COMPUTER NETWORKING ASSIGNMENT 1

Jonathan Kelsi & Amit Moshcovitz

NOVEMBER 24, 2022

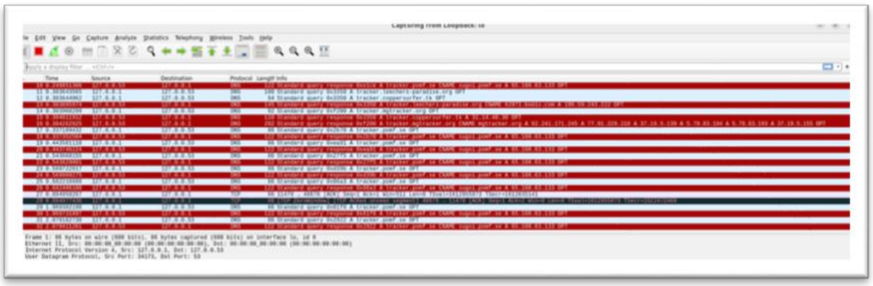
BAR ILAN UNIVERSITY

Computer Networking - 89535001

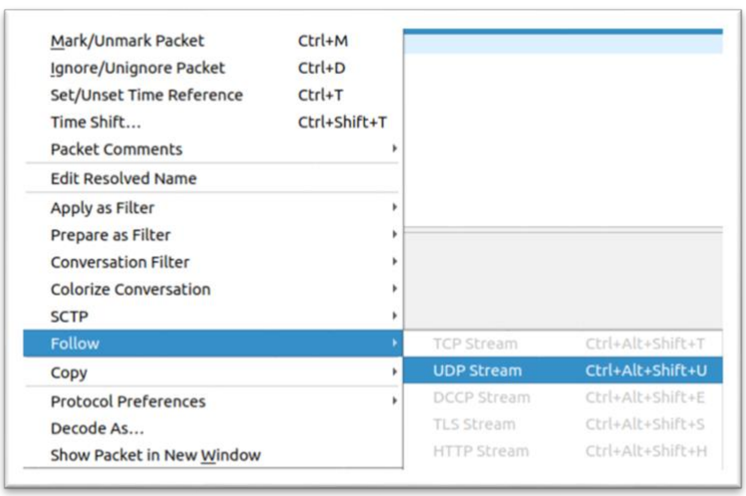
Part 1

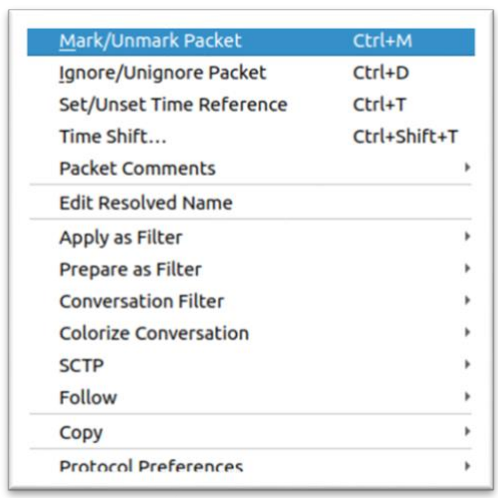
1.1)
We ran both the client and the server and saw they managed to communicate with one another:

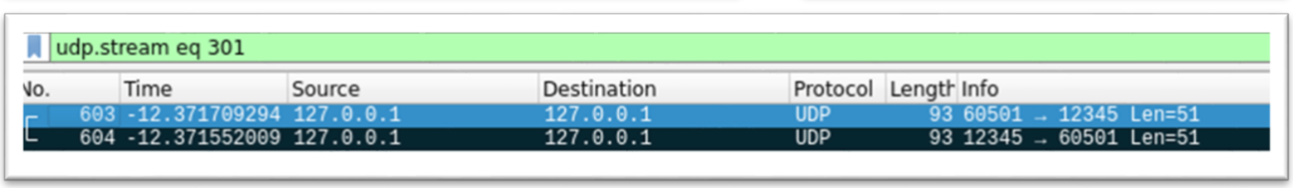
At the same we ran Wireshark and sniffed the traffic:



1.2)
We followed the UDP steam and marked the packets so we would be able to isolate and export them:







Note: We could have also used the "udp.stream" filter right away as shown above.

1.3)
A port number is a unique identifier assigned to a connection endpoint, to direct data to a specific service. In other words, we use a port number in order to differentiate between processes running on a machine.

In the code (when creating the socket):

```
# bind a name to the program
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', int(sys.argv[1])))
```

In this part, we've given our server the port number 12345. Using it, our client was able to locate the server on the machine we ran both of them on, and the server was able to locate the client.

From WireShark:

"Frame 1: 93 bytes on wire (744 bits), 93 bytes captured (744 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, **Src Port: 46209, Dst Port: 12345**
Data (51 bytes)"

"Frame 2: 93 bytes on wire (744 bits), 93 bytes captured (744 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, **Src Port: 12345, Dst Port: 46209**
Data (51 bytes)"

As we can see, the frames captured in the trace specify the source and destination ports.
The information regrading the ports is specified in the Transport layer.

1.4)
From the WireShark captures, the packets were sent from the IP address 127.0.0.1 to 127.0.0.1.

Running the ifconfig command we get:

"lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 1210515 bytes 5165996539 (5.1 GB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 1210515 bytes 5165996539 (5.1 GB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 172.18.70.40 netmask 255.255.252.0 broadcast 172.18.71.255
inet6 fe80::4700:6f39:e4ee:43e4 prefixlen 64 scopeid 0x20<link>
ether dc:e9:94:8f:6a:13 txqueuelen 1000 (Ethernet)
RX packets 18060089 bytes 19641777303 (19.6 GB)
RX errors 0 dropped 382 overruns 0 frame 0
TX packets 6015343 bytes 1077931250 (1.0 GB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0"

We can tell our virtual IP address is indeed 127.0.0.1, but the IP address that correspond with our Network Card is 127.18.70.40.

