**ELEC3506/9506 Communication Networks**



The University of Sydney

**– Lab Report 1 of Lab Report**

**Xinran He (540662177), contributions (50%)**

**Zhixuan Lin (530034414),**

**contributions ( 50%)**



**Table of Contents**

[1. Introduction 3](#_Toc28653)

[2. Phase 1: Getting Started 3](#_Toc5284)

[2.1. Experimental Procedures 4](#_Toc15416)

[2.2. Results 4](#_Toc19709)

[2.2.1. Different Protocols Observed 4](#_Toc27165)

[2.2.2. HTTP GET and 200 OK Exchange 5](#_Toc12963)

[2.2.3. Time Difference (Latency Measurement) 5](#_Toc3789)

[2.2.4. IP Address Identification 5](#_Toc22731)

[3. Phase 2: Ethernet and ARP 6](#_Toc8272)

[3.1. Ethernet Frame Analysis 6](#_Toc22135)

[3.2. ARP Cache Table 6](#_Toc22094)

[3.3. ARP Request and Reply 6](#_Toc22171)

[3.4. A Special Case: ARP Request Unattended 7](#_Toc25132)

[3.5. EX-1: Manually Adding an Incorrect ARP Entry 7](#_Toc13884)

[4. Appendix 8](#_Toc16687)

# Introduction

During this lab we learnt about network protocols through the observation of them in use using Wireshark. Their main goal was to gain some insight into how packets of data are sent and processed across various protocol layers, primarily at the application level (HTTP) and link level (Ethernet and ARP). Taking the real-time traffic as capture, we observed packet encapsulation, Ethernet frame structure and addressing resolution between IP address and MAC addresses.

This hands-on experience complemented theoretical knowledge we had gained from lectures and allowed us to relate abstract ideas with concrete examples based on packet traces of a large-scale live network. The lab was not only meant to give us a glimpse into the hierarchical duties of protocols in different layers for the network stack, but also to remind that anecdotal data (e.g. screenshots) should always be employed so as to verify our answers. In general, the lab improved our analytical and critical insights into communication networks in line with both learning outcomes and marking criteria such as clarity, accuracy of interpretation, and evidence-based response.

# Phase 1: Getting Started

In the first phase, we focused on two areas. First we became acquainted with Wireshark as a packet sniffer and analyzer. Then we observed the actual behavior of network protocols in use. To put it concretely, we wanted monitor between our computer and a remote web server traffic and breakdown the way that data packets are shaded through diverse levels of protocol. This phase also introduced us to the essential operating methods of Wireshark, including start stop capture, apply a filter and open out protocol headings for detailed observation.

During this experiment, we successfully took a number of network packages including HTTP messages, TCP segments, IP datagrams, and Ethernet frame. We used an HTTP display filter to show just the HTTP GET request and HTTP/1.1 200 OK response between us and our server gaia.cs.umass.edu. In addition we observed the time difference between OK and GET messages. This gave us an estimate of server response time. In addition our machine box's own IP address and the server's IP were both picked up. This confirmed there was correct end-to-end communication.

## Experimental Procedures

### Launching Wireshark

Launched the Wireshark program and chose the current Internet access network interface.

### Starting Packet Capture

Started sniffing packets on the selected interface.

### Generating HTTP Traffic

Opened browser and navigated to URL:

[http://gaia.cs.umass.edu/wireshark-labs/INTRO-wireshark-file1.html](http://gaia.cs.umass.edu/wireshark-labs/INTRO-wireshark-file1.html" \t "https://app.justdone.ai/tools/_blank)

The webpage loaded and said congratulations.

### Stopping Packet Capture

terminated the capturing process after page retrieval completion.

### Filtering HTTP Packets

terminated the capturing process after page retrieval completion.

### Identifying Key Packets

### Found the client's computer to make HTTP GET request.

### Locate the below response depends on your trigger from server : HTTP 200 OK.

### Analyzing Packet Details

### I opened the packet details to look at encapsulation in different headers: Ethernet frame, IP datagram, TCP segment, and HTTP.

