CS536 Lab 3

Borna Tavasoli

October 15, 2025

> we don't make mistakes, we have happy accidents
>
> *Bob Ross*

Problem 1.

1.1 System Set-Up, Traffic Generation, and Capture

To begin, the Ethernet interface `veth0` on one of the amber machines in the HAAS G050 lab was used. Promiscuous mode was required for sniffing Ethernet frames, which necessitated superuser privileges.

```
sudo /usr/local/etc/tcpdumpwrap—veth0 —c 32 —w — > etherlogfile
```

This command captures 32 Ethernet frames and stores them into `etherlogfile`. The captured file was placed under the directory `lab3/v1/` as required.

Traffic generation: Traffic was generated using the UDP ping application from Lab 2. The ping server was bound to IP address `192.168.1.1`, and the client was executed from the same machine with:

```
veth 'udppingc 192.168.1.1 8080 123456 8081 100'
```

Here, the `veth` wrapper executed the client on the virtual interface with IP `192.168.1.2`. This allowed packet exchange between `192.168.1.1` and `192.168.1.2` over a virtual Ethernet link. The configuration of `veth0` at the remote end was verified using:

```
veth 'ifconfig veth0'
```

1.2 Traffic Analysis

After generating sufficient traffic, the file `etherlogfile` was analyzed using Wireshark.

Step 1 - Opening in Wireshark: The file was opened in Wireshark with:

```
/usr/bin/wireshark etherlogfile
```

Wireshark automatically decoded the Ethernet, IP, and UDP headers, displaying packet structure. Results is shown in Figure 1 and 2.

[h]

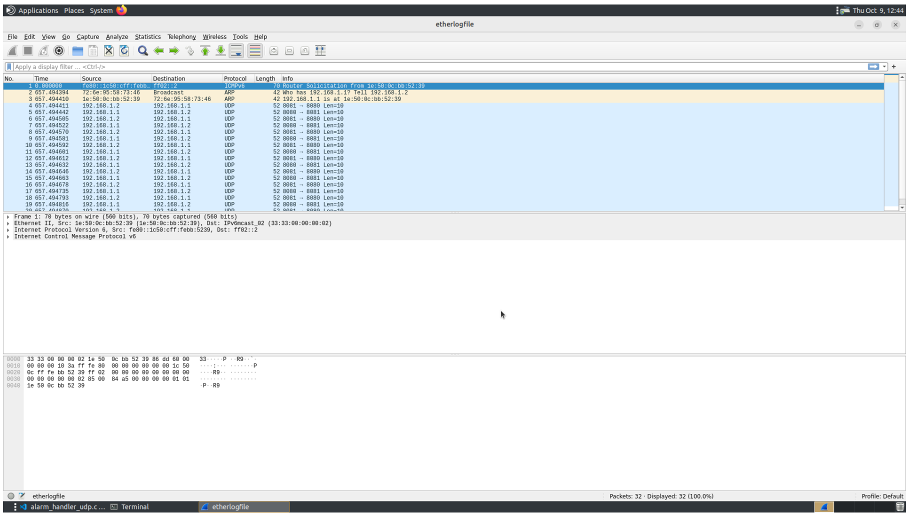


Figure 1: Wireshark environment

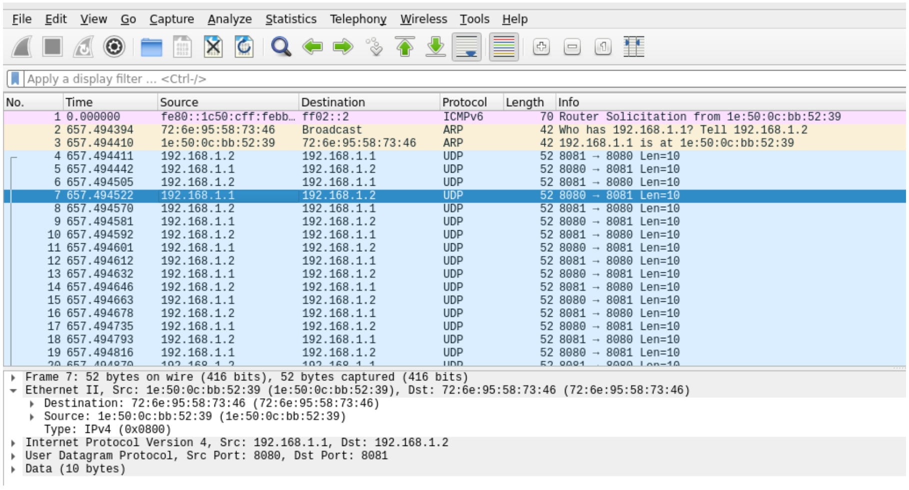


Figure 2: Wireshark closer look

Step 2 - Identifying MAC addresses: The MAC addresses corresponding to `192.168.1.1` and `192.168.1.2` were obtained using:

```
ifconfig -a
veth 'ifconfig -a'
```

These were then matched with the source and destination addresses in the Ethernet frames. Results is shown in Figure 3 and 4.

Note that `0xc0a80101` is hex for 192.168.1.1. Similarly `0xc0a80102` is hex for 192.168.1.2.

Step 3 - Checking Frame Type: The Ethernet Type field was inspected and confirmed to correspond to IPv4 (`0x0800`), meaning the frames used the DIX (Ethernet II) format. Results is shown in Figure 5.

Step 4 - Inspecting IP and UDP headers: The source and destination IP addresses, as well as port numbers, were confirmed to match the ones used in the `udppingc` application. Results is shown in Figure 6.

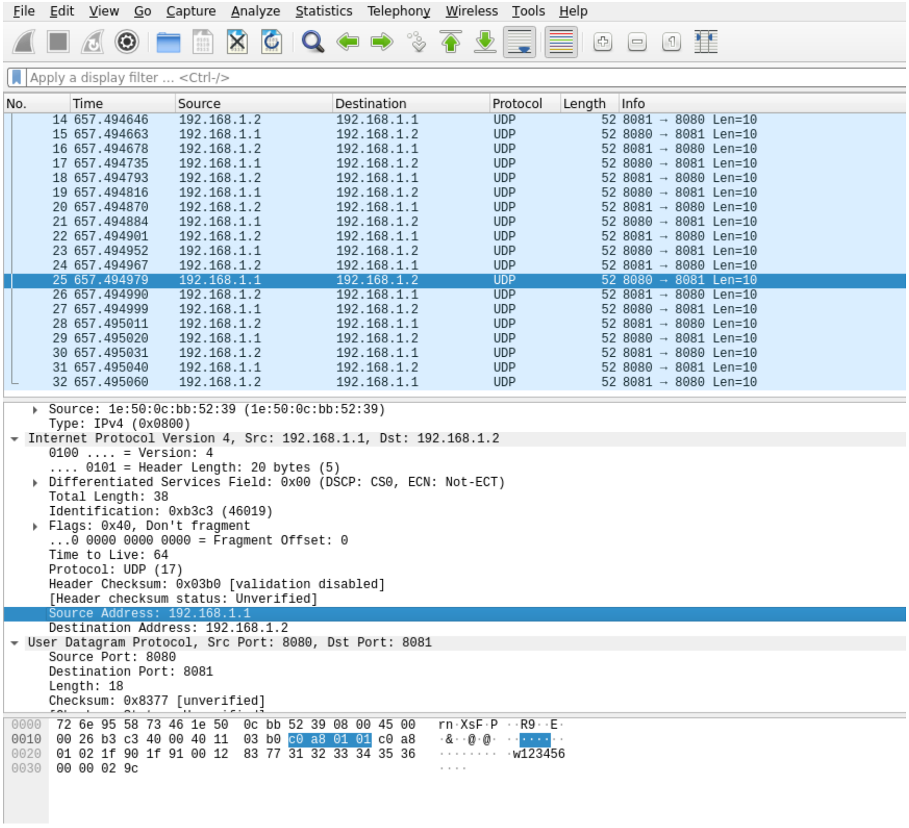


Figure 3: Source Mac Address

Step 5 - Inspecting application layer payload: The remaining bytes beyond the UDP header correspond to the application-layer payload sent via `sendto()`. By switching to "View → Packet Bytes" in Wireshark, the payload was viewed in raw hexadecimal format to verify transmitted data. Results is shown in Figure 7.

