Computer Networks
Assignment 1

Students Jaskirat Singh Maskeen (23110146) & Karan Sagar Gandhi (23110157)
Professor Sameer G. Kulkarni
GitHub Repository (Contains instructions to run) https://github.com/jsmaskeen/CS331-Assignment1

1 DNS Resolver

The objective of this task was to develop a custom DNS resolution system. The system consists of a client that parses DNS queries from a PCAP file and a server that resolves these queries based on a set of custom rules.

1.1 DNS Packet Structure

We read the RFC 1035 [5], and the packet structure is summarized in the figure below (following that we have given an example).

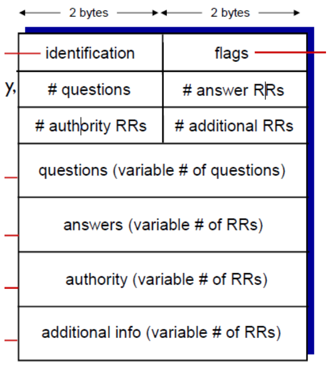

Figure 1: Packet structure from slides [3].

Then we write our custom parser, which parses all the fields from the DNS packet. This is done so that, we can send the response as a DNS frame with the Answer field attached. The meaning of fields can be read from the RFC [5] (Table 1), however the main part we had to implement was decompression of the labels.

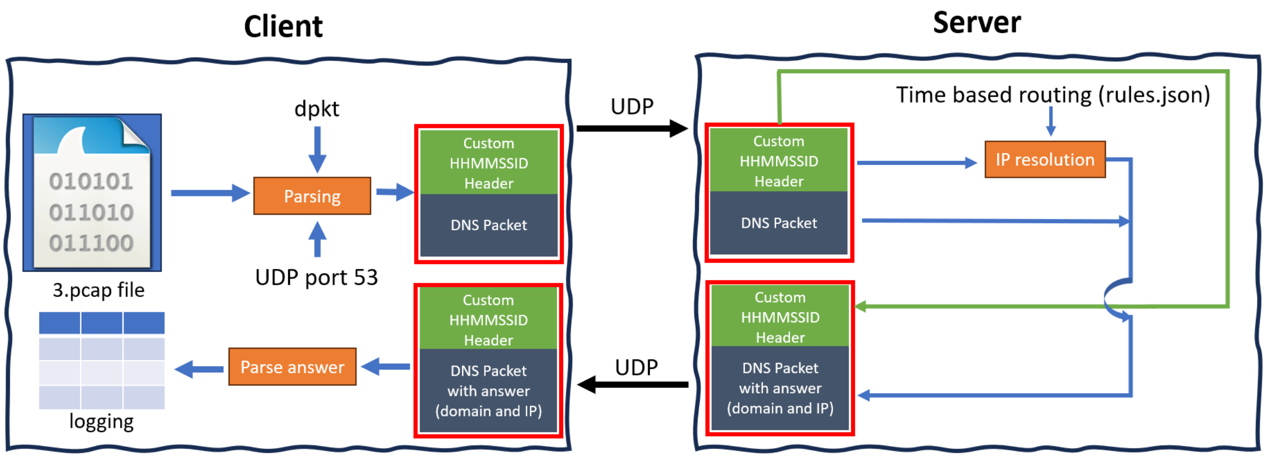

Figure 2: Overview of our workflow for Task 1

Section	Field Name	Description
Header Section (12 Bytes)		
Header	ID (Transaction ID)	16-bit identifier to match queries with responses.
Header	Flags	A 16-bit field containing: **QR** (1), **Opcode** (4), **AA** (1), **TC** (1), **RD** (1), **RA** (1), **Z** (3), **RCODE** (4).
Header	QDCOUNT	16-bit field specifying the number of questions.
Header	ANCOUNT	16-bit field specifying the number of answer records.
Header	NSCOUNT	16-bit field specifying the number of authority records.
Header	ARCOUNT	16-bit field specifying the number of additional records.
Data Sections (Variable Length)		
Question	QNAME, QTYPE, QCLASS	Contains the actual question for the name server.
Answer	Resource Records	Contains resource records that directly answer the query.
Authority	Resource Records	Lists authoritative name servers for the queried domain.
Additional	Resource Records	Provides extra information related to the query.

Table 1: A summary of the DNS Packet Structure.

Strings which appear more than once are stored as a pointer and a offset, after their first appearance. For example:

Suppose our question is `A` (IPv4) record for `iitgn.ac.in`, and the answer will be `CNAME` pointing to `iitgn.ac.in`. So to save space, the answer will reuse the bytes of `iitgn.ac.in` which would be present in the question itself.

```
Header (12 bytes):
  TxnID   Flags
  1A 2B   81 80
   QD      AN
  00 01   00 01
   NS      AR
  00 00   00 00
             ^
            0x0C
```

Question section (starts at offset 0x0C = 12, which is the size of the header): This section is stored in a length-labelled format. The 03 before www (77 77 77) means "this label is 3 bytes long")

```
          0x10
            v
  03 77 77 77 05 69 69 74 67 6E 02 61 63 02 69 6E 00
  [3] w  w  w [5] i  i  t  g  n [2] a  c [2] i  n  (terminated by 00)
  00 01     (QTYPE = A)
  00 01     (QCLASS = IN)

Answer section:
  C0 0C     (NAME = pointer to offset 0x0C -> "www.iitgn.ac.in")
            (C0 means 1100 0000, which means that this is a pointer)
  00 05     (TYPE = CNAME)
  00 01     (CLASS = IN)
  00 00 00 3C (TTL = 60)
  00 02     (RDLENGTH = 2 bytes)
  C0 10     (RDATA = pointer to offset 0x10 -> "iitgn.ac.in")
            (C0 means 1100 0000, which means that this is a pointer)
```

1.2 System Architecture and Flow

The operational flow of the system is as follows:

1. **Packet Filtering:** The client begins by reading a given `.pcap` file and filtering it to isolate the DNS query packets (these are the queries which are sent to UDP port 53).