### Replaced superfluous fields of the protocol and added most of the missed HTTP header information.

### Recording Measurements

### Observed the delta between GET and OK.

### Determined the IP address of server (gaia. cs. umass. edu) and the local host.

## Results

## Different Protocols Observed

### Multiple other protocols were intercepted in the unfiltered packet listing (see Figure 1) as well as HTTP traffic. Three examples include:

### TCP – for reliable transport or data between applications.

### TLSv1. 2/1.3 – demonstrating secure session setup and encrypted traffic.

### IGMPv3 – Multicast Group Management running on multicast capable switches/routers as well as end nodes to manage membership of group. This shows that there are many hidden protocols running behind a website even if one just asks for a simple webpage as it is the case here.

## HTTP GET and 200 OK Exchange

### Then by using HTTP display filter (as shown in Figure 2), we located the HTTP GET request sent by client (192.168.0.80) to server (182.254.116.116) and the corresponding response from server an HTTP/1.1 200 OK response, were found After applying the display filter of HTTP we identified point-to-point details about Data Transfer, its acknowledgment signal sharing between devices and responses are as follows: Source: 192.168.0.80 Destination: 182.india/myweb/trialpage/.5%13C).

### GET message: Frame 160, Source = 192.168.0.80, Destination = 182.254.116.116

### OK message:Frame 167, Source = 182.254.116.116, Destination = 192.168.0.80

## Time Difference (Latency Measurement)

### From looking at the Time column in Wireshark (Figure 3), we can see that the time span between our GET request being issued (at 2.452839 seconds) and the corresponding 200 OK response (at 2.719622 seconds) was:

### Δt=2.719622−2.452839≈0.267seconds

### This number represents the server response time over the Internet path.

## IP Address Identification

### Server IP (gaia.cs.umass.edu / www-net.cs.umass.edu): 182.254.116.116

### Local host IP: 192.168.0.80

### These addresses were verified in the IP header of each of the GET & OK packets

# Phase 2: Ethernet and ARP

## Ethernet Frame Analysis

### We then analyzed the Ethernet frames containing the HTTP GET request and its 200 OK response (Figure 2, Figure 3).

### Source MAC (GET): c0:bf:be:5e:ee:a6 (local NIC: AzureWaveTec).

### Destination MAC (GET): 20:23:51:ee:dc:a9 (default gateway-> TPLINK router).

### Frame Type: 0x0800 (which is the IPv4 payload).

### ASCII offset: The “G” of character of the GET request in ~54th byte with respect to beginning of the Ethernet frame payload and similarly web server has replied “O” (200 OK) at similar offset in reply frame.

### This proves that Ethernet frames encapsulate upper-layer protocols; and the MAC address for router, not the remote server, will be located( as it needs to send packet through Internet).

## ARP Cache Table

### We displayed the cached entries with arp -a before clearing the ARP cache (Figure 4).

### This is the table of some mappings etc.

### 192.168.0.1 → 20-23-51-ee-dc-a9 (dynamic)

### 192.168.0.80 → c0-bf-be-5e-ee-a6 (local)

### Columns includes: Internet Address (IP), Physical Address (MAC) and Type (dynamic or static).

### This has proven that the machine holds IP ↔ MAC mappings locally, thus avoiding excessive ARP requests.

## ARP Request and Reply

### From the ARP-only filtered capture (part ARP. pdf, Figure 1):

### ARP Request:

### Source MAC = 32:3c:d6:64:b6:99 (host).

### Destination MAC = ff:ff:ff:ff:ff:ff (broadcast).

### Opcode = 1 (Request).

### Query: “Who has 192.168.0.1? Tell 192.168.0.243”.

### ARP Reply:

### Source MAC = c0:bf:be:5e:ee:a6 (host interface).

### Destination MAC = 20:23:51:ee:dc:a9 (router).

### Opcode = 2 (Reply).

### Answer: “192.168.0.80 is at c0:bf:be:5e:ee:a6”

### These prove ARP is working fine to resolve IP address to MAC within the local subnet.

## A Special Case: ARP Request Unattended

### The 6th ARP Request in the ethernet-ethereal-trace-1 example (Figure 5), went:

### "192.168.1.117 is where are you? Tell 192.168.1.104. But no answer cam back; it opsped the message only to jerk up a moment later in this fashion."

### This time, no reply was received. The reason open to all is because at this time the queried IP address was not in use on the subnet. Besides, there were no hosts by back-turia politeness to bounce off of nor anything at all one might even call hospitalitie. This demonstrates that when a device target is not online or somewhere exterior from one's reachable local network, ARP requests will presumably fail.

