Download application & Configuration and Study of a computer network

João António Morgado Parada (up201405280@up.pt)
Tiago Alexandre Rebelo Oliveira (up202009302@up.pt)

Redes de Computadores - 3LEIC11

23 December 2024

Summary

This report describes the development and analysis of an FTP client application in C. The application enables downloading files from FTP servers using standard FTP commands, including authentication, passive mode, and binary file transfer. The report also includes network configuration, packet analysis using Wireshark, and an evaluation of the application and experiments conducted during the project.

Introduction

The primary objective of this work is to deepen understanding of network protocols and their practical applications through two main components: developing an FTP download application and conducting network experiments. The development of the FTP client emphasizes the implementation of core networking concepts such as the client-server model, socket programming, and the FTP protocol, providing hands-on experience with TCP/IP communication and protocol behaviors outlined in RFC standards.

The experiments complement this by exploring real-world network configurations, including routing, NAT, and bridge implementations, while analyzing packet flows and protocol interactions using tools like Wireshark.

Part 1 - Download application

Architecture of the Download Application

The FTP client application is designed to allow users to download files from FTP servers using the File Transfer Protocol (FTP) as defined in RFC 959. The architecture of the application consists of the following key components:

- **DNS Resolution**: The hostname of the FTP server is resolved to an IP address using the `gethostbyname` function.
- **Control Connection**: A TCP control connection is established with the server on port 21. This connection is used to send FTP commands (`USER`, `PASS`, `PASV`, `RETR`) and receive server responses.
- **Data Connection**: The client switches to passive mode (`PASV` command), and a second TCP connection is established to transfer file data. The data connection uses the IP and port provided by the server in response to the `PASV` command.
- **File Transfer**: The client sends the `RETR` command to initiate the file download. Files are downloaded in binary mode (`TYPE I`) and saved locally.

Report of a successful download

To download a file the user has to use the command `download {ftp_url}`, where `{ftp_url}` can have one of the following formats:

- `ftp://<user>:<password>@<host>/<path>;`
- `ftp://<host>/<path>;`

In the case of the anonymous login, the application assumes anonymous login credentials `user = anonymous`, `password = guest`

Here is the screenshot of the Wireshark logs showing the FTP packets captured when we run the command `download ftp://rcom:rcom@ftp.netlab.fe.up.pt/pipe.txt`

192	195.558014799	172.16.1.10	172.16.10.1	FTP	116	Response: 220 ProFTPD Server (Debian) [::ffff:172.16.1.10]
194	195.558142958	172.16.10.1	172.16.1.10	FTP	77	Request: USER rcom
196	195.559104598	172.16.1.10	172.16.10.1	FTP	98	Response: 331 Password required for rcom
197	195.559160611	172.16.10.1	172.16.1.10	FTP	77	Request: PASS rcom
199	195.734850715	172.16.1.10	172.16.10.1	FTP	112	Response: 230-Welcome, archive user rcom@172.16.1.19 !
200	195.734865730	172.16.1.10	172.16.10.1	FTP	69	Response:
201	195.734872016	172.16.1.10	172.16.10.1	FTP	112	Response: The local time is: Sat Dec 14 08:34:02 2024
202	195.734932149	172.16.1.10	172.16.10.1	FTP	142	Response:
203	195.734938924	172.16.1.10	172.16.10.1	FTP	129	Response: please report them via e-mail to <root@ftp.netlab.fe.up.pt>.
204	195.734983762	172.16.1.10	172.16.10.1	FTP	94	Response:
206	195.735241405	172.16.10.1	172.16.1.10	FTP	72	Request: PASV
208	195.736025021	172.16.1.10	172.16.10.1	FTP	115	Response: 227 Entering Passive Mode (172,16,1,10,168,55).
212	195.736616155	172.16.10.1	172.16.1.10	FTP	74	Request: TYPE I
213	195.737177118	172.16.1.10	172.16.10.1	FTP	85	Response: 200 Type set to I
214	195.737219302	172.16.10.1	172.16.1.10	FTP	81	Request: RETR pipe.txt
215	195.738084073	172.16.1.10	172.16.10.1	FTP	132	Response: 150 Opening BINARY mode data connection for pipe.txt (418 bytes)
216	195.738306377	172.16.1.10	172.16.10.1	FTP-DA…	484	FTP Data: 418 bytes (PASV) (TYPE I)
229	206.539869664	172.16.1.10	172.16.10.1	FTP	89	Response: 226 Transfer complete
232	206.540015492	172.16.10.1	172.16.1.10	FTP	72	Request: QUIT
233	206.540643782	172.16.1.10	172.16.10.1	FTP	80	Response: 221 Goodbye.