2. **Custom Header Addition:** For each DNS query, the client generates an 8-byte custom header with the format `"HHMMSSID"`. This header contains the current time and a two-digit sequence ID for the query (The sequence ID is incremented after each query).

3. **Communication:** The client sends the original DNS query, prefixed with this custom header, to the server.

4. **Server-Side Processing:** The server receives the message, parses the custom header to determine the appropriate IP pool based on the timestamp, and extracts the domain name from the DNS query payload (following [5]).

5. **Response and Logging:** The server sends the resolved IP address, the domain name, and the original custom header back to the client in a DNS frame using UDP. The client then logs this information. To better simulate real-world conditions, we allow the client introduces an artificial delay (random, configurable) between sending DNS queries.

6. **Visualization:** For the last DNS packet sent, we print the DNS frame (without the custom header) in human readable format. To better see what the client exactly sends to the server, and what the server responds with.

1.3 Transport Protocol: TCP to UDP

1.3.1 Initial TCP Implementation

Our initial implementation for the client-server communication was built using TCP (`SOCK_STREAM`). However, TCP being a stream protocol, without message boundaries, we had to implement our own mechanism to identify the start of the message. This was done by prefixing each payload with a 4-byte unsigned integer which represents the payload's length. The receiver (client or server) would first read these 4 bytes to determine the message size and then read that exact number of bytes to get the complete message.

1.3.2 Current UDP Implementation

After a discussion with the professor, we decided to use UDP (`SOCK_DGRAM`) as our communication protocol. This change was motivated by the fact that real-world DNS queries majorly use UDP due to its low overhead (No handshakes, unlike TCP). However it is to be noted that incase size of the message is more than 512 bytes, TCP will be used [5], but we stick to UDP in our implementation.

UDP is a message-oriented protocol, meaning it preserves message boundaries automatically. Hence this allowed us to remove the manual 4-byte length prefixing, simplifying our message handling logic.

1.4 Implementation Details

- `3.pcap`: The pcap file from where we process the DNS queries. ($146 + 157 \equiv 3 \pmod{10}$)
- `client.py`: Manages reading the PCAP file, sending queries, and displaying results.
- `server.py`: Listens for incoming queries, applies the routing logic, and sends back responses.
- `helpers.py`: Contains utility functions for DNS packet parsing, including logic to handle domain name decompression as specified in [5].
- `rules.json`: An external configuration file that defines the time-based routing rules, allowing for easy modification without changing the server code.

1.5 Results

The client successfully processed the DNS queries from `3.pcap` and received the resolved IP addresses from the server. The final output is shown in the Table 2.
Note that we ran this at `Tuesday 09/09/2025 22:32:20`.

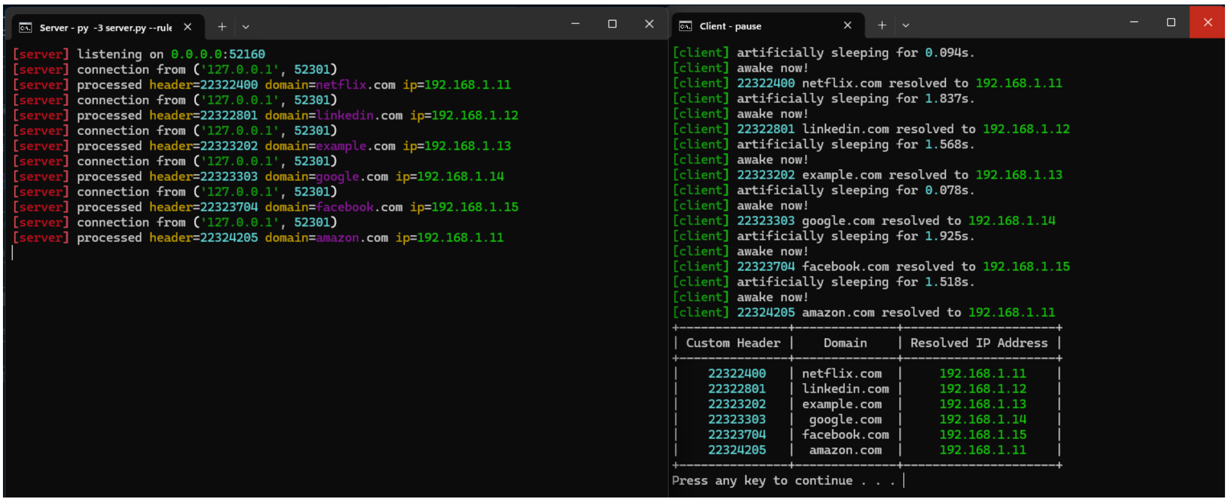

Figure 3: Screenshot of the client and server running.

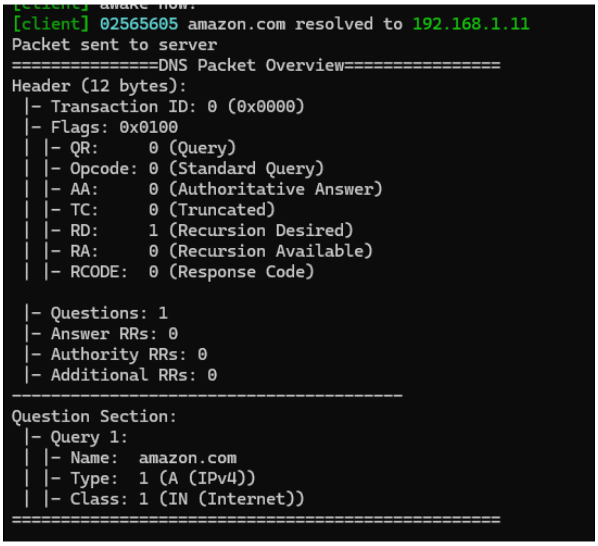

Figure 4: DNS frame that was sent.