## EX-1: Manually Adding an Incorrect ARP Entry

### Steps

### Launch the Command Prompt in Administrator mode.

### Insert an static ARP entry manually, for example:

### arp -s 192.168.0.100 02-00-00-00-00-01

### I will use the address 192.168.0.100.

### 02-00-00-00-00-01 is a deliberately invalid MAC address.

### Check ARP table with this command:

### arp -a

### The entry for 192.168.0.100 should be static.

### Test connectivity with:

### ping 192.168.0.100

### Results

### 改地址后

*Fig. EX1. 1: Now the ARP table has an entry that maps 192.168.0.100 to a user configured MAC address, which is marked as "static".*

### 超时显示

*Fig. EX1. 2: 0% packet loss while pinging to 192.168.0.100 indicating no activity can be performed.*

### Analysis

### When the mapping of IP to MAC is wrong, all packets are sent to a false physical address. The addressing host never sees it, so they can't communicate.

### This experiment shows one of the reasons why ARP entries need to be accurate. Incorrect entry, whether intentional or accidental, interferes with communication. In real-world networks, this could result in attempts for ARP spooﬁng/poisoning attacks that may cause denial-of-service or man-in-the-middle attack.

### Conclusion

### Inserting an invalid ARP entry will prevent a successful conversation with the target host.

### This particular experiment shows the weak point in ARP and its risk when the ARP cache has been altered.

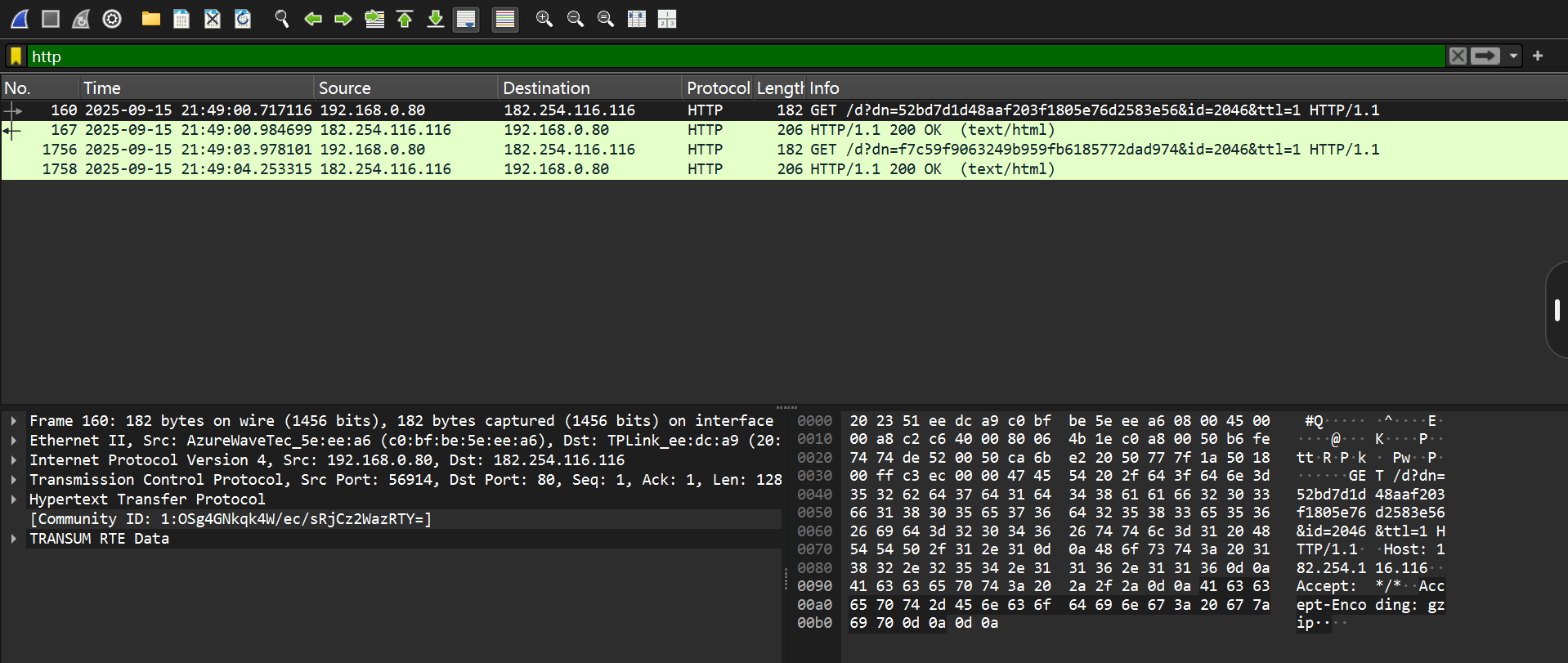
# Appendix

### 未抓包前的列表3

*Figure 2 Unfiltered Wireshark packet listing showing TCP, TLS, and IGMPv3 protocols.*

