd 5e 06 78 11 22 93 7c 08 00 45 00  ...H..x...|..E.
0010 00 23 d1 26 40 00 40 11 47 44 0a 00 07 8d 0a 00  #. @. @. ....
0020 06 d3 23 28 1a 55 00 0f d1 5e 30 30 31 33 39 39  #(-U... 001399
0030
  
```

Transaction ID: 0013994

2.1.2: Accept Message packet of Experiment 2 in capture2.pcap tracefile

udp.srcport == 9000 filtered by packets with the UDP Source Port of 9000

No.	Time	Src IP	Dest IP	Src Port	Dest IP	Protocol	Length	Info
73	11.956458	10.0.7.141	10.0.6.211	9000	6741	UDP	49	9000 → 6741 Len=7
first filtered packet is frame 73								
100	19.049325	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55
120	21.081043	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55
134	24.100075	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55

> Frame 73: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface eth0, id 0
 > Ethernet II, Src: 06:78:11:22:93:7c (06:78:11:22:93:7c), Dst: 06:c6:11:48:bd:5e (06:c6:11:48:bd:5e)
 > Internet Protocol Version 4, Src: 10.0.7.141, Dst: 10.0.6.211
 > User Datagram Protocol, Src Port: 9000, Dst Port: 6741
 > Data (7 bytes)
 Data: 30303134303037
 [Length: 7]

```

0000 06 c6 11 48 bd 5e 06 78 11 22 93 7c 08 00 45 00  ...H..x...|..E.
0010 00 23 7d 2f 40 00 40 11 9b 3b 0a 00 07 8d 0a 00  #. @. @. ....
0020 06 d3 23 28 1a 55 00 0f d7 5e 30 30 31 34 30 36  #(-U... 001400
0030
  
```

Transaction ID: 0014007

2.1.3: Accept Message packet of Experiment 3 in capture3.pcap tracefile

udp.srcport == 9000

No.	Time	Src IP	Dest IP	Src Port	Dest IP	Protocol	Length	Info
90	8.7009377	10.0.7.141	10.0.6.211	9000	6741	UDP	49	9000 → 6741 Len=7
first filtered packet is frame 90								
102	10.731219	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55
108	11.760051	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55
114	13.767787	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55

> Frame 90: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface eth0, id 0
 > Ethernet II, Src: 06:78:11:22:93:7c (06:78:11:22:93:7c), Dst: 06:c6:11:48:bd:5e (06:c6:11:48:bd:5e)
 > Internet Protocol Version 4, Src: 10.0.7.141, Dst: 10.0.6.211
 > User Datagram Protocol, Src Port: 9000, Dst Port: 6741
 > Data (7 bytes)
 Data: 303031343030233
 [Length: 7]

```

0000 06 c6 11 48 bd 5e 06 78 11 22 93 7c 08 00 45 00  ...H..x...|..E.
0010 00 23 25 fc 40 00 40 11 f2 6e 0a 00 07 8d 0a 00  #. @. @. ....
0020 06 d3 23 28 1a 55 00 0f db 5c 30 30 31 34 30 32  #(-U... 001402
0030
  
```

Transaction ID: 0014023

2.1.4: Accept Message packet of Experiment 4 in capture4.pcap tracefile

udp.srcport == 9000

No.	Time	Src IP	Dest IP	Src Port	Dest IP	Protocol	Length	Info
49	6.4902405	10.0.7.141	10.0.6.211	9000	6741	UDP	49	9000 → 6741 Len=7
first filtered packet is frame 49								
116	20.656641	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55
134	24.857678	10.0.7.141	10.0.6.211	9000	6741	UDP	97	9000 → 6741 Len=55

> Frame 49: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface eth0, id 0
 > Ethernet II, Src: 06:78:11:22:93:7c (06:78:11:22:93:7c), Dst: 06:c6:11:48:bd:5e (06:c6:11:48:bd:5e)
 > Internet Protocol Version 4, Src: 10.0.7.141, Dst: 10.0.6.211
 > User Datagram Protocol, Src Port: 9000, Dst Port: 6741
 > Data (7 bytes)
 Data: 303031343030332
 [Length: 7]