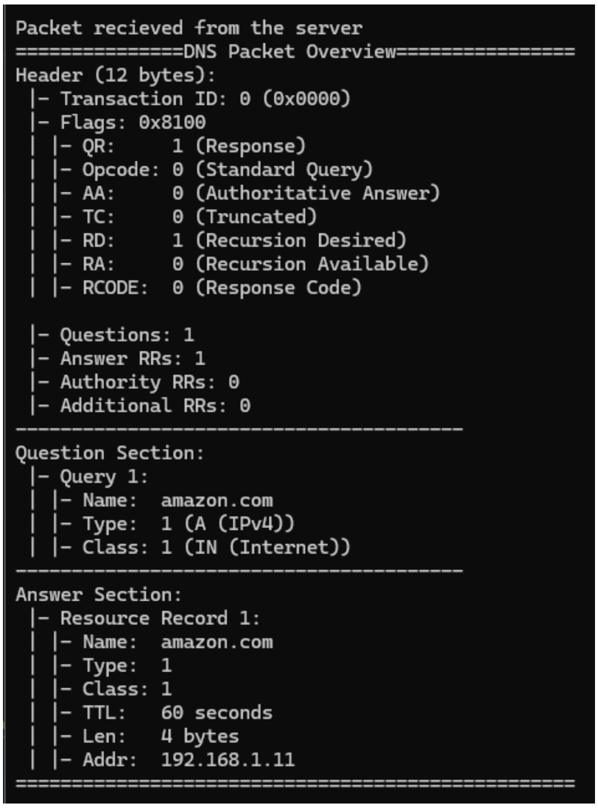

Figure 5: DNS frame that was recieved.

Custom Header	Domain	Resolved IP Address
22322400	netflix.com	192.168.1.11
22322801	linkedin.com	192.168.1.12
22323202	example.com	192.168.1.13
22323303	google.com	192.168.1.14
22323704	facebook.com	192.168.1.15
22324205	amazon.com	192.168.1.11

Table 2: Resolved domain name and IP Address with custom header.

1.6 Caveats

1. For the custom header (in `HHMMSS`), we use the system's current time obtained via `datetime.datetime.now()`. While pcap files record the actual request/response times, these timestamps are metadata added by the capturing tool (such as wireshark) and are not part of the packet bytes themselves. Since the DNS packets inherently do not carry the timing information, we take the assumption of using the current time for the `HHMMSS` of the header.

2. We use `dpkt` solely to read the pcap file and extract the DNS packets. Beyond this step, `dpkt` is not used; instead, the parsing of DNS packets is handled by our custom parser.

3. We return the resultant IP address determined by our time based routing rules. This value does not necessarily correspond to the actual resolved IP address for the domain in the DNS packet. If the true resolution is required, our server can forward the DNS query to a public resolver (UDP 8.8.8.8, port 53), obtain the resolved IP address, and include it in the answer fields alongside the rule based IP.

2 traceroute/tracert Protocol Behaviour

Traceroute is a network diagnostic tool used to trace the path that data packets take from our computer (the source) to a destination host (like a website or server).

It helps identify the "route", which packets follow, through the routers on the Internet and measures how long each "hop" (step) along the way takes.

The way it works is that it sends multiple packets or probes with an increasing TTL (Time To Live) field, which basically tells us how many hops the packet is valid for. When the probe fails to reach the destination it sends another packet back to the destination computer saying that it failed to reach the destination (ICMP "Time Exceeded" [4]). This packet contains the IP address of the last router. Hence, by sending packets with increasing TTL, we can find out the path that the packet takes to reach the destination.

2.1 What protocol do Windows and Linux use by default?

By default, the Windows Tracert tool uses ICMP (Internet Control Message Protocol) packets as probes to find out the path that the packet takes to reach the destination. As you can see in the "`tracert windows discord no block.pcap`" file, which stores the captured packets when we ran the tracert tool for `www.discord.com`. A lot of ICMP packets or probes are sent from our laptop with an increasing TTL.

The response packets which the router sends back when the time exceeds are also ICMP packets.

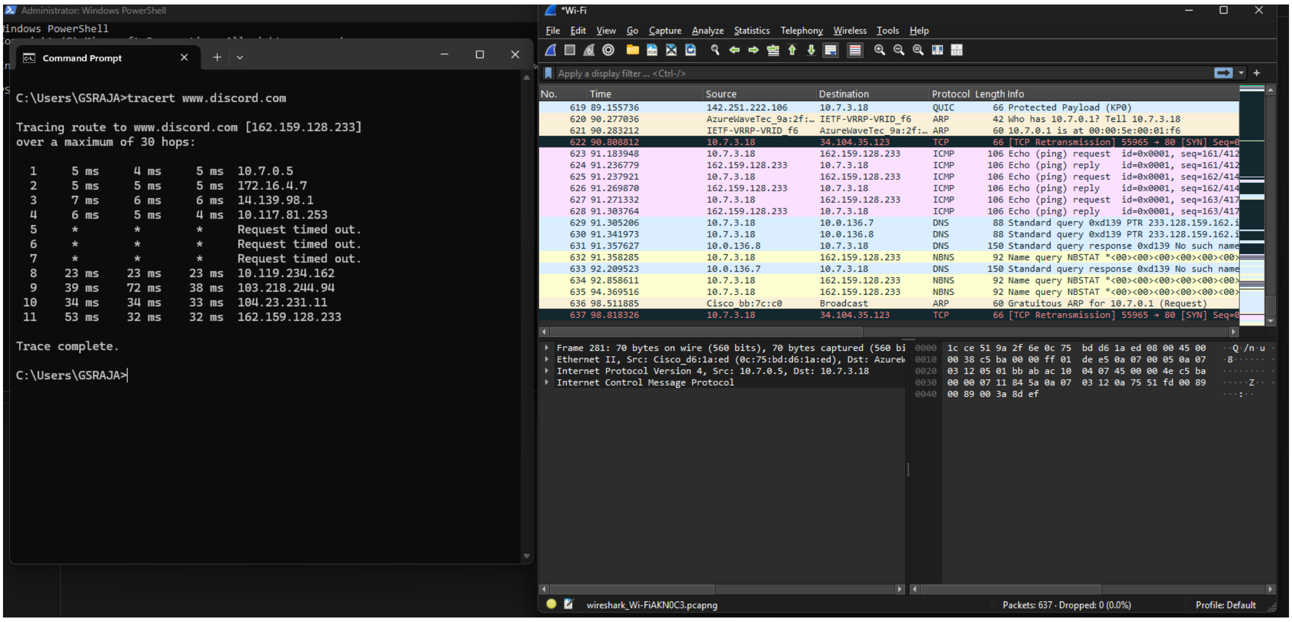

Figure 6: Default behaviour of tracert on Windows

By default, the Linux Traceroute tool uses UDP packets to probe and find the path that the packet takes to reach the destination. We can also see this from the `traceroute discord linux no block.pcap` file, which stores the captured packets when we ran the traceroute tool for `www.discord.com` on Linux. However, the response which we get from the router is still an ICMP packet.