### P1 http滤波后

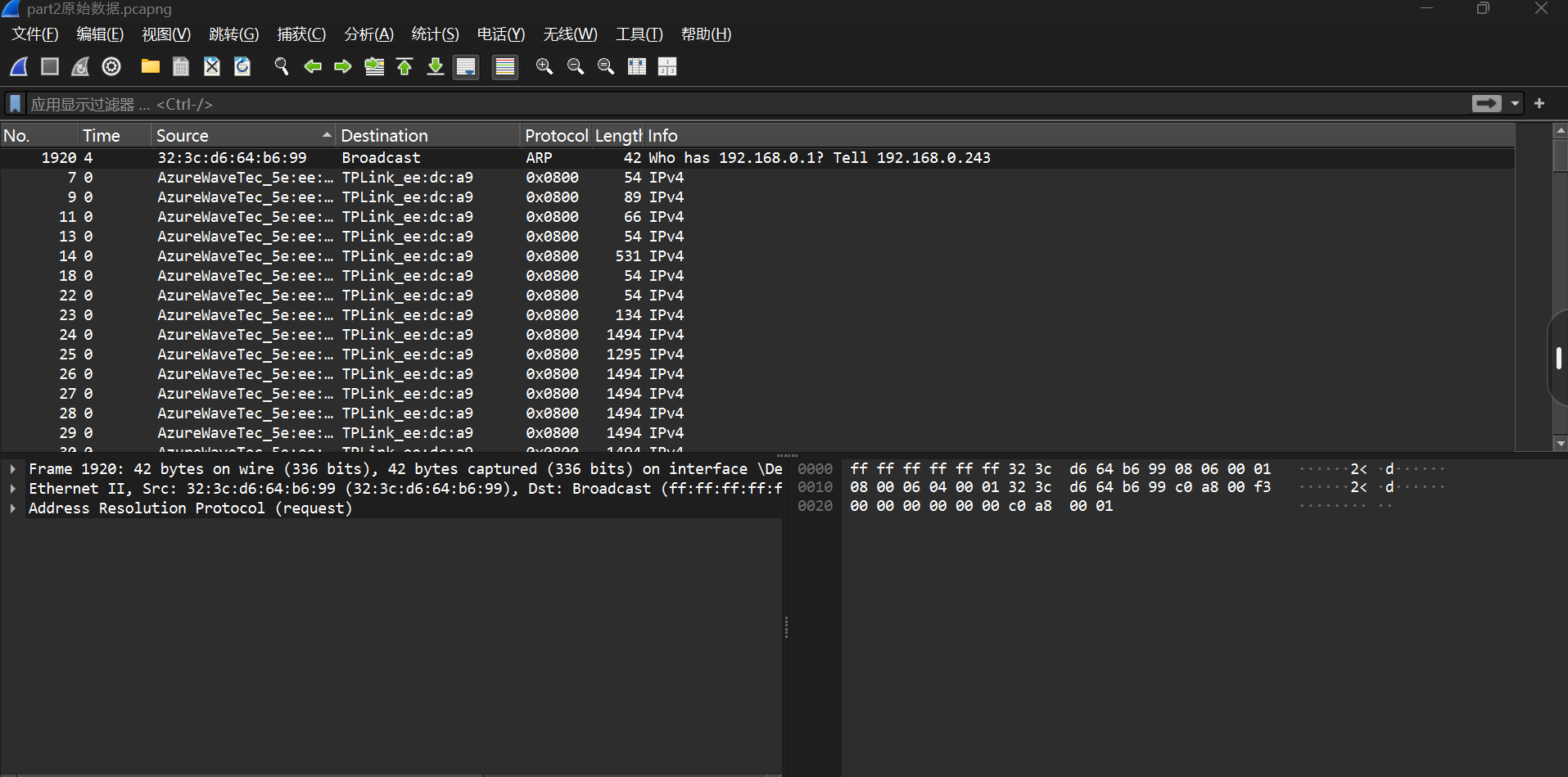
*Figure 3 HTTP display filter applied, highlighting the GET request and 200 OK response.*