```

0000 06 c6 11 48 bd 5e 06 78 11 22 93 7c 08 00 45 00  ...H..x...|..E.
0010 00 23 28 32 40 00 40 11 f0 38 0a 00 07 8d 0a 00  #. @. @. ....
0020 06 d3 23 28 1a 55 00 0f dc 5b 30 30 31 34 30 33  #(-U... 001403
0030
  
```

Transaction ID: 0014032

2.1.5: Accept Message packet of Experiment 5 in capture5.pcap tracefile

Transaction Logs

Looking up the derived transaction numbers from the **Transactions Log** show that they are all indeed registered, and are all unique transactions with their Frequency Used values of 1 as shown in images 2.1.6 to 2.1.10. That they all share **Failed to send data** statuses however also reinforces the sender program's declaration of having the Level 1 grading.

0013937	2022-06-07 08:51:59.428818	120.000	Failed to send data	1
2.1.6: Transaction Log of Experiment 1				
0013994	2022-06-07 09:50:07.629751	120.000	Failed to send data	1
2.1.7 Transaction Log of Experiment 2				
0014007	2022-06-07 10:00:00.487482	120.000	Failed to send data	1
2.1.8: Transaction Log of Experiment 3				
0014023	2022-06-07 10:12:48.642737	120.000	Failed to send data	1
2.1.9: Transaction Log of Experiment 4				
0014032	2022-06-07 10:21:06.378240	120.000	Failed to send data	1
2.1.10: Transaction Log of Experiment 5				

2.2 Definitions

The entirety of the sender program is in the Python file **sender.py**. And to understand **sender.py**, we must first explore the many components of **sender.py**.

Imported Modules and Setup

The four imported modules are easy to understand. The module **hashlib** is used for checksum derivation, the module **re** is for **RegEx** operations used in parsing messages, **socket** for the UDP functionality, and **sys** for managing the inline arguments.

```
sender.py > ...
1  import hashlib
2  import re
3  import socket
4  import sys
5
6  DO_LOG = True
7
8  # PAYLOAD file name
9  PAYLOAD_FILENAME = "03383a1b.txt"
10
11 # THIS setup
12 MY_ID = "03383a1b"
13 MY_HOSTNAME = socket.gethostname()
14 MY_IP = socket.gethostbyname(MY_HOSTNAME)
15 MY_PORT = 6741
16
17 # OTHER setup
18 OTHER_IP = "10.0.7.141"
19 OTHER_PORT = 9000
20
21 # Setting up UDP socket
22 UDPsocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
23 UDPsocket.bind(('', MY_PORT))
```

2.2.1: Top section of **sender.py**

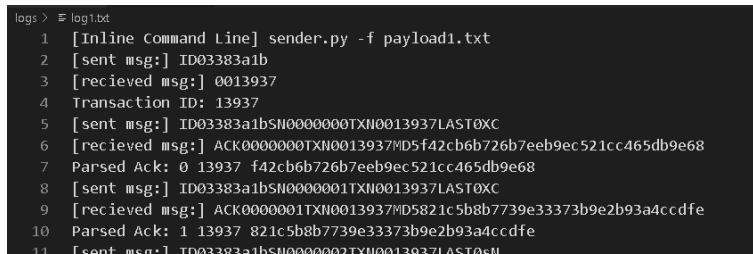
The four imported modules are easy to understand. The module `hashlib` is used for checksum derivation, the module `re` is for `RegEx` operations used in parsing messages, `socket` for the UDP functionality, and `sys` for managing the inline arguments.

The `DO_LOG` variable is simply a bool that tells if the sender program should log its real time information in the console log or do the entirety of the sender program functionality silently. A preview of what the recorded console log activity for Experiment 1 is shown in [image 2.2.2](#).

The `PAYLOAD_FILENAME` variable should be self-explanatory, as it is the name of the payload file used by `sender.py`.

The next declared variables are for establishing the information between the host sender and the receiver. The `MY_ID` variable stores the unique ID provided to each student through email. The `MY_IP` and `MY_PORT` variables hopefully describe themselves, being respectively the IP address and port used by the sender host. Meanwhile `OTHER_IP` and `OTHER_PORT` are the IP address and port used by the receiver. Note that most of the values here excluding the `MY_ID` and `MY_IP` are defined to the default values given in the project description.

Lastly there is the `UDPsocket` defined to be socket object that will be used by `sender.py` for performing its protocol functionality through UDP. Note that it is bound to `MY_PORT`.