We can verify that this payload is consistent among packets. Results is shown in Figure 8.

Verification with tcpdump: To double-check, the following command was used:

tcpdump −r − < etherlogfile

This confirmed consistent IP addresses, port numbers, and packet structure as seen in Wireshark. Results is shown in Figure 9.

Summary

- The Ethernet frames were confirmed to use Ethernet II (DIX) format with Type field value `0x0800`.
- IP headers contained source and destination IPs of `192.168.1.1` and `192.168.1.2`.
- UDP headers contained the correct source (8080) and destination (8081) port numbers used by the client and server.
- The application payload matched the data generated by the `udppingc` ping client.
- Both Wireshark and tcpdump produced consistent interpretations of the captured data.

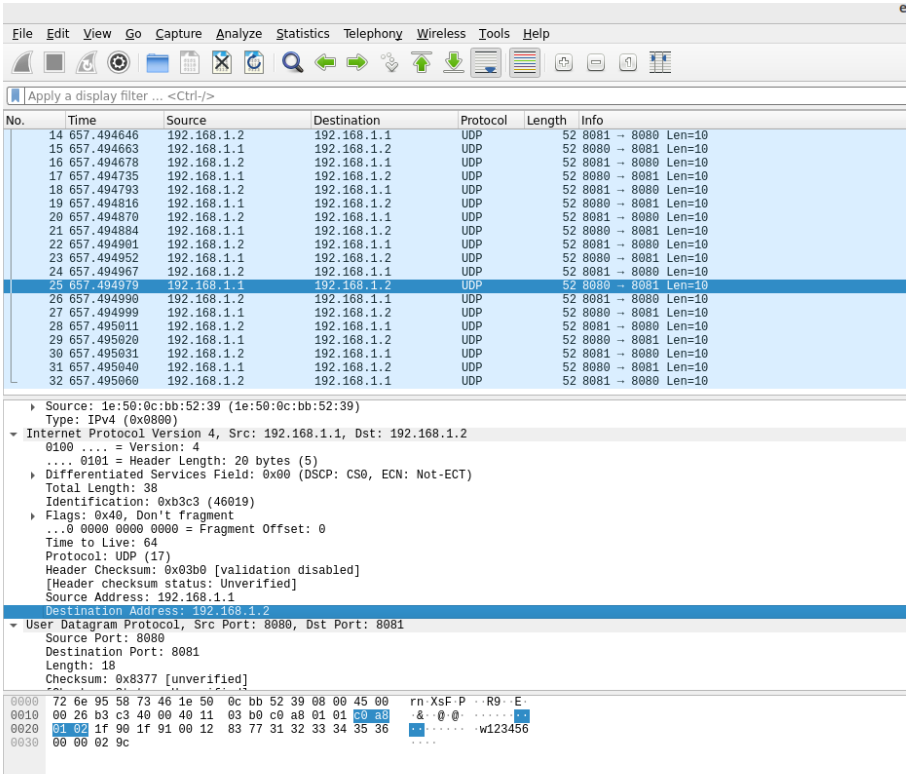


Figure 4: Destination Mac Address

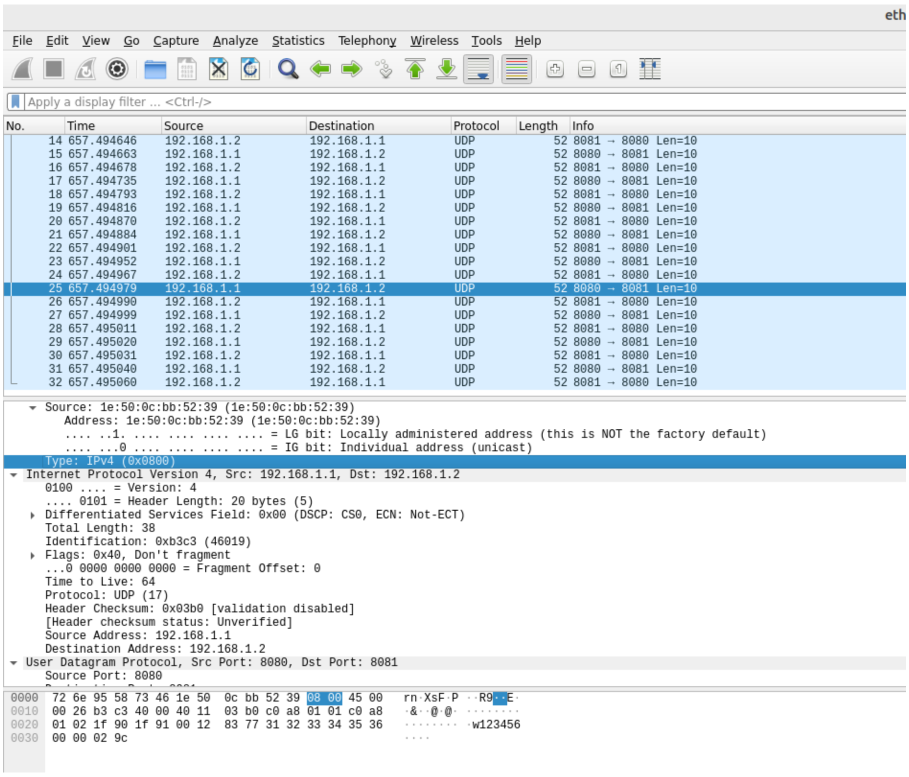


Figure 5: Ethernet Type

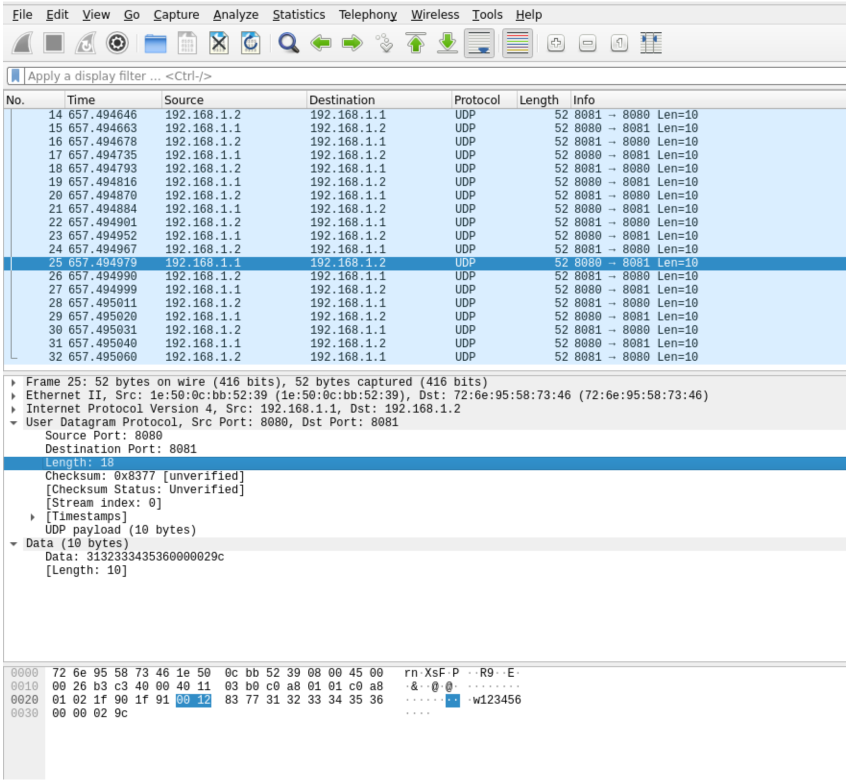


Figure 6: UDP Headers

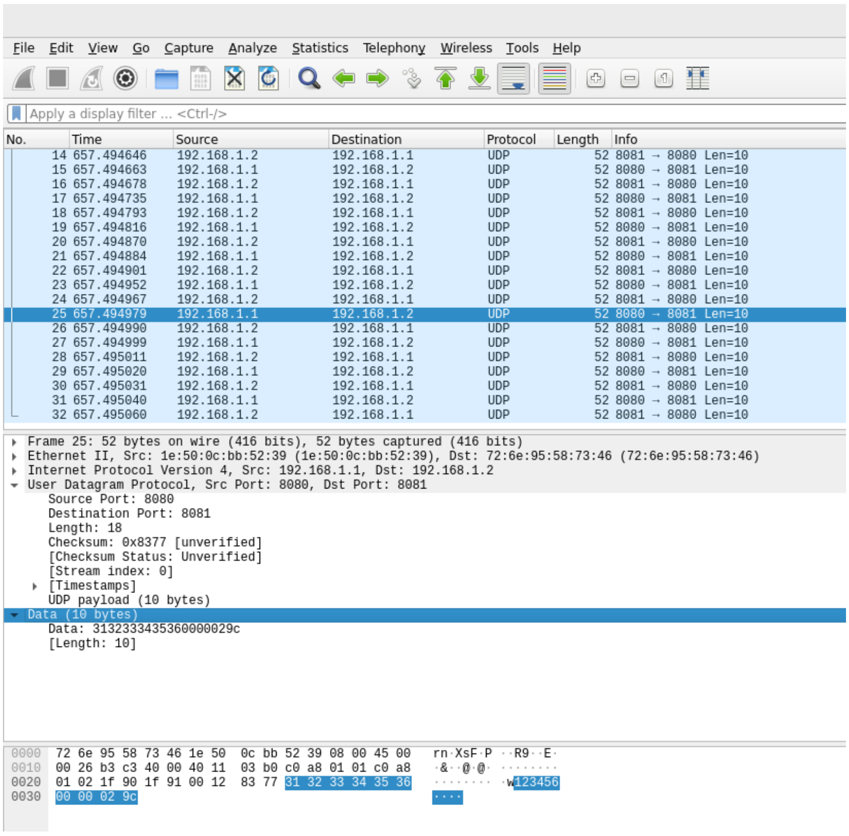


Figure 7: Payload View

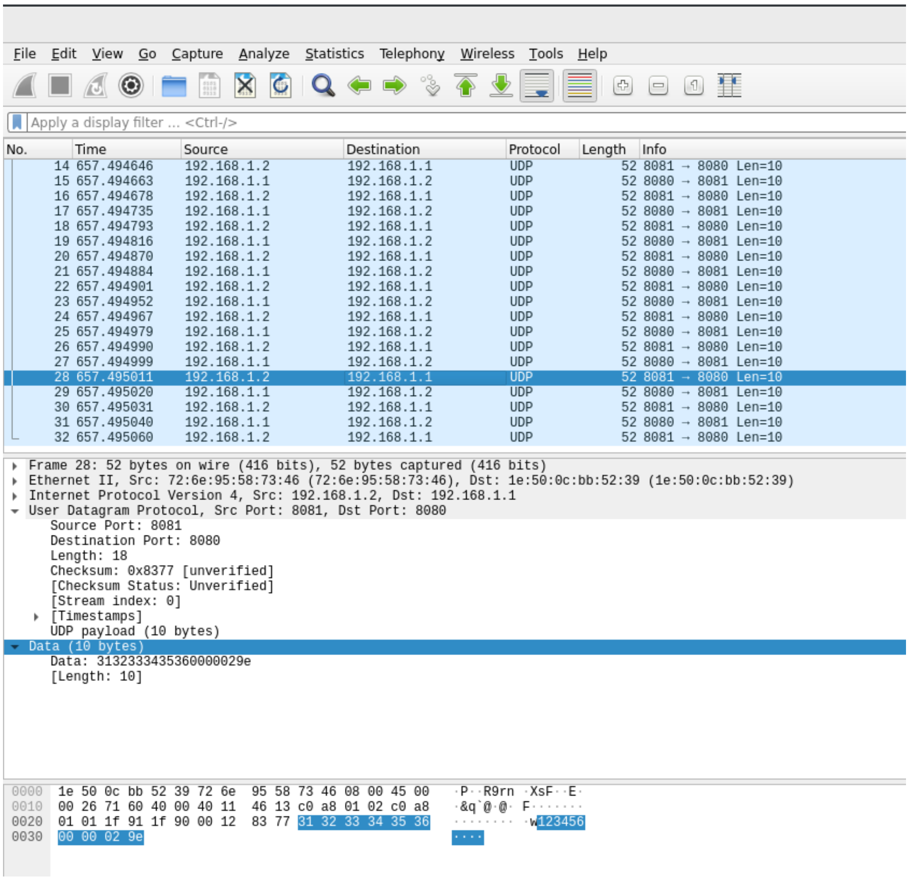


Figure 8: Checking payload consistency

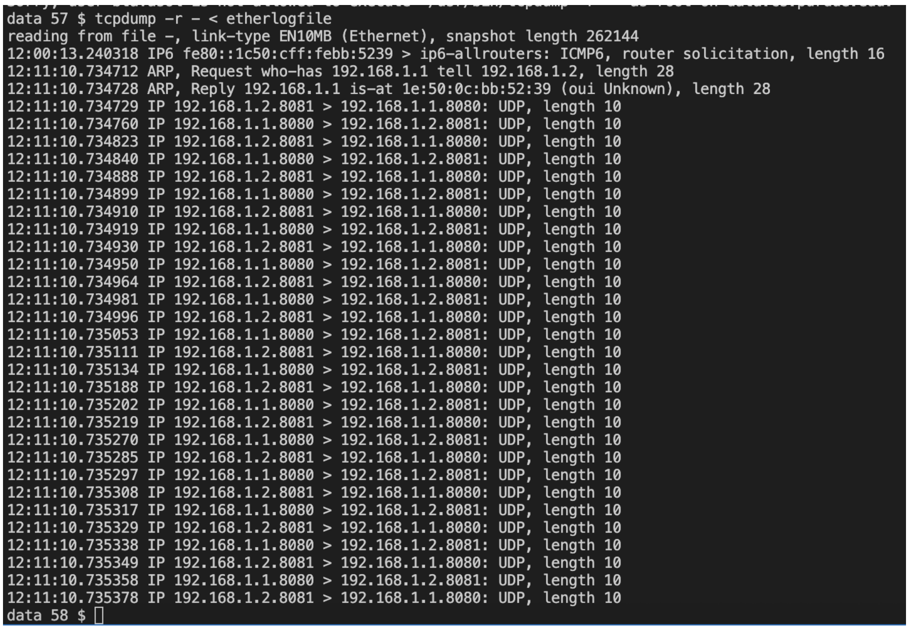


Figure 9: TCP dump output

Problem 2.

Asynchronous Concurrency Management

The sender implementation introduces asynchrony through the `SIGIO` signal handler, which processes incoming ACK packets while the main synchronous loop transmits data packets. This creates potential race conditions when both the signal handler and main loop access shared data structures, specifically the `ack_received[]` array that tracks which packets have been successfully acknowledged.

Our implementation uses a reader-writer separation strategy that avoids the need for complex synchronization primitives such as mutexes or signal masking. If you look at the code:

- Synchronous code (main loop): Only reads from the `ack_received[]` array
- Asynchronous code (SIGIO handler): Only writes to the `ack_received[]` array

By doing so, we avoid race conditions because:

1. Neither component performs atomic read-modify-write sequences on shared data
2. The signal handler only transitions flags from 0 to 1, never reverses them
3. If the main loop reads a stale value (0 instead of 1), it simply retransmits the packet unnecessarily, which is safe because the receiver handles duplicates correctly
4. On modern architectures, byte-sized writes are atomic which means we won't have any incomplete reads of individual array elements

Thus by enforcing strict read-only and write-only roles for the synchronous and asynchronous components respectively, our implementation achieves correct concurrent operation without the overhead and complexity of explicit synchronization primitives.

Duplicate File Transfer Initiation Packet Handling

After the receiver sends an acknowledgment to a file transfer initiation packet, it sets a 250ms timer to wait for the first data packet. During this waiting period, duplicate initiation packets may arrive from the sender: either due to network delays causing the sender to timeout and retransmit, or due to genuine packet duplication in the network. The receiver must handle these duplicates correctly without disrupting the established session.