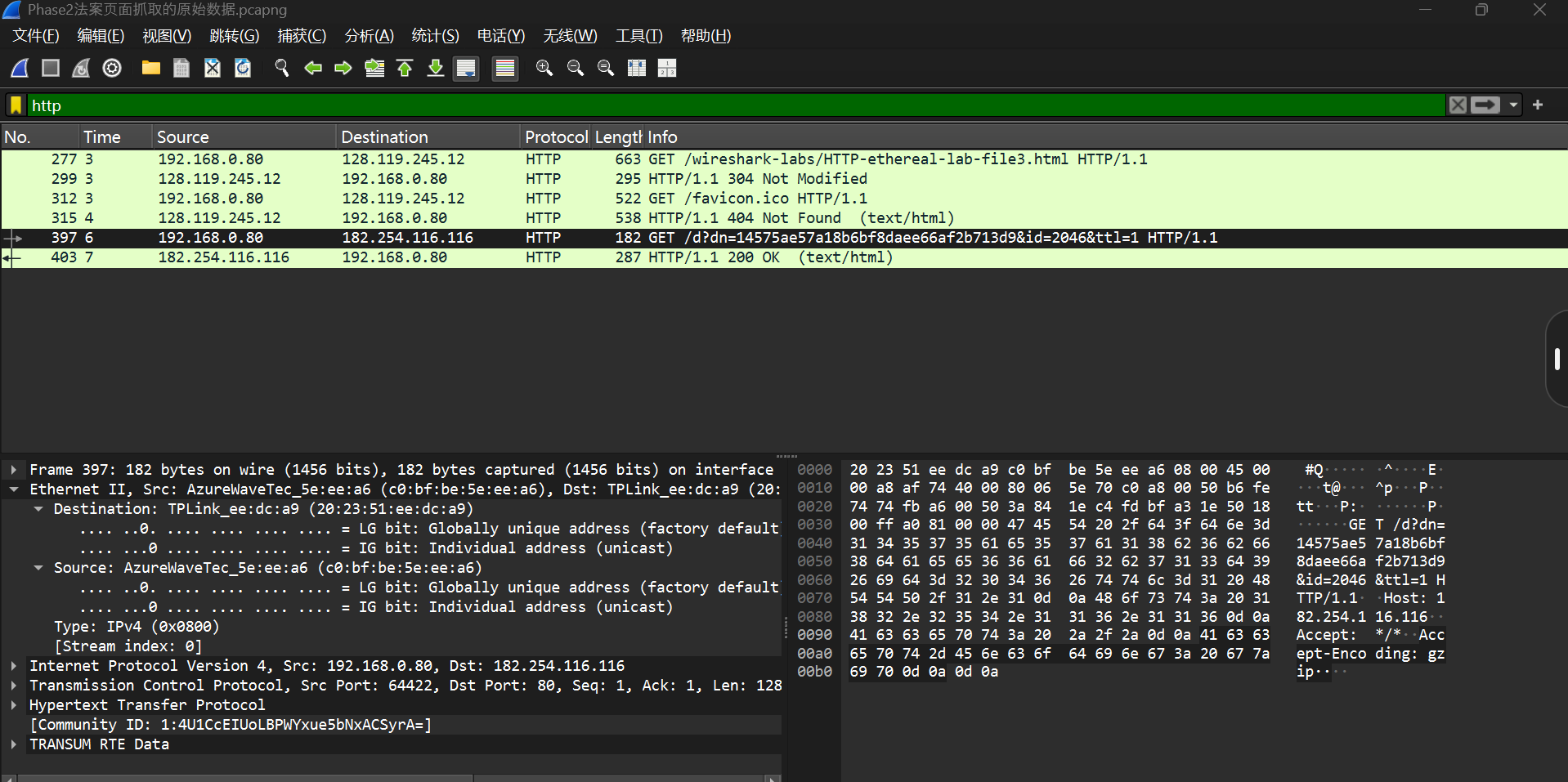
*Figure 4Time calculation between GET and OK packets (≈ 0.267 seconds).*