By analyzing these logs, we can clearly see the entire flow of a file download using the FTP protocol. The process begins with the client establishing a control connection and sending the `USER` and `PASS` commands to authenticate with the server. Once the authentication is successful, as indicated by the `331` and `230` responses, the client switches to passive mode using the `PASV` command. The server responds with the `227` code, providing the IP address and port for the data connection. The client then sets binary mode with the `TYPE I` command (response `200`) to ensure accurate file transfer for non-text files (in this case this step would not be necessary, but we found that other files would become corrupted if we did not include it). The `RETR` command is issued to request the file (`pipe.txt`), and the server responds with `150`, indicating that the data connection is open and ready for transfer. The file data is transmitted (418 bytes in this case), and the server confirms the completion of the transfer with the `226` response. Finally, the client sends the `QUIT` command to terminate the session, and the server acknowledges with the `221` response, gracefully closing the connection.

Part 2 - Network configuration analysis

<u>Experience 1 - Configure an IP Network</u>

Network Architecture: The network consists of two devices (`tux13` and `tux14`) connected via a switch within the subnet `172.16.10.0/24`. Both devices are configured with unique IP addresses.

Experiment Objectives: Configure the IP network interfaces on `tux13` using `ifconfig eth1 172.16.10.1/24` and `tux14` using `ifconfig eth1 172.16.10.254/24`, verify connectivity using `ping`, and analyze ARP table behavior using `arp -a`.

Relevant Logs: Captured ARP requests and replies for resolving MAC addresses, as well as ICMP packets (echo requests and replies) for verifying connectivity. The logs demonstrate proper device communication over the network.

Analysis: The ARP requests captured in the logs show how IP-to-MAC address resolution occurs before initiating communication. ICMP packets confirm successful connectivity between `tux13` and `tux14`. The experiment highlights the importance of ARP in facilitating communication within a subnet. (See annexed logs for detailed packet captures.)

What are the ARP packets and what are they used for?

- ARP (Address Resolution Protocol) is a protocol used to map an IP address into a MAC address inside the same local network (LAN). When a device wants to send a packet to another device on the same network he needs its MAC address. If he does not have it however, he can send an ARP Request packet where he can find out the MAC address corresponding to the IP.

What are the MAC and IP addresses of ARP packets and why?

- ARP Request packets have the MAC address of the device that sent the request as the source, and FF:FF:FF:FF:FF:FF (broadcast) as the destination. The source IP address used is the transmitter's address while the destination IP address is the one which we wanna map.
- An ARP Reply packet includes the MAC address of the responding device as the source, while the destination MAC address is that of the requester. The source IP is the responder's IP, and the destination IP is the requester's address.

What packets does the ping command generate?

- The ping command uses the ICMP (Internet Control Message Protocol) protocol and it generates two types of packets:
    - Echo Request: Sent by the device who 'pings' to the destination;
    - Echo Reply: Answer sent by the destination device to the one who sent the ping;

What are the MAC and IP addresses of the ping packets?