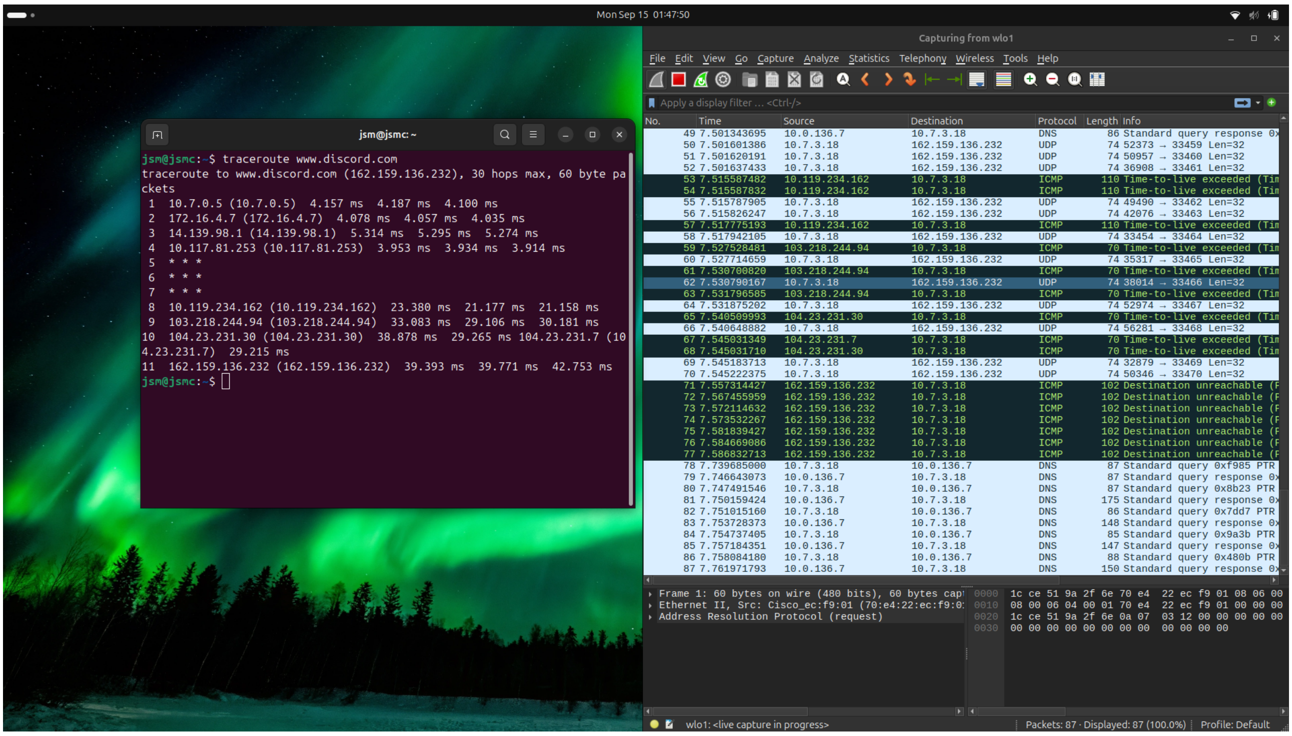

Figure 7: Default behaviour of traceroute on Linux

2.2 Why do we get ***? What are the two reasons why a router might not reply?

The three columns which we see in the tracert/traceroute command are the RTT (Round-Trip Time) for three probes with the same TTL. When some or all of the probes for a given TTL don't receive a reply, that is indicated by a * in place of the RTT. There are two reasons why a router might not reply:

1. The router sends back an ICMP "Time Exceeded" packet when the TTL expires, but this ICMP packet might be dropped when it tries to reach the client back.
2. It could be the case that the router is configured not to answer traceroute probes because of policy/firewall. It might have rate-limits or deprioritise the probe replies. The router might forward transit traffic, but is set to not send ICMP “Time Exceeded” or other probe replies. This may be done to avoid clogging the router, especially when it is busy.

2.3 In Linux traceroute, which field in the probe packets changes between successive probes sent to the destination

Between sucessive probe packets, there are two fields that might differ in packets which are sent by the Linux traceroute tool: the destination port and the Time to Live (TTL). There are multiple probes (default 3) with the same TTL sent. But these probes are sent to different ports. As we can see from the image below, Traceroute starts with a base destination port like 33434 and increments it for each successive probe. So the first probe will have a port of 33434, then the next probe will have a port of 33435, and so on. This can be done because the routers don't care about the final destination port when they are sending the packets. At the final destination, the host sees a UDP packet addressed to some unused high port, and replies with ICMP "Port Unreachable". By matching replies to the unique destination port, traceroute can identify which probe each ICMP message corresponds to. This is because Linux, by default, uses UDP packets as probes; Windows ICMP packets already have a sequence number field in the header to identify the packet and match it with the packet that it is sending.