Part 2

We started with running the server and three clients and sent the same massages as the ones in the appendix. Similarly to part 1, we isolated the relevant packets from the trace and saved them. The result:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	49	32999 → 12345 Len=7
2	0.000611528	127.0.0.1	127.0.0.1	UDP	43	12345 → 32999 Len=1
3	2.146714571	127.0.0.1	127.0.0.1	UDP	47	46729 → 12345 Len=5
4	2.146884704	127.0.0.1	127.0.0.1	UDP	43	12345 → 46729 Len=1
5	2.146905796	127.0.0.1	127.0.0.1	UDP	47	12345 → 46729 Len=5
6	6.043189601	127.0.0.1	127.0.0.1	UDP	52	46729 → 12345 Len=10
7	6.043374750	127.0.0.1	127.0.0.1	UDP	43	12345 → 46729 Len=1
8	13.981379654	127.0.0.1	127.0.0.1	UDP	51	58123 → 12345 Len=9
9	13.981565641	127.0.0.1	127.0.0.1	UDP	43	12345 → 58123 Len=1
10	13.981586174	127.0.0.1	127.0.0.1	UDP	52	12345 → 58123 Len=10
11	20.999393884	127.0.0.1	127.0.0.1	UDP	49	58123 → 12345 Len=7
12	20.999579382	127.0.0.1	127.0.0.1	UDP	43	12345 → 58123 Len=1
13	26.531133670	127.0.0.1	127.0.0.1	UDP	53	58123 → 12345 Len=11
14	26.531285085	127.0.0.1	127.0.0.1	UDP	43	12345 → 58123 Len=1
15	29.921973372	127.0.0.1	127.0.0.1	UDP	43	32999 → 12345 Len=1
16	29.922153422	127.0.0.1	127.0.0.1	UDP	43	12345 → 32999 Len=1
17	29.922168158	127.0.0.1	127.0.0.1	UDP	56	12345 → 32999 Len=14
18	29.922177796	127.0.0.1	127.0.0.1	UDP	55	12345 → 32999 Len=13
19	29.922187225	127.0.0.1	127.0.0.1	UDP	60	12345 → 32999 Len=18
20	29.922196304	127.0.0.1	127.0.0.1	UDP	56	12345 → 32999 Len=14
21	29.922204965	127.0.0.1	127.0.0.1	UDP	60	12345 → 32999 Len=18
22	43.322117130	127.0.0.1	127.0.0.1	UDP	51	32999 → 12345 Len=9

As requested, now we will present three packets and explain how they correspond to the code we wrote:

1) The first packet we chose:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	49	32999 → 12345 Len=7

```
Frame 1: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 32999, Dst Port: 12345
Data (7 bytes)
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 23 a2 82 40 00 40 11  9a 45 7f 00 00 01 7f 00   ·#··@·@· ·E······
0020  00 01 80 e7 30 39 00 0f  fe 22 31 20 41 6c 69 63   ····09·· ·"1 Alic
0030  65                                                 e
```

When Alice enters the group, she sends a join request to the server – “1 Alice”.

The relevant code segment:

```
message = input()
s.sendto(message.encode(), (server_ip, server_port))
```

In Wireshark we can indeed see that the message is part of the frame in the data layer. In the transport layer we can see the source and destination ports.
Alice's port (the source port): 32999. The server's port (the destination port): 12345.
In the Network layer we can see both the source and destination IP are 127.0.0.1 because we ran the programs on the same machine. In addition, we can see that no MAC address is in use because the frame never leaves the machine.

2) The second packet we chose:

```
15  29.921973372  127.0.0.1    127.0.0.1    UDP    43  32999 → 12345 Len=1
16  29.922153422  127.0.0.1    127.0.0.1    UDP    43  12345 → 32999 Len=1
```

```
Frame 15: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 32999, Dst Port: 12345
Data (1 byte)
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 1d a9 bd 40 00 40 11  93 10 7f 00 00 01 7f 00   ····@·@· ········
0020  00 01 80 e7 30 39 00 09  fe 1c 35                  ····09·· ··5
```

When Alice wants an update, she sends an update request to the server – "5".
The relevant code segment is the same. All the other information is identical since the sender and the receiver remain the same. The only difference is the message the client sends – means the data in the application layer.

3) Last but not least, the third and final packet we chose:

```
15 29.921973372  127.0.0.1   127.0.0.1   UDP   43 32999 → 12345 Len=1
16 29.922153422  127.0.0.1   127.0.0.1   UDP   43 12345 → 32999 Len=1
17 29.922168158  127.0.0.1   127.0.0.1   UDP   56 12345 → 32999 Len=14
```

```
Frame 16: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 12345, Dst Port: 32999
Data (1 byte)
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 1d a9 be 40 00 40 11  93 0f 7f 00 00 01 7f 00   ····@·@· ········
0020  00 01 30 39 80 e7 00 09  fe 1c 35                  ··09···· ··5
```

The relevant code segment:

```python
def send_data(user_addr, n, lst):
    s.sendto(str(n).encode(), user_addr)

    for m in lst:
        s.sendto(m.encode(), user_addr)
```

At first glance, it might seem exactly like the previous packet. The only difference between the two is the source and destination - In fact, they are the exact opposite. The reason the data is the same is our implementation. Whenever the server intends to send messages to the client is specifies how many massages it is going to send.

Note: It was pure luck that the data is “5” as before.