**

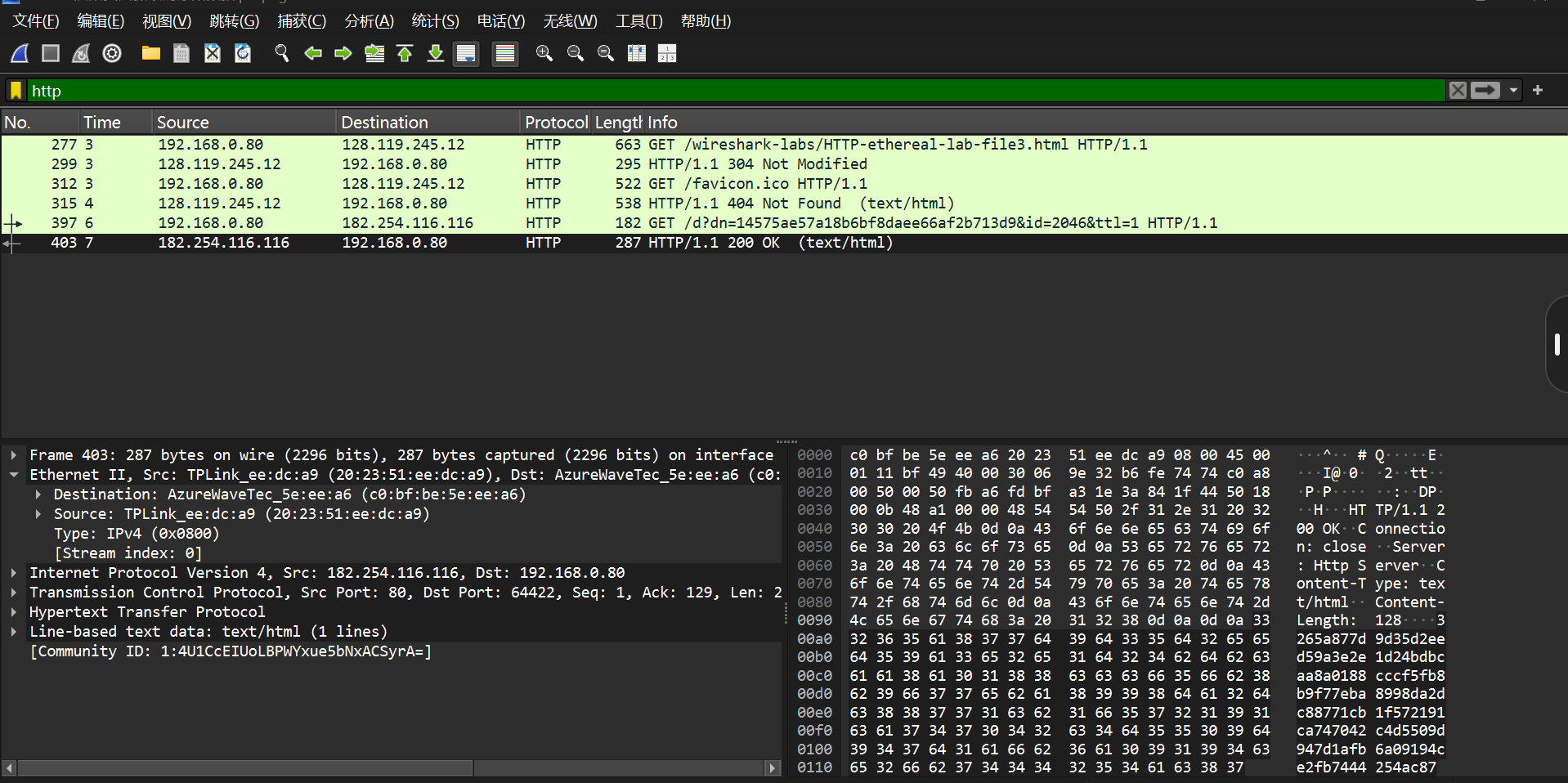
*Figure 5 Detailed packet printouts of the HTTP GET and 200 OK messages.*