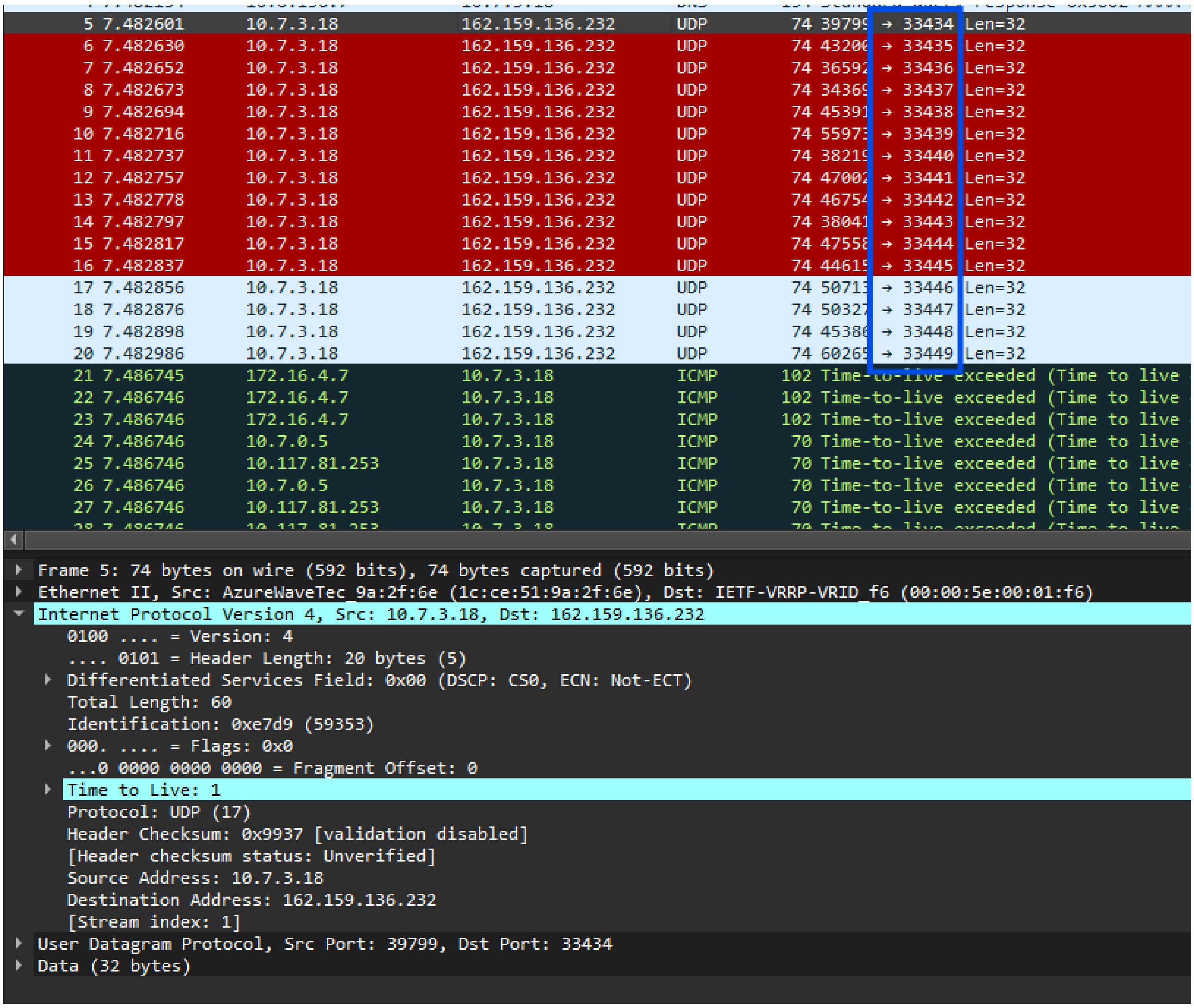

Figure 8: Screenshot of the port differing between the successive probe packets

2.4 At the final hop, how is the response different compared to the intermediary hop ?

At the final hop, the response is different because the packet successfully reaches its destination. Instead of a router sending an ICMP Time Exceeded message, the destination host itself sends a different type of response. This difference is what signals to the traceroute or tracert utility that the path has been successfully traced to its end. This packet that is returned is different for both Linux and Windows.

2.4.1 Linux traceroute

For intermediary hops, traceroute sends a probe with a Time-To-Live (TTL) value that expires at an intermediate router. The router, seeing the TTL reach zero, discards the packet and sends an ICMP "Time Exceeded" message back to the source. This Time Exceeded message also contains the IP address of the last router, which we can use to figure out the path which the packet takes to reach the destination.