```
logs > log1.txt
1 [Inline Command Line] sender.py -f payload1.txt
2 [sent msg:] ID03383a1b
3 [recieved msg:] 0013937
4 Transaction ID: 13937
5 [sent msg:] ID03383a1b5H0000000TXN0013937LAST0XC
6 [recieved msg:] ACK0000000TXN0013937MD5f42cb6b726b7eeb9ec521cc465db9e68
7 Parsed Ack: 0 13937 f42cb6b726b7eeb9ec521cc465db9e68
8 [sent msg:] ID03383a1b5H00000001TXN0013937LAST0XC
9 [recieved msg:] ACK00000001TXN0013937MD5821c5b8b7739e33373b9e2b93a4ccdfc
10 Parsed Ack: 1 13937 821c5b8b7739e33373b9e2b93a4ccdfc
11 [sent msg:] ID03383a1b5H00000002TXN0013937LAST0XC
```

2.2.2: Top section of the record of console logs during Experiment 1

2.3 Functions

Understanding `sender.py` is understanding its many helper functions, as plenty of its functionality lies in the said functions. In fact, what could be considered the main part of the code is actually just three lines.

UDP Socket Communication

```
# Sends an ascii msg to OTHER
def UDP_sendmsg(msg):
    if (DO_LOG):
        print("[sent msg:] " + msg)
        UDPsocket.sendto(msg.encode('ascii'), (OTHER_IP,OTHER_PORT))

# Recieving a UDP ascii msg
def UDP_getmsg():
    while True:
        data, addr = UDPsocket.recvfrom(1024)
        if len(data) > 0:
            msg = data.decode('ascii')
            if (DO_LOG):
                print("[recieved msg:] " + msg)
            return msg
```

2.2.3: UDP Socket Communication functions

The functions `UDP_sendmsg()` and `UDP_getmsg()` are the entirety of the UDP message transmission and reception functionalities within function wrappers. This can be done as the information about the sender and receiver's IP addresses and ports are well defined and static throughout the runtime, and that everything uses purely ASCII encoding. Note that both functions log packet information if `DO_LOG` is `True`.

Content Management

```
# Opens and returns all contents of PAYLOAD_FILENAME
def readFile():
    with open(PAYLOAD_FILENAME,encoding='ascii') as f:
        contents = f.read()
        return contents

# Splits a msg string by i characters, and returns the split string as a tuple
def splitString(msg, i):
    if i >= len(msg):
        return (msg,'')
    if i <= 0:
        return ('',msg)

    first = msg[:i]
    second = msg[i-len(msg):]

    return (first,second)

# Encodes the payload and parameters to the given message format
def encodePayload(seqNo, transaction, isLast, payload):
    Z = '1' if isLast else '0'
    msg = 'ID' + MY_ID + 'SN' + str(seqNo).zfill(7) + 'TXN' + str(transaction).zfill(7) + 'LAST' + Z + payload
    return msg
```

2.2.4: Content Management functions

Next functions are used for managing the contents that will be sent to the receiver. The `readFile()` function is used to extract the entire message to be sent to the receiver from the payload file. The `splitString()` is then used for the segmentation the message. Lastly, the `encodePayload()` function is what makes the message being passed compliant to the provided protocol requirements.

ACK Handling

```
# Gets checksum of the string msg
def getChecksum(msg):
    return hashlib.md5(msg.encode('utf-8')).hexdigest()