In our approach, the receiver maintains a `transfer_active` flag that tracks whether a file transfer session is currently in progress. This flag transitions through three states:

1. Idle (`transfer_active = 0`): Waiting for a new initiation packet
2. Handshake (`transfer_active = 1, next_expected = 0`): Init ACK sent, waiting for first data packet
3. Active transfer (`transfer_active = 1, next_expected > 0`): Receiving data packets

If a duplicate initiation packet (16 bytes) arrives before the first data packet, the receiver responds with an idempotent acknowledgment and resends the 6-byte filename ACK without modifying any state (memory buffers, tracking arrays, session parameters remain unchanged). This is safe because multiple identical ACKs don't corrupt protocol state. Moreover, if the original ACK was lost, the duplicate provides recovery. We will then restart the 250ms timer automatically and wait for the data. Once the first data packet arrives, initiation packets from different senders are ignored to prevent session hijacking. Note that once the connection is established, getting 16 byte packets (whole packet or remainder of a packet) will not cause any issues since we will be in the third stage.

Performance Optimization

Experimental Setup and Result

We conducted performance experiments using a 102.0 kB test file to evaluate the impact of protocol parameters on transfer completion time. The two primary tunable parameters are:

- Micropace: Delay in microseconds between consecutive packet transmissions
- Payload size: Number of data bytes per UDP packet (excluding 4-byte sequence number header)

All tests were performed in the local lab environment (low latency, high bandwidth, minimal background loss). Below is our result:

Micropace (µs)	Payload (bytes)	Transfer Time (ms)
500	200	100
500	500	35
500	1000	22
500	2000	14
500	3000	10
300	1000	26
100	1000	15

Table 1: Transfer completion time for 102.0 kB file

Analysis

Impact of Payload Size Holding micropace constant at 500µs while varying payload size reveals a clear inverse relationship between payload size and transfer time. Key observations:

- Larger payloads reduce the number of packets required: 102 kB with 200-byte payload requires 510 packets, while 3000-byte payload requires only 34 packets
- Fewer packets mean less per-packet overhead (UDP/IP headers, system calls, protocol bookkeeping)
- Transfer time improves dramatically: from 100ms (200B payload) to 10ms (3000B payload)

However, payload size is constrained by the maximum UDP payload that avoids IP fragmentation. We limited our experiments to payloads $\leq$ 3000 bytes to stay safely below the typical MTU of 1500 bytes.

Impact of Micropace Holding payload constant at 1000 bytes while varying micropace shows diminishing returns:

- Reducing micropace from 500µs to 100µs improves time from 22ms to 15ms (32% improvement)
- However, reducing micropace from 500µs to 300µs *increases* time from 22ms to 26ms

The non-monotonic behavior at 300µs suggests measurement variability or system scheduling effects at small time scales. The general trend shows that lower micropace (faster transmission rate) reduces completion time by minimizing idle time between packet transmissions.

Through iterative experimentation, we identified that a configuration of approximately:

- **Micropace**: 300µs
- **Payload**: 8000 bytes

yields transfer times around **3ms** for the 102 kB file in our lab environment. This is a 33× speedup compared to the baseline configuration (500µs, 200B payload).

Final Notes

While decreasing micropace improves performance in ideal conditions, excessively low values introduce reliability concerns:

- Bursts of packets may exceed the receiver's socket buffer capacity, causing kernel-level packet drops
- In shared network environments, aggressive sending can overwhelm switch buffers or compete unfairly with other traffic
- Higher packet loss rates trigger more retransmissions, potentially negating performance gains

We observed that micropace values below 100µs occasionally resulted in failed transfers due to excessive packet loss, even in the controlled lab environment.

While larger payloads improve efficiency, they have drawbacks:

- Larger payloads mean more wasted bandwidth when a packet is lost and must be retransmitted
- For small files, large payloads may not improve performance and can increase latency variance

Performance optimization in reliable UDP protocols involves balancing throughput (achieved through large payloads and low micropace) against reliability (which degrades under aggressive parameters). For small file transfers in low-loss, low-latency environments like our lab, the protocol can achieve sub-5ms completion times with careful parameter tuning. However, these optimal parameters are environment-specific and may require adjustment for production deployments with higher loss rates, variable latency, or shared network resources.

Bonus

File: `m2_ch1_2024-02-19.pcap`

Selected interval: 13:00:05.07 – 13:00:05.60 (first 0.5 s, 806 frames)

- **Network type:** 802.11b/g/n (2.4 GHz band, Channel 1).
- **Frame types observed:** Predominantly *Beacon* frames, with occasional *Probe Request*, *Acknowledgment*, *Clear-to-Send (CTS)*, and *Null Function* frames.
- **Data rates:** Initially 1 Mbps for the first few frames, later increasing up to 12 Mbps.
- **SSID(s):** PAL3.0, PAL-Recreational, attwifi, eduroam
- **Signal strength** ≈ -65 dBm
- **Signal strength ratio** $\approx [-5, 90]$ dBm
- **Notes:** The presence of multiple beacon frames suggests nearby access points broadcasting their presence. The mixture of management and control frames indicates active association and channel scanning activity.

Below is a summary of our result:

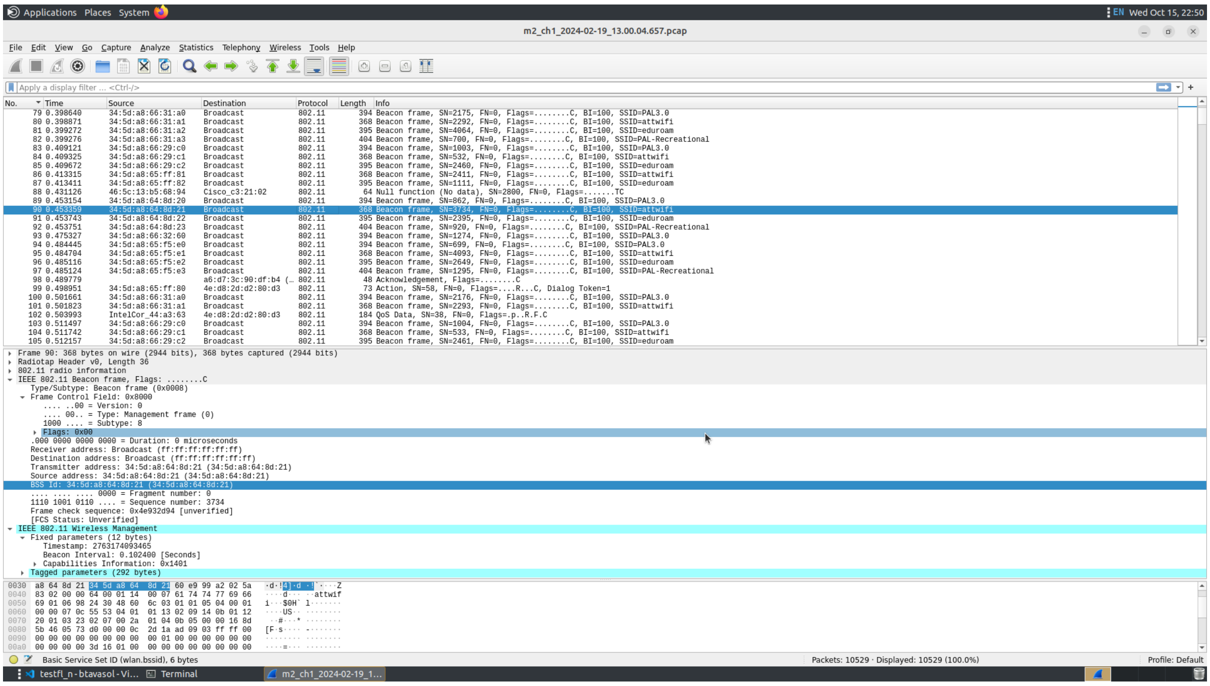


Figure 10: Bssid and ssid

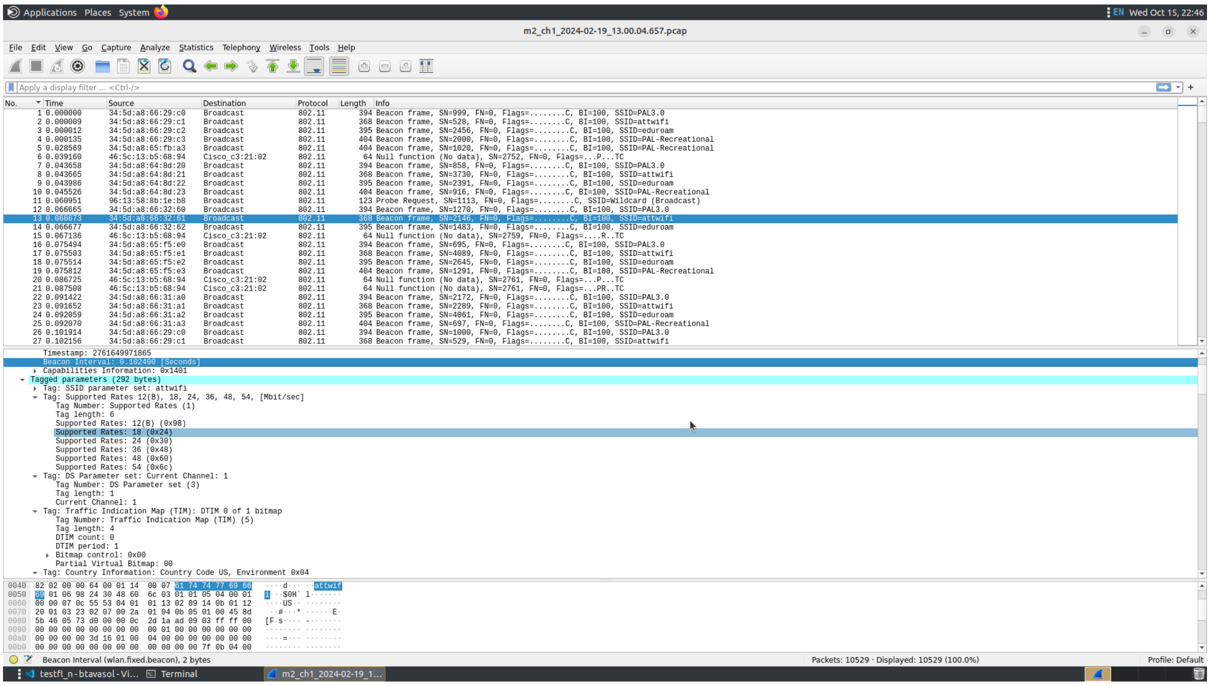


Figure 11: Channel info

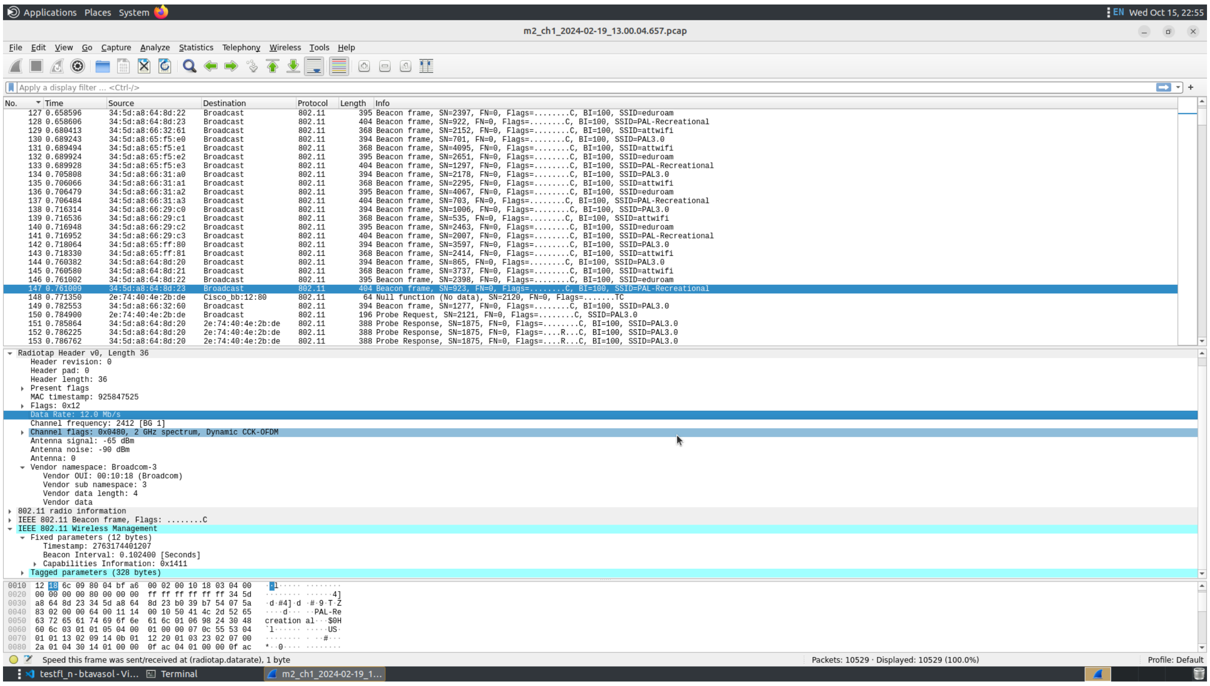
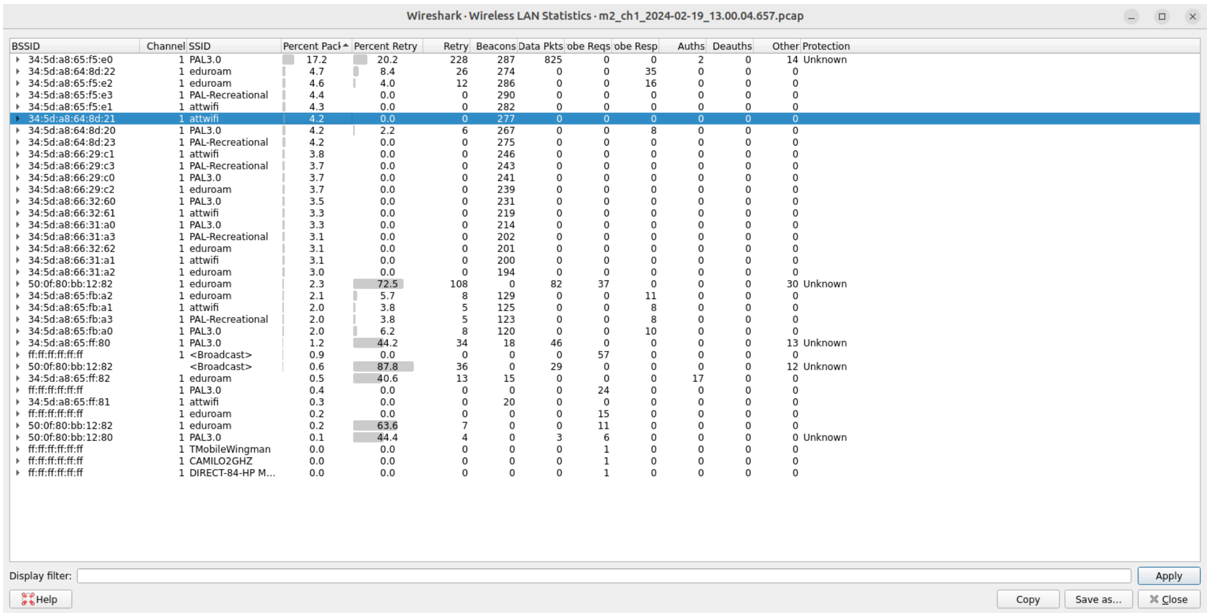