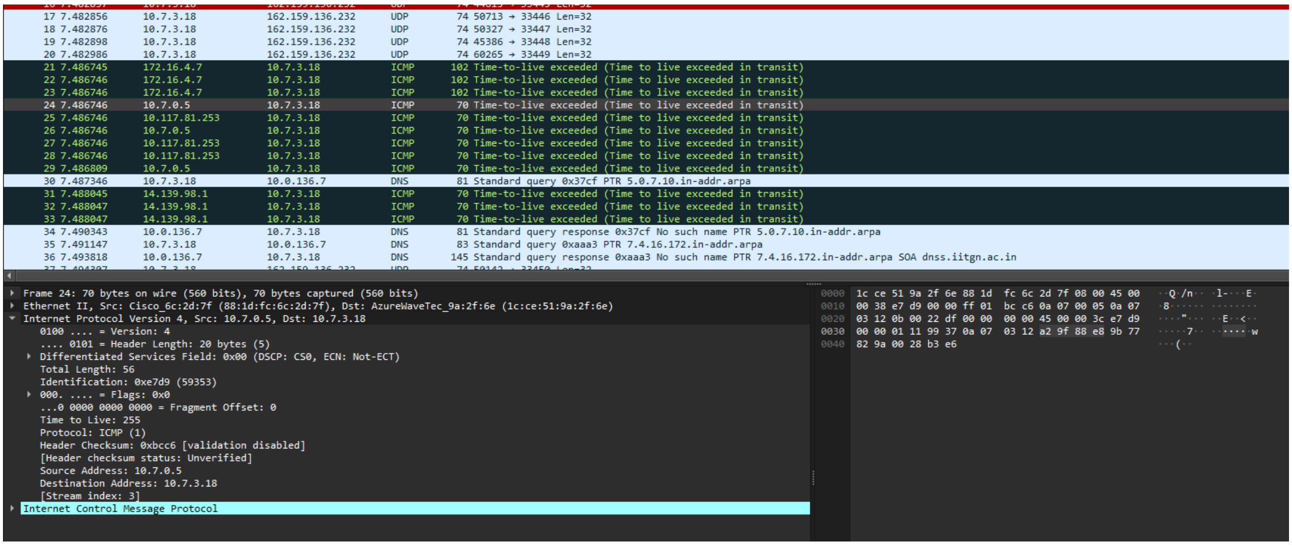

Figure 9: Time to live exceeded packet received in Linux

For the final hop, the last probe packet has a high enough TTL to reach the destination host. By default, Linux traceroute uses UDP packets with a destination port that's very unlikely to be in use (starting at 33434). When the packet arrives at the destination, the host's operating system sees that no application is listening on that specific UDP port. According to network protocol rules, it then generates and sends back an ICMP "Port Unreachable" message. This message confirms the packet reached the host and, since it's a different ICMP message from a Time Exceeded one, traceroute knows that the packet reached the destination.

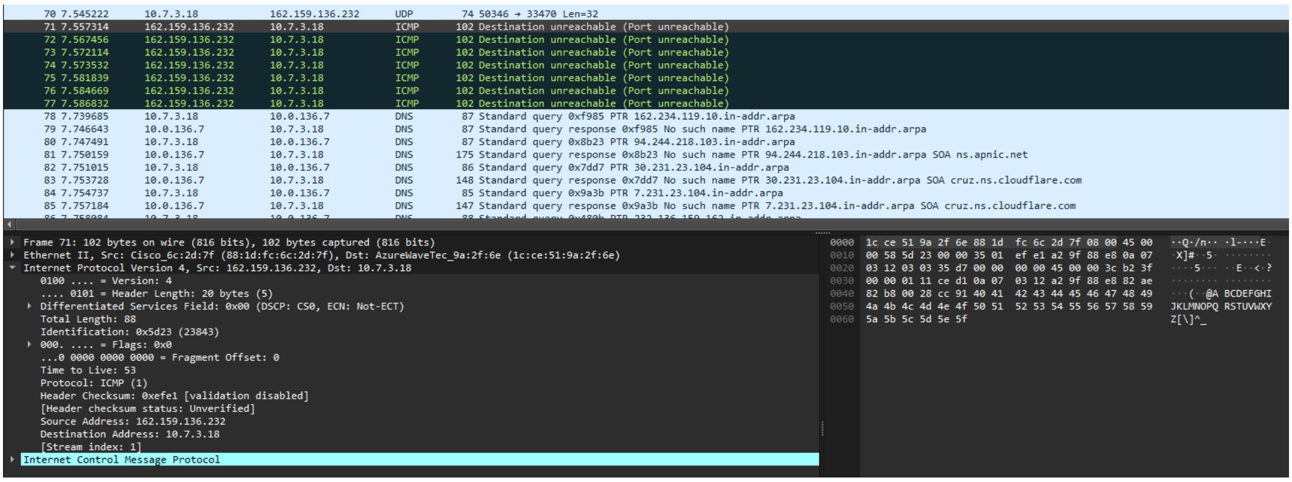

Figure 10: "Port Unreachable" message in Linux

2.4.2 Windows tracert

For intermediary hops, tracert uses ICMP "Echo Request" packets. As with traceroute, an intermediary router's TTL expires, and it sends back an ICMP "Time Exceeded" message.

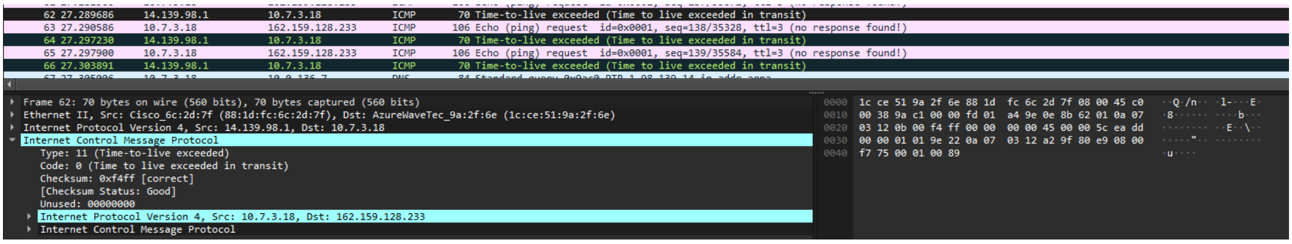

Figure 11: Time to live exceeded in Windows

The final probe packet, being an ICMP "Echo Request", reaches the destination host. The host replies with an ICMP "Echo Reply" message. tracert recognizes this different ICMP message as the sign that the destination was reached and stops the trace.

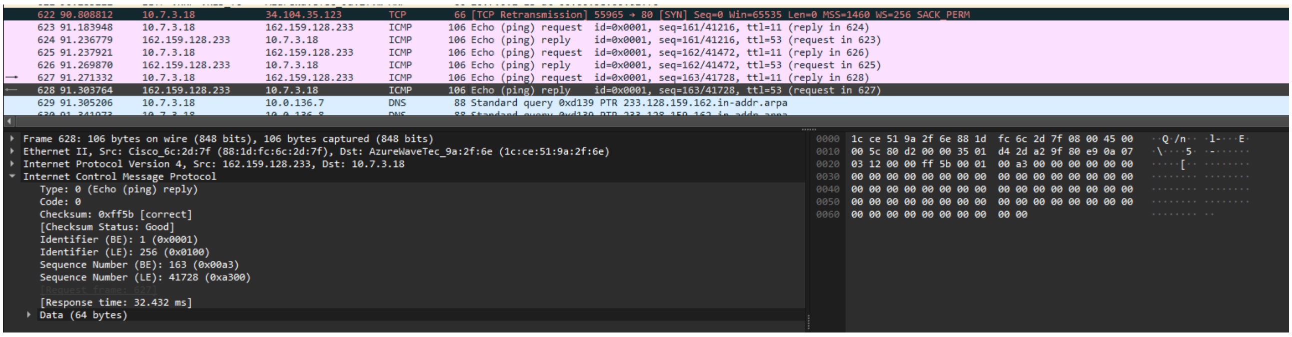

Figure 12: "Echo Reply" message sent back in Windows

2.5 Suppose a firewall blocks all UDP traffic but allows ICMP, how would this affect the results of Linux traceroute vs Windows tracert?