**

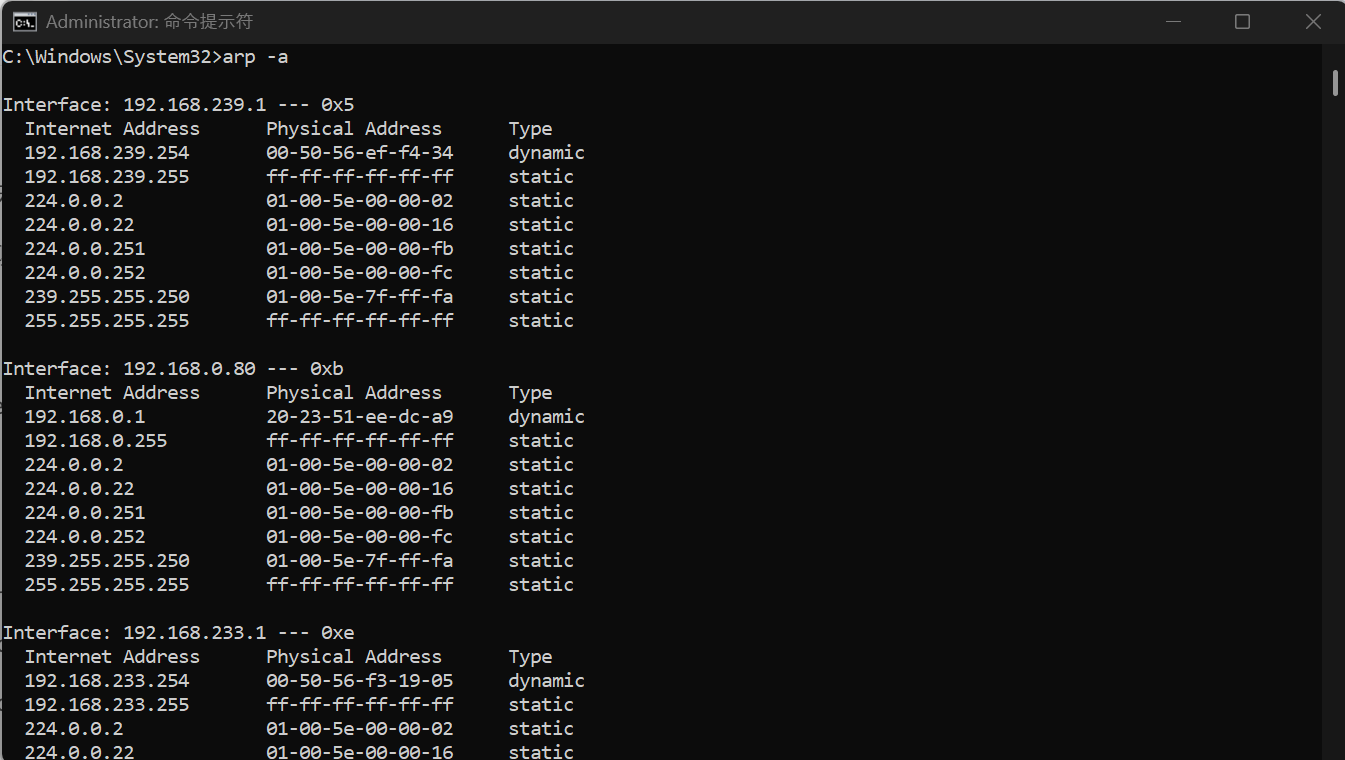
*Figure 6 Wireshark capture after applying IP filter, showing only packets related to the specified IP addresses*.



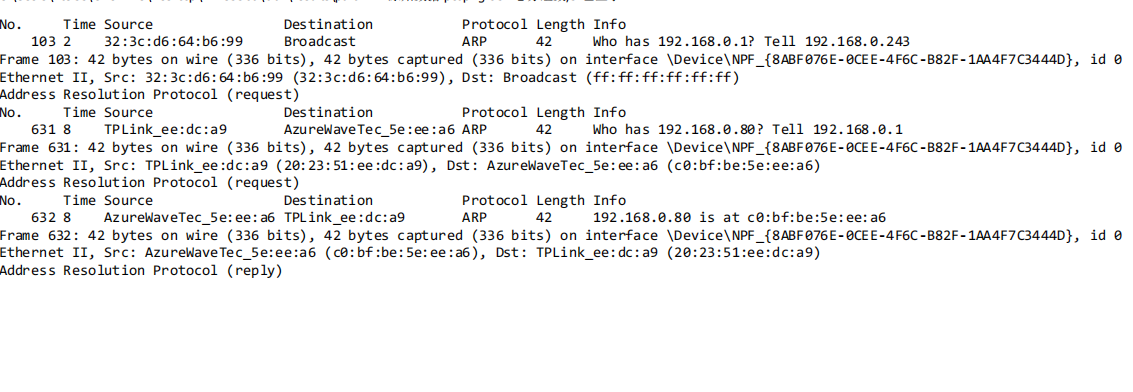
*Figure 7 Ethernet frame of an HTTP GET request captured in Wireshark.*