Figure 12: Data rate info



Wireshark · Wireless LAN Statistics · m2_ch1_2024-02-19_13.00.04.657.pcap

BSSID	Channel	SSID	Percent Packets	Percent Retry	Retry	Beacons	Data Pkts	Probe Reqs	Probe Resp	Auths	Deauths	Other	Protection
34:5d:a8:65:f5:e0	1	PAL3.0	17.2	20.2	228	287	825	0	0	2	0	14	Unknown
34:5d:a8:64:8d:22	1	eduroam	4.7	8.4	26	274	0	0	35	0	0	0	
34:5d:a8:65:f5:e2	1	eduroam	4.6	4.0	12	286	0	0	16	0	0	0	
34:5d:a8:65:f5:e3	1	PAL-Recreational	4.4	0.0	0	290	0	0	0	0	0	0	
34:5d:a8:65:f5:e1	1	attwifi	4.3	0.0	0	282	0	0	0	0	0	0	
34:5d:a8:64:8d:21	1	attwifi	4.2	0.0	0	277	0	0	0	0	0	0	
34:5d:a8:64:8d:20	1	PAL3.0	4.2	2.2	6	267	0	0	8	0	0	0	
34:5d:a8:64:8d:23	1	PAL-Recreational	4.2	0.0	0	275	0	0	0	0	0	0	
34:5d:a8:66:29:c1	1	attwifi	3.8	0.0	0	246	0	0	0	0	0	0	
34:5d:a8:66:29:c3	1	PAL-Recreational	3.7	0.0	0	243	0	0	0	0	0	0	
34:5d:a8:66:29:c0	1	PAL3.0	3.7	0.0	0	241	0	0	0	0	0	0	
34:5d:a8:66:29:c2	1	eduroam	3.7	0.0	0	239	0	0	0	0	0	0	
34:5d:a8:66:32:60	1	PAL3.0	3.5	0.0	0	231	0	0	0	0	0	0	
34:5d:a8:66:32:61	1	attwifi	3.3	0.0	0	219	0	0	0	0	0	0	
34:5d:a8:66:31:a0	1	PAL3.0	3.3	0.0	0	214	0	0	0	0	0	0	
34:5d:a8:66:31:a3	1	PAL-Recreational	3.1	0.0	0	202	0	0	0	0	0	0	
34:5d:a8:66:32:62	1	eduroam	3.1	0.0	0	201	0	0	0	0	0	0	
34:5d:a8:66:31:a1	1	attwifi	3.1	0.0	0	200	0	0	0	0	0	0	
34:5d:a8:66:31:a2	1	eduroam	3.0	0.0	0	194	0	0	0	0	0	0	
50:0f:80:bb:12:82	1	eduroam	2.3	72.5	108	0	82	37	0	0	0	30	Unknown
34:5d:a8:65:fb:a2	1	eduroam	2.1	5.7	8	129	0	0	11	0	0	0	
34:5d:a8:65:fb:a1	1	attwifi	2.0	3.8	5	125	0	0	8	0	0	0	
34:5d:a8:65:fb:a3	1	PAL-Recreational	2.0	3.8	5	123	0	0	8	0	0	0	
34:5d:a8:65:fb:a0	1	PAL3.0	2.0	6.2	8	120	0	0	10	0	0	0	
34:5d:a8:65:ff:80	1	PAL3.0	1.2	44.2	34	18	46	0	0	0	0	13	Unknown
ff:ff:ff:ff:ff:ff	1	<Broadcast>	0.9	0.0	0	0	0	57	0	0	0	0	
50:0f:80:bb:12:82		<Broadcast>	0.6	87.8	36	0	29	0	0	0	0	12	Unknown
34:5d:a8:65:ff:82	1	eduroam	0.5	40.6	13	15	0	0	0	17	0	0	
ff:ff:ff:ff:ff:ff	1	PAL3.0	0.4	0.0	0	0	0	24	0	0	0	0	
34:5d:a8:65:ff:81	1	attwifi	0.3	0.0	0	20	0	0	0	0	0	0	
ff:ff:ff:ff:ff:ff	1	eduroam	0.2	0.0	0	0	0	15	0	0	0	0	
50:0f:80:bb:12:82	1	eduroam	0.2	63.6	7	0	0	11	0	0	0	0	
50:0f:80:bb:12:80	1	PAL3.0	0.1	44.4	4	0	3	6	0	0	0	0	Unknown
ff:ff:ff:ff:ff:ff	1	TMobileWingman	0.0	0.0	0	0	0	1	0	0	0	0	
ff:ff:ff:ff:ff:ff	1	CAMILO2GHZ	0.0	0.0	0	0	0	1	0	0	0	0	
ff:ff:ff:ff:ff:ff	1	DIRECT-84-HP M...	0.0	0.0	0	0	0	1	0	0	0	0	