If in Linux, in some part of the network, UDP traffic is blocked by some firewall, then we will get a * * * in the traceroute request for all routers after that part of the network. We might also not be able to reach the destination. But it will not affect ICMP since ICMP doesn't use UDP and is directly enclosed in an IP packet.

We can see how blocking UDP traffic at the client affect traceroute/tracert behaviour:
In Linux, the operation completely fails since sending UDP packets is not allowed (as seen in the figure below).

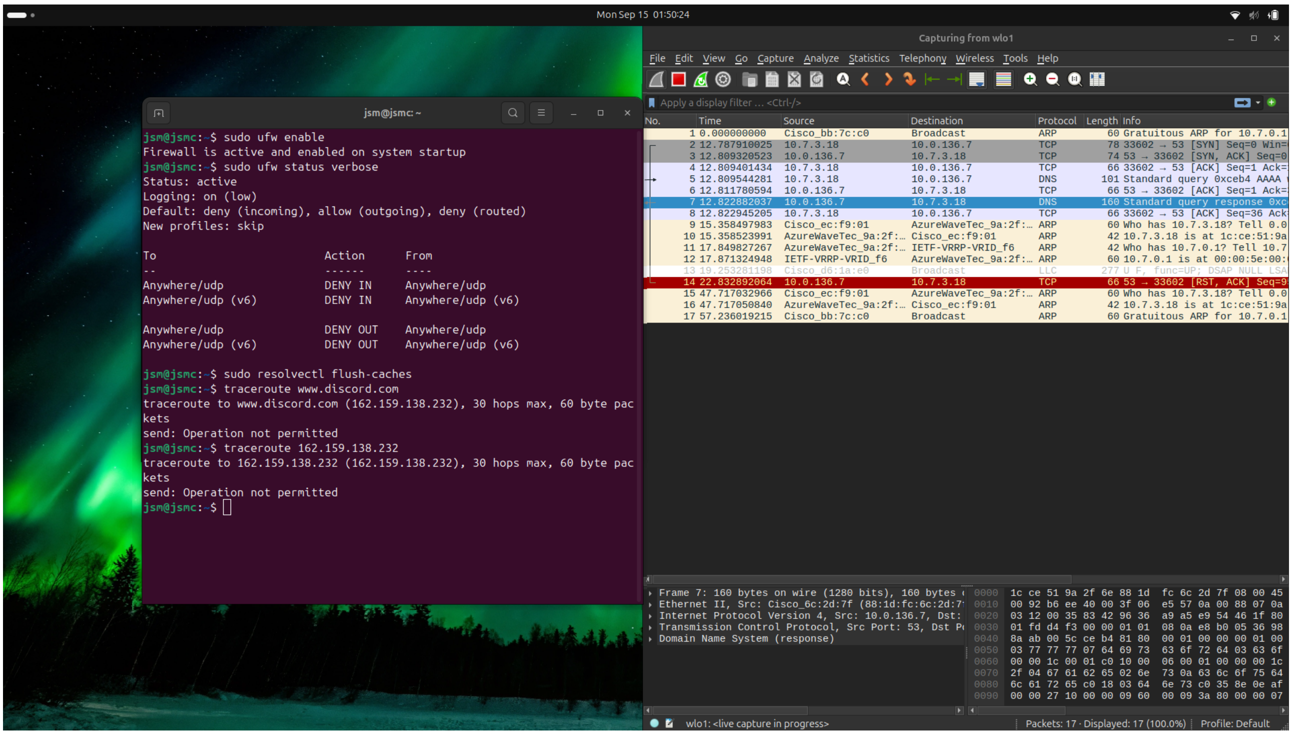

Figure 13: Running traceroute in Linux after blocking all UDP connections (both incoming and outgoing)

In Windows, since tracert by default uses ICMP packets, enclosed within an IP frame, blocking all the Inbound and outbound UDP connections doesn't make any difference (as seen in the figure below)

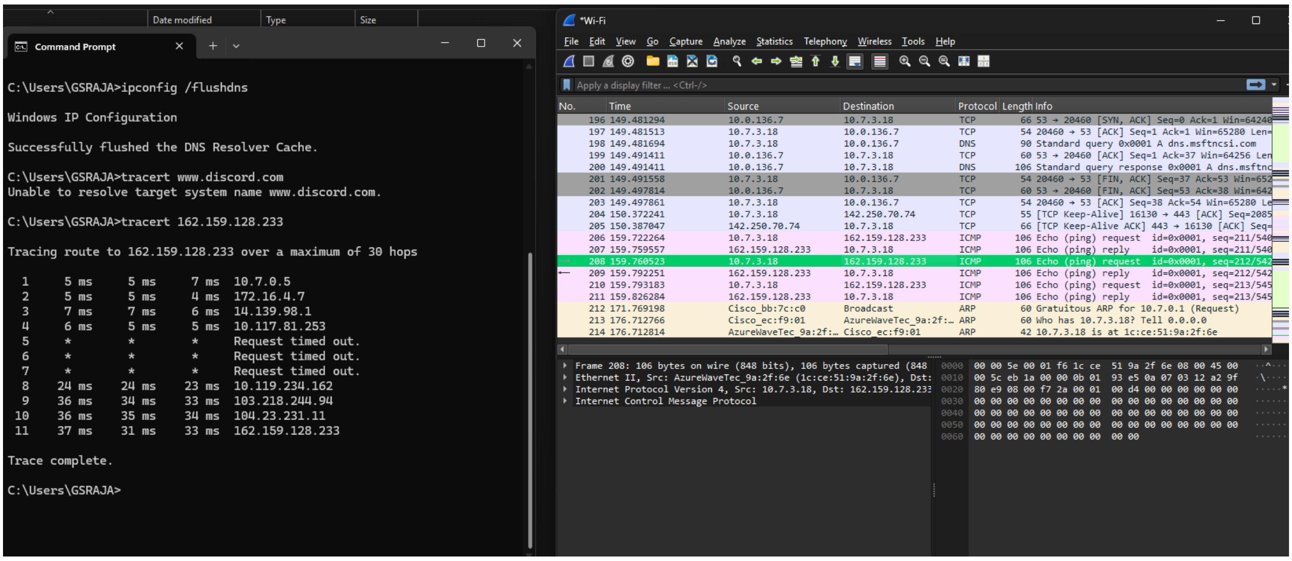

Figure 14: Tracert in Windows after blocking all UDP connections

Note that in Windows, we have to provide the IP address instead of the domain name since DNS resolution doesn't work (UDP traffic is blocked, and it doesn't try TCP). It works in Linux, as DNS will try to use TCP (See Fig. 13).