**

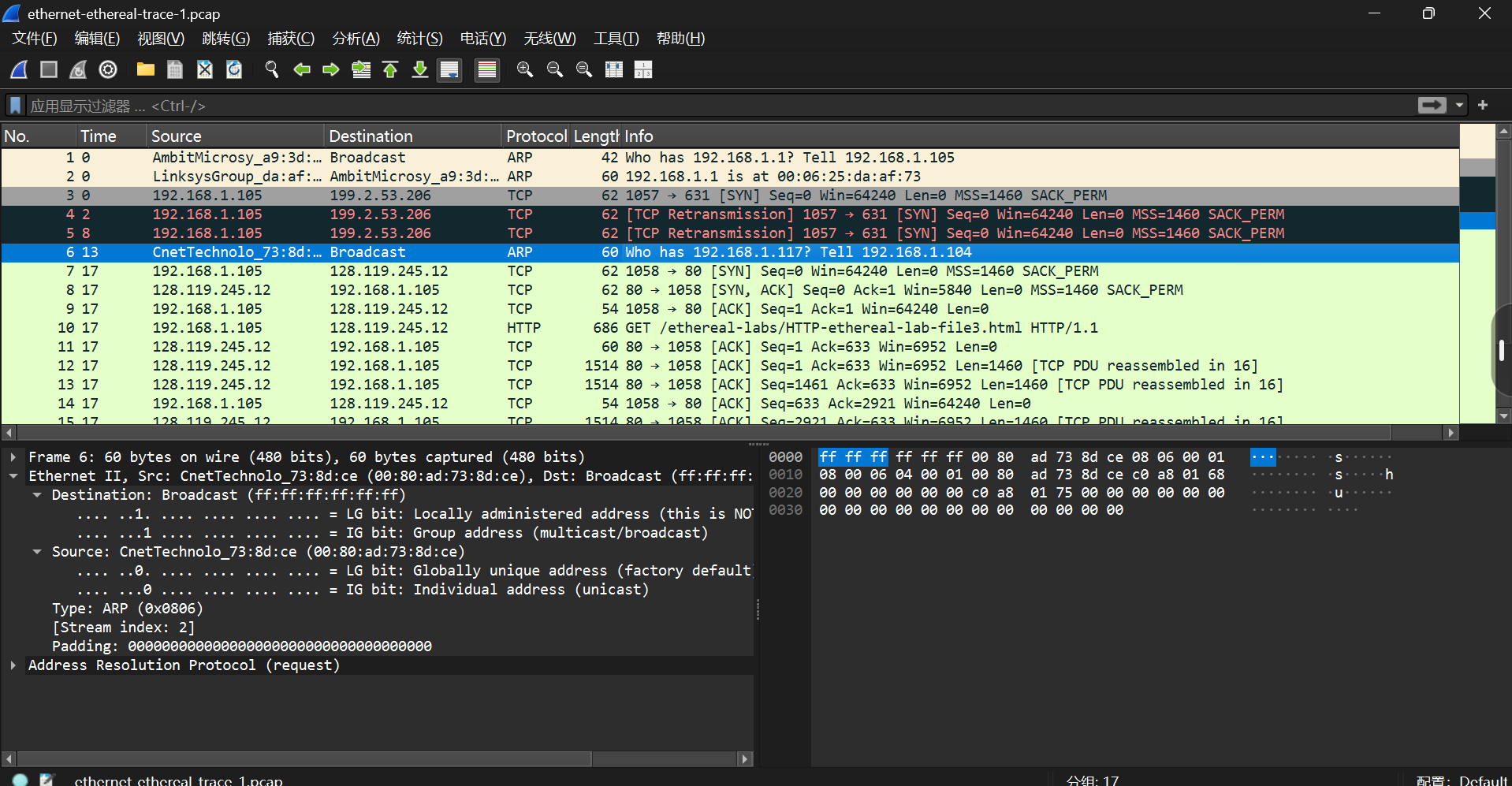
*Figure 8 HTTP 200 OK response observed, confirming successful communication.*

**

*Figure 9 ARP cache table displayed using arp -a command before modification.*

**

*Figure 10 ARP request and reply messages captured in Wireshark.*



*Figure 11 Example of ARP request (Frame 6) from the provided ethernet-ethereal-trace-1.pcap trace file.*