Figure 13: Frame type summary

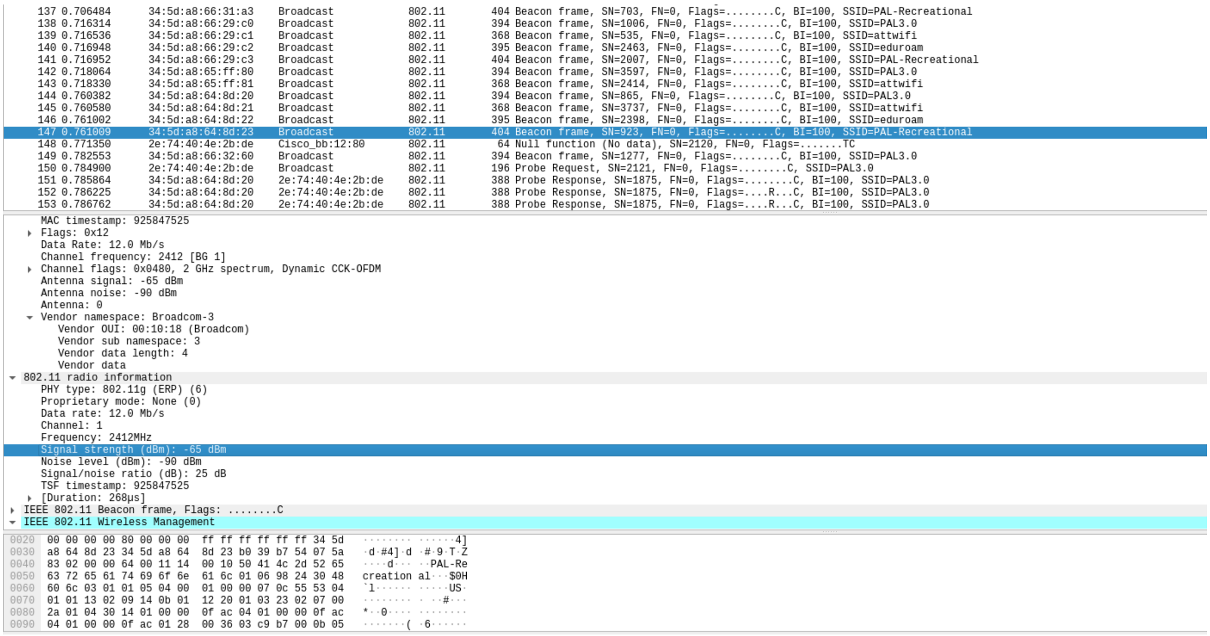


Figure 14: Signal strength

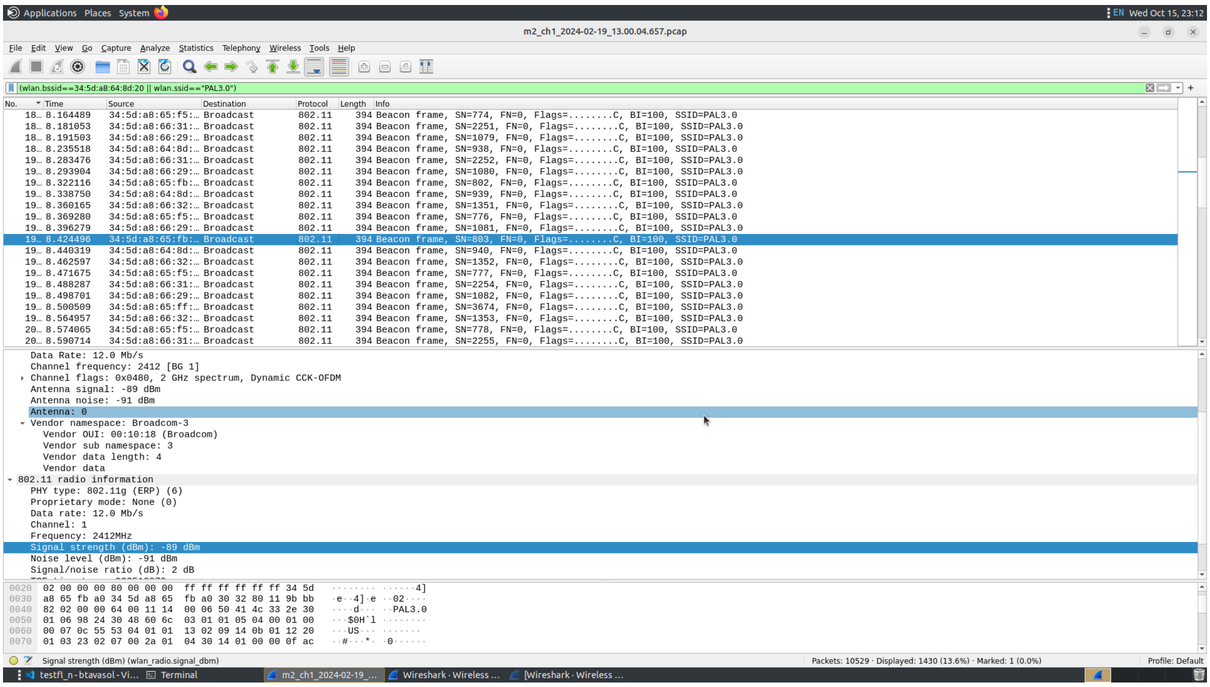


Figure 15: SNR of hightraffic basic service sets and their MAC addresses

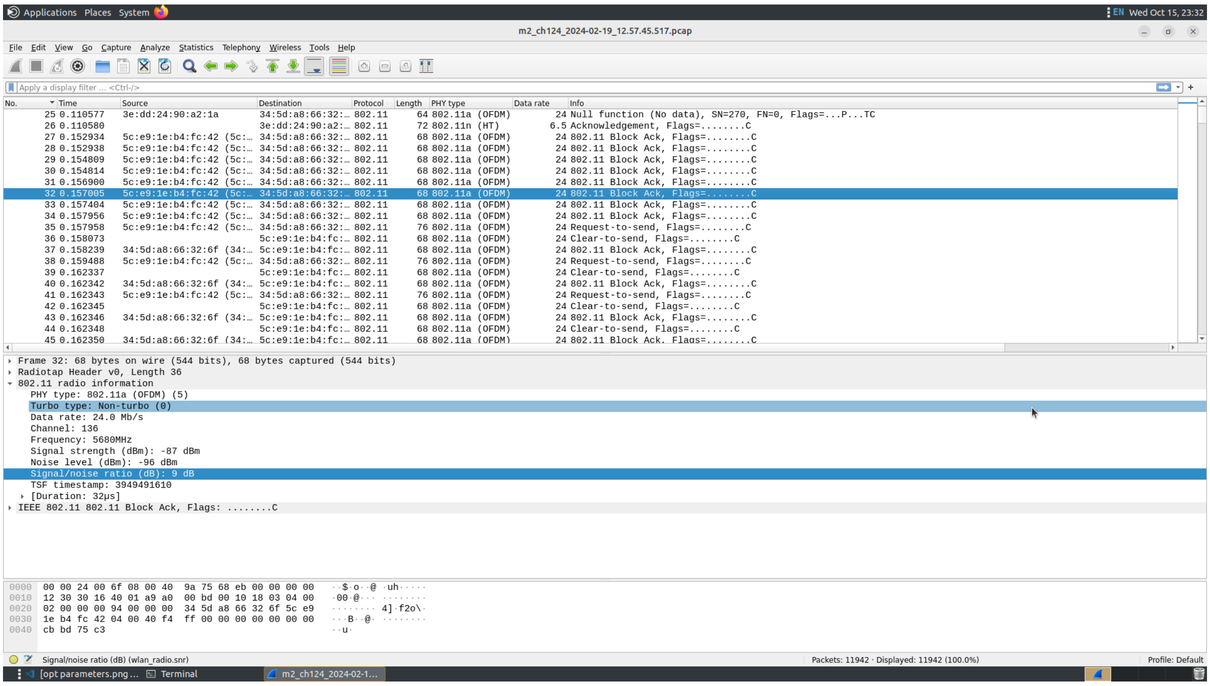


Figure 16: SNR, network type, and signal strength

File: m2_ch124_2024-02-19.pcap

Selected interval: [Specify e.g., 12:57:45.88 – 12:57:46.39] (about 256 packets)

- **Network type:** 802.11a (5 GHz band, Channel 136).
- **Frame types observed:** Mostly Block Ack, RTS and CTS, some beacon
- **Data rates:** 24 to 12 Mb/s (we also have 6, 6.5, and 48)
- **SSID(s):** Same as before
- **Signal strength:** ≈ -80 dBm
- **Signal strength ratio:** $\approx [9, 15]$ dBm
- **Notes:** Signal seems to be stronger than before. Also we have a wider band.

Below is a summary of our results:

File: m2_ch153_2024-02-19.pcap

Selected interval: The complete interval.

- **Network type:** Likely 802.11a same as before (5 GHz band, Channel 153).
- **Frame types observed:** Block Ack, RTS, CTS, VHT/HE NDP Announcement, Probe Request
- **Data rates:** 6, 6.5 and 24
- **SSID(s):** Wildcard, PAL3.0, Hilton Honors, Rajkiran, FreeORDWiFi, SSN, Hotel bobo

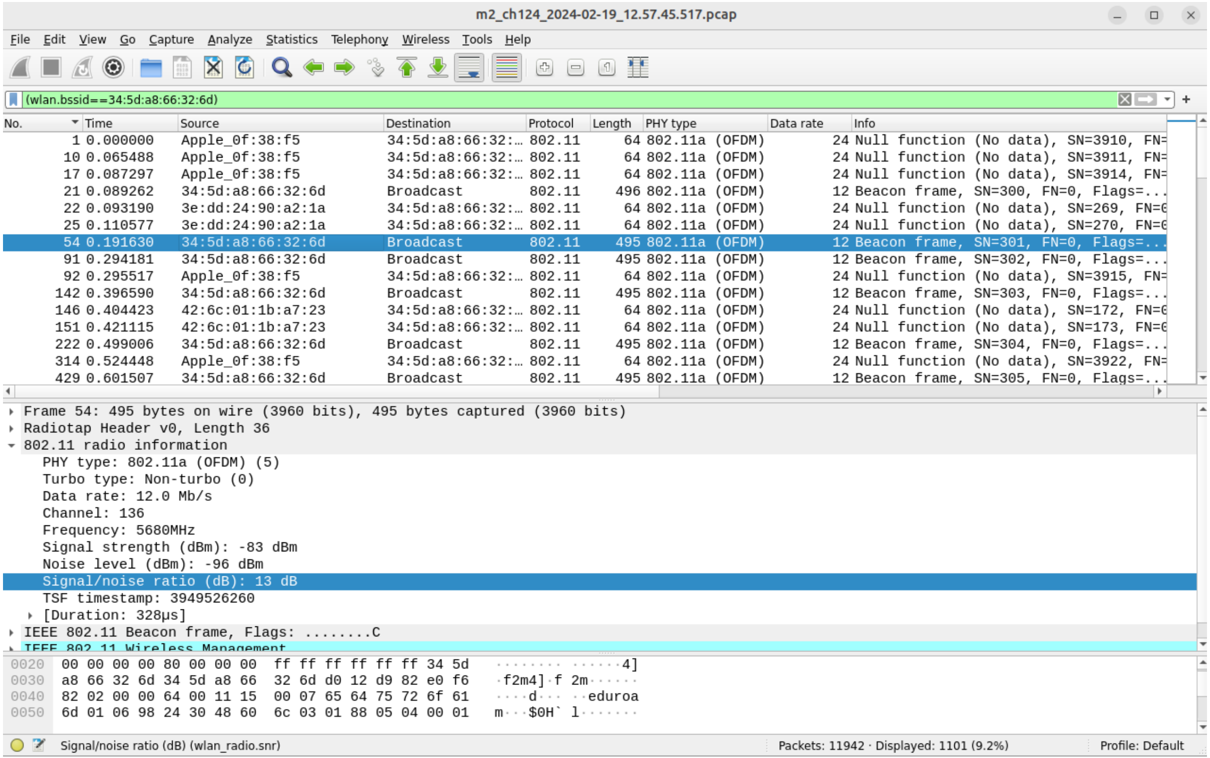


Figure 17: SNR of hightraffic basic service sets and their MAC addresses

- **Signal strength: -86** dBm
- **Signal strength ratio: 45** dBm
- **Notes:** Same band as before. Better signal ratio. Less different data rates. Different SSIDs.