- The Echo Request packet uses as the source MAC address the one of the device who sends the ping and as the MAC destination address, the MAC of the destination's device on the same network (or of the gateway if in separate networks). The source IP address corresponds to the address who has sent the ping and the IP destination is the one specified on the ping command.
- For an Echo Reply, the source MAC and IP addresses are those of the responding device, and the destination MAC and IP addresses belong to the original sender.

How to determine if a receiving Ethernet frame is ARP, IP, ICMP?

- Using the tool, Wireshark it's easy since you just need to check the "Protocol" field of the receiving packets.

How to determine the length of a receiving frame?

- Also using Wireshark, just see the "Length" field.

What is the loopback interface and why is it important?

- A loopback interface is a virtual network that a system uses to talk to itself. It's really important since it allows the device to test his own network stack without a physical interface being active. Additionally, it is used by services that need local connectivity and plays a critical role in software development and diagnostics.

Experience 2 - Implement two bridges in a switch

Network Architecture: The experiment involved a switch configured with two separate bridges (`bridge10` and `bridge11`), connecting devices in isolated networks. `Tux13` and

`Tux14` were connected to `bridge10`, while `Tux12` was connected to `bridge11`. This setup was used to analyze traffic isolation and the behavior of broadcast packets across bridges.

Experiment Objectives:

1. Verify connectivity between devices in the same bridge (e.g., `Tux13` to `Tux14`).
2. Demonstrate lack of connectivity between devices in different bridges (e.g., `Tux13` to `Tux12`).
3. Analyze the propagation of broadcast pings across bridges and evaluate received responses.

Relevant Logs:

1. **Tux13 pinging Tux14**: Successful communication observed with ICMP echo requests and replies (see logs).
2. **Tux12 Unreachable**: "Network is unreachable" message confirms lack of connectivity between devices in separate bridges (see logs).
3. **Broadcast Pings**: Captured ARP requests and broadcast ICMP packets, showing that devices within the same bridge (Tux13 ping broadcast logs, Tux14 broadcast reception) receive and respond to broadcasts, while devices in separate bridges do not (Tux12 ping broadcast logs).

Analysis:
The experiment demonstrates the functionality of bridges in isolating traffic between subnets while allowing intra-bridge communication. Broadcast traffic is limited to devices within the same bridge, confirming proper bridge isolation. These findings validate the use of bridges in sementing network traffic and improving security by limiting broadcast domain sizes.

How to configure bridge10?

- In order to configure a bridge we have to use the Mikrotik switch. A bridge connects multiple interfaces in one single broadcast. To create a bridge we just need to execute:
    - /interface bridge add name=bridge10
- Next, we need to connect an interface to our bridge10:
    - /interface bridge port add bridge=bridge10 interface=ether1
- In the end, we can confirm if our job is done by executing two commands:
    - Check the created bridges with: /interface bridge print
    - Check the ports associated with the bridge: /interface bridge port print

How many broadcast domains are there? How can you conclude it from the logs?

- A broadcast domain is the set of devices that are receiving packets sent by broadcast (ARP or ICMP packets) without needing a router.
- The interfaces were divided into different bridges (tux13 and tux14 in bridge10, tuxY2 in bridge11). So, right now there are two broadcast domains: bridge10 and bridge11.
- What we can conclude from the logs is:
    - Packets that were sent over a bridge were not showing up in interfaces associated with other bridges.

Experiment 3 - Configure a Router in Linux

Network Architecture: In this experiment, `Tux14` was configured as a router connecting two subnets, `172.16.10.0/24` and `172.16.11.0/24`. Devices `Tux13` and `Tux12` were placed in different subnets and used the router (`Tux14`) as a gateway to enable communication between them.

Experiment Objectives:

1. Configure `Tux14` as a router by enabling IP forwarding.
2. Test connectivity between `Tux13` and other devices in different subnets.
3. Capture traffic on `Tux14` to analyze the forwarding behavior across its interfaces.