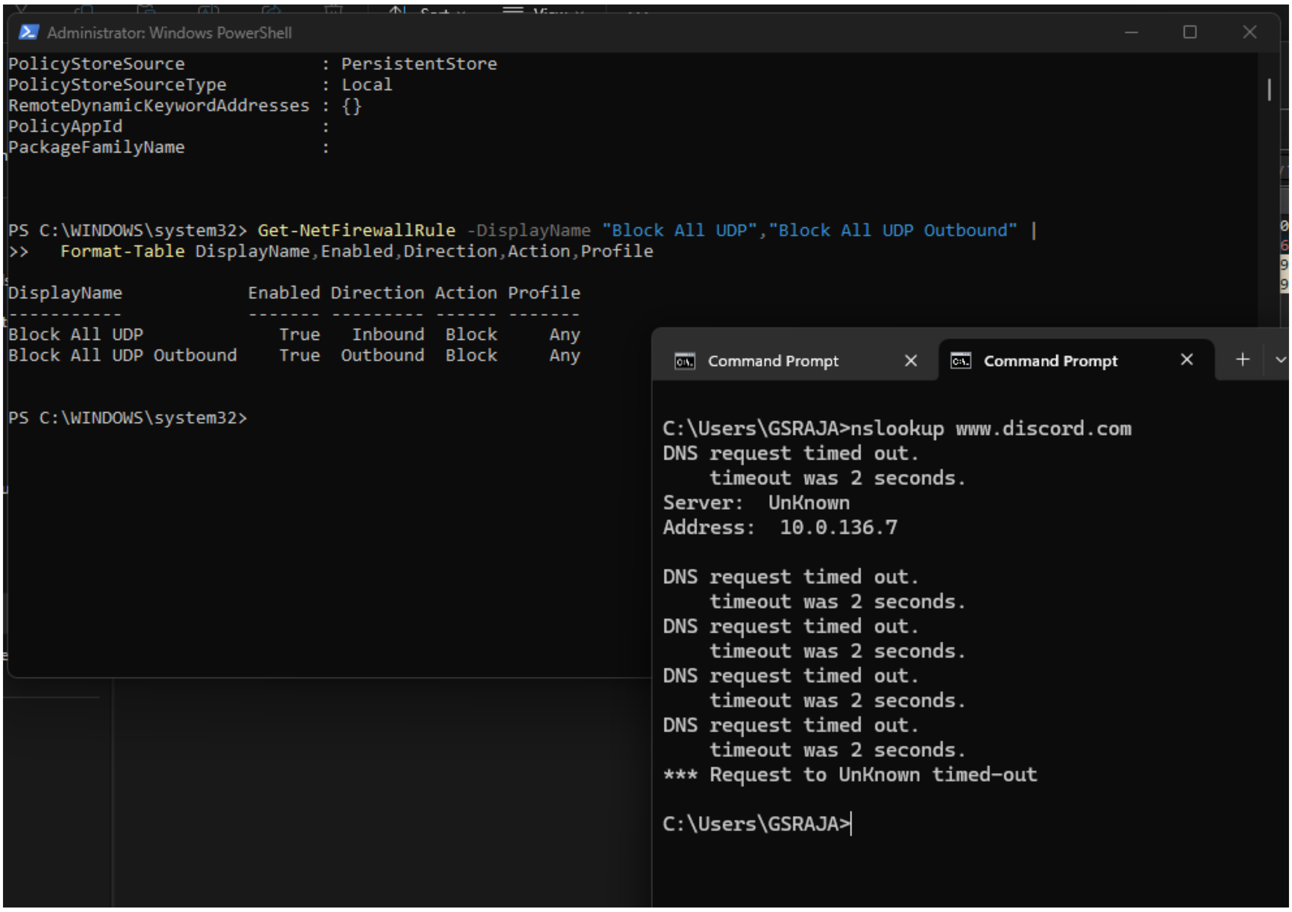

Figure 15: Blocking UDP connections in Windows (Making a DNS message to check if it is blocked or not)

Note that we can pass the -I flag in Linux to make it run the same way as Windows. Adding the -T flag uses TCP SYN probes (looks like normal connection traffic, and is often allowed through firewalls).

To view the pcap file for all the above configurations you can see the following files in the Github repo:

1. `./Task 2/Linux/traceroute discord linux block.pcap`
2. `./Task 2/Linux/traceroute discord linux no block.pcap`
3. `./Task 2/Windows/tracert windows discord block.pcap`
4. `./Task 2/Windows/tracert windows discord no block.pcap`

References

[1] traceroute(8) — linux manual page. https://linux.die.net/man/8/traceroute. Accessed 2025-09-15.

[2] Tyler Carrigan. Traceroute: Finding meaning among the stars. Red Hat Blog, https://www.redhat.com/en/blog/traceroute-finding-meaning, October 2019. Accessed 2025-09-15.

[3] S. G. Kulkarni. DNS PROTOCOL, MESSAGES. Microsoft Teams, September 2025.

[4] Larry and Sachin Divekar. What does "***" mean when traceroute. https://serverfault.com/questions/334029/what-does-mean-when-traceroute, 2011. Stack Exchange, Question asked Nov 23, 2011; accessed 2025-09-15.

[5] P. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, November 1987.