Below is a summary of our findings:

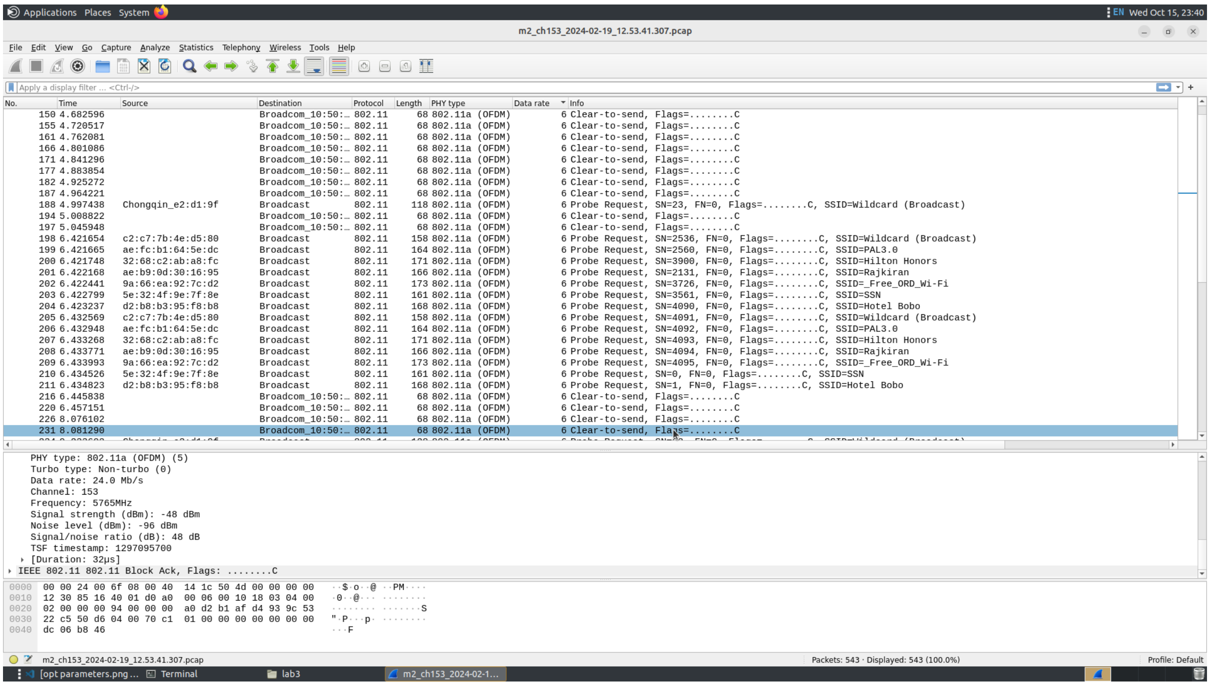


Figure 18: SSID and datarates

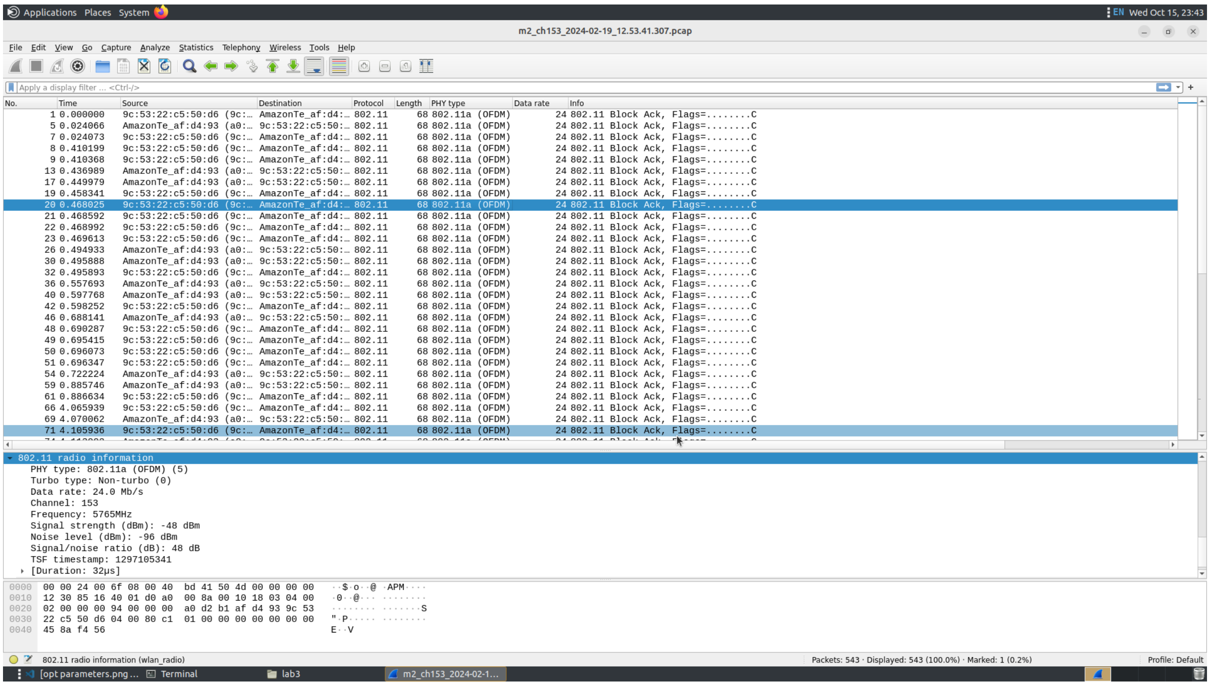


Figure 19: SNR and strength

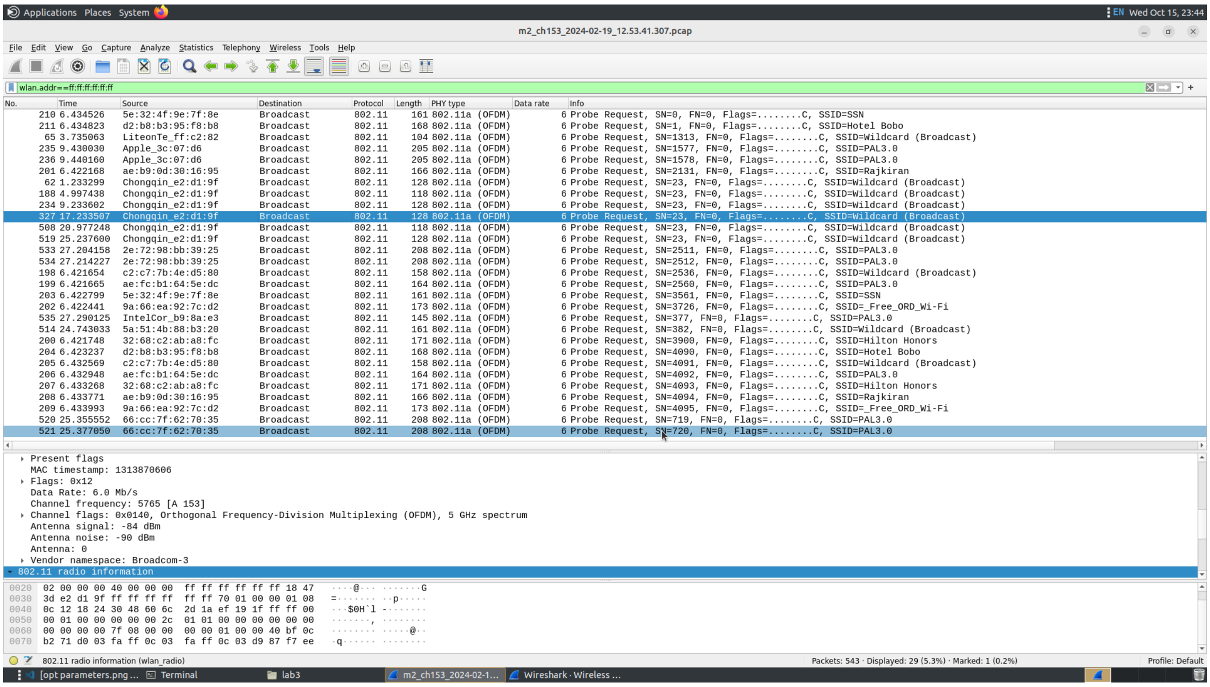


Figure 20: SNR of hightraffic basic service sets and their MAC addresses