Relevant Logs:

1. Tux13 Pinging Other Interfaces:
   - Successful ICMP requests and replies demonstrate routing through `Tux14`.
   - ARP packets confirm gateway resolution for packets destined to other subnets.
2. Tux14 Capturing Both Network Interfaces:
   - Captured packets on `eth1` and `eth2` of `Tux14` show routing behavior, with incoming traffic from one subnet being forwarded to the other.
   - Logs highlight that `Tux14` modifies the source/destination MAC addresses as part of the forwarding process.

Analysis:
The experiment validates the use of `Tux14` as a router between two subnets. IP forwarding enables communication between devices in different subnets by directing traffic through the gateway. The logs confirm correct routing behavior, with ICMP packets being forwarded across interfaces. ARP logs further demonstrate how `Tux14` resolves and updates MAC addresses for devices in each subnet.

What routes are there in the tuxes? What are their meaning?

- There are **directly connected networks** which are routes for the subnets, to which the interfaces are directly connected.
  - 172.16.10.0/24 -> associated with tuxY3 and tuxY4
  - 172.16.11.0/24 -> associated with tuxY2
- There is the **default route**, which is a route pointing to the next router looking for packets destined for networks outside the directly connected ones.

What information does an entry of the forwarding table contain?

- A forward table entry contains the following information:
  - **Destination IP or Network:** IP address or subnet the entry applies to
  - **Gateway:** Address which we want to send packets to (if necessary)
  - **Interface:** The network interface to use for forwarding packets
  - **Metric:** A value that tells the preferred route
  - **Flag:** Displays specific characteristics such as if the route is up or if it is a gateway route

- **Genmask:** Defines the size

What ARP messages, and associated MAC addresses, are observed and why?

- **ARP Request:** Is sent when a device wants to resolve a MAC address of another device's IP address. The sender's MAC address and IP are included, while the destination MAC address is broadcast (FF:FF:FF:FF:FF:FF).
  - Example: A tux sends "Who has 172.16.Y0.1? Tell 172.16.Y0.2".
- **ARP Reply:** Sent by the device that owns the desired IP address, and provided its MAC address. The source MAC and IP address belong to the one who answers, the destination MAC and IP address belong to the one who requested.
  - Example: 172.16.Y0.1 answers "It is at AA:BB:CC:DD:EE:FF".

What ICMP packets are observed and why?

ICMP is used to make diagnoses and communicate errors between devices.

- **Echo Request** packet:
  - Sent by the tux who starts a ping command
  - Motive: To verify the connectivity with another device.
- **Echo Reply** packet:
  - Sent by the destination answering the Echo Request packet.
  - Motive: To confirm that the communication was established.

What are the IP and MAC addresses associated to ICMP packets and why?

- **Echo Request:**
  - **Source MAC/IP:** From the device that started the ping command.
  - **Destination MAC/IP:** Depends on the network configuration:
    - If in different subnets, the destination MAC is the same as the router's interface MAC;
    - If within the same subnet, the destination MAC is the recipient device's MAC;

Experiment 4 - Configure a Commercial Router and Implement NAT

Network Architecture: In this experiment, `Rc` (a commercial router) connects two subnets: `172.16.10.0/24` and `172.16.11.0/24`. NAT (Network Address Translation) is enabled on the router to facilitate communication between the subnets.

Experiment Objectives:

1. Configure `Rc` to act as a router between the subnets and enable NAT.
2. Verify connectivity between `Tux12` and `Tux13` through the router.
3. Analyze the routing behavior with tools like `ping` and `traceroute`.
4. Demonstrate how NAT affects communication between subnets.

Relevant Logs:

1. **Tux13 Pinging All Network Interfaces**:
   - ICMP echo requests and replies from different interfaces confirm connectivity within the network.
2. **Tux12 Pinging Tux13 Through Rc**:
   - ICMP echo requests and replies confirm successful communication across the subnets via the router.