# Parses an ACK message for information using RegEx
def parseAck(msg):
    SN = int(re.search(r"ACK.{7}",msg).group()[3:])
    TXN = int(re.search(r"TXN.{7}",msg).group()[3:])
    CHECKSUM = re.search(r"MD5.*",msg).group()[3:]

    if (DO_LOG):
        print("Parsed Ack: ",end='')
        print(SN,TXN,CHECKSUM)

    return (SN, TXN, CHECKSUM)
```

2.2.5: ACK Handling functions

Next functions are used for managing the contents that is used to process the ACK messages from the receiver. The `getChecksum()` function is the wrapper to MD5 hashing as provided from the project specifications. The `parseAck()` function is then what extracts the information from the raw ACK messages through `RegEx` operations from the `re` module.

Minimum Sender Functionality

```
# Initiates transaction and returns the transaction ID
def startTransaction():
    intentMessage = "ID" + MY_ID
    UDP_sendmsg(intentMessage)
    transaction = int(UDP_getmsg())
    if (DO_LOG):
        print("Transaction ID: " + str(transaction))
    return transaction
```

2.2.6: `startTransaction()` function

The `startTransaction()` function is what handles the initialization of the communication between the sender and the receiver. The function first sends an **Intent Message** to the receiver using the provided unique ID, then waits for the **Accept Message** reply which the function will extract the transaction ID, from the **Accept Message** packet and return.

```

# Basic automated sending of the contents
# Segments contents by equally sized packets
# Waits for Ack before sending next packet
def ConstantSizeSend_WaitForAck(contents,transaction,bytes):
    seqNo = 0
    while(contents):
        # Split the defined size of characters from the contents
        payload, contents = splitString(contents,bytes*2)

        # Check if the segmentation is the last
        isLast = (contents=='')

        # Encode the segmentation to a packet as per the parameters
        encodedMsg = encodePayload(seqNo,transaction,isLast,payload)

        # Sending the UDP message
        UDP_sendmsg(encodedMsg)

        # Recieving the ACK message
        ack = UDP_getmsg()
        SN, TXN, CHECKSUM = parseAck(ack)

        # Increase the sequence number
        seqNo+=bytes

```

2.2.7: ConstantSizeSend_WaitForAck() function

The `ConstantSizeSend_WaitForAck()` function is what supplies the aforementioned functionality of `sender.py` to be a working sender. It simply divides the payload with a constant size of bytes, before sending each of them to the receiver. It also waits for an ACK message before sending the next packet. Note that there's no such thing as error handling or timeout values, (the parsed ACK information is unused) so this functionality is not only inefficient but also very unreliable due to its vulnerability to transmission errors.

2.4 Main Code

With most of the helper functions explored, it should be clear how sender.py works by now. So here we see how the many helper functions are used.

Command Line Argument Processing

```
# Processing Inline Arguments
if (DO_LOG):
    print('[Inline Command Line]',end=' ')
    for i in sys.argv:
        print(i,end=' ')
    print('')

if '-f' in sys.argv:
    idx = sys.argv.index('-f') + 1
    PAYLOAD_FILENAME = sys.argv[idx]

if '-a' in sys.argv:
    idx = sys.argv.index('-a') + 1
    OTHER_IP = sys.argv[idx]

if '-s' in sys.argv:
    idx = sys.argv.index('-s') + 1
    OTHER_PORT = int(sys.argv[idx])

if '-c' in sys.argv:
    idx = sys.argv.index('-c') + 1
    MY_PORT = int(sys.argv[idx])

if '-i' in sys.argv:
    idx = sys.argv.index('-i') + 1
    MY_ID = sys.argv[idx]
```

2.2.7: Part of the main code that handles the in line arguments

This part of the code parses the command line arguments for flags using the `in` keyword, and upon finding them would update the corresponding defined variable with the passed values.

Main Proper

```
contents = readFile()
transaction = startTransaction()
ConstantSizeSend_WaitForAck(contents,transaction,1)
```

This is the core of how `sender.py` works. It first reads the contents of the payload file, initiates the transaction and stores the transaction ID, before finally then running the sender functionality.