        - Each response includes a "Redirect Host" message from 172.16.11.254, indicating that routing adjustments are being made by the router..
3. **Tux12 Traceroutes**:
    - Traceroute logs display the 2 routing paths: One using Rc as the gateway to the other subnet, and one using Tux14.

Analysis: The experiment validates the role of NAT in enabling communication between private subnets and external networks. The captured logs confirm proper routing and address translation through the commercial router. NAT ensures that devices in 172.16.11.0/24 can communicate with devices in 172.16.10.0/24 while preserving network security and address space efficiency.

How to configure a static route in a commercial router?

- In order to configure a static route we need to following a few steps:
    1. Define the destination subnet;
    2. Specify the next-hop IP address or gateway for that subnet;
    3. Associate the route with an interface;
- With the GTKterm:
    1. Add a static route:
        - /ip route add dst-address=<DESTINATION>/<MASK> gateway=<GATEWAY>
    2. List configured routes:
        - /ip route print
- We can test the connectivity with commands like ping and traceroute.

What are the paths followed by the packets, with and without ICMP redirect enabled, in the experiments carried out and why?

- **Without ICMP Redirect:** The commercial router's default route setting routes packets from tux13 to tux12. Since tux14 is specifically designated as the gateway, the router handles the packets even though tux13 and tux12 can connect directly through it. The road gains one additional hop as a result.
- **With ICMP Redirect:** The router notifies tux13 that tux14 offers a more direct path when ICMP redirect is enabled. Consequently, packets are delivered through tux14 directly between tux13 and tux12, avoiding the commercial router. This behavior lowers latency and optimizes the path.

How to configure NAT in a commercial router?

- Enabling NAT on the interface:
    - /ip firewall nat add chain=srcnat action=masquerade out-interface=ether1

- Verify NAT rules:
    - /ip firewall nat print

What does NAT do?

- Network address translation, or NAT, converts private IP addresses used on internal networks into distinct public IP addresses so that external networks can communicate with one another.

What happens when tux13 pings the FTP server with the NAT disabled? Why?

- When NAT gets disabled tuxY3 cannot receive replies from the FTP server. That's because the packets sent by it use the private IP address as the source. On top of this, the ftp server responds to the private IP, which cannot be routable to the public. Without NAT, routers cannot translate private IPs into public IPs to establish communication with external sources.

Experiment 5 - DNS

Network Architecture:
This experiment focuses on enabling and testing DNS functionality within the network. `Tux12` is configured to use a specific DNS server (`10.227.20.3`) for resolving domain names. The functionality is validated by testing name resolution for internal and external hosts (e.g., `google.com`).

Experiment Objectives:

1. Configure `Tux12` to use the correct DNS server.
2. Verify DNS functionality by resolving hostnames to IP addresses.
3. Test connectivity to external domains using `ping` to validate DNS resolution.

Relevant Logs:

1. **DNS Configuration**:
   - The `/etc/resolv.conf` file confirms the correct setup of the nameserver (`10.227.20.3`) and search domains.
   - This configuration ensures that DNS queries can be processed by the designated server.
2. **Testing Name Resolution**:
   - Pinging `google.com` demonstrates successful hostname resolution, with the hostname translating into its corresponding IP address.
3. **Pinging External Domain**:
   - Logs show successful DNS resolution for `google.com`, followed by ICMP echo requests and replies.
   - The `Wireshark` capture highlights DNS query and response packets exchanged between `Tux12` and the DNS server (`10.227.20.3`), confirming proper DNS operation.

Analysis:
This experiment validates the role of DNS in facilitating name resolution within a network. The logs and packet captures confirm that DNS queries are correctly routed to the specified server and responses are processed as expected. This ensures communication with hosts using domain names rather than IP addresses.

How to configure the DNS service in a host?
1. Locate and edit the /etc/resolv.conf:
   - Example: sudo nano /etc/resolv.conf
2. Add the DNS server addresses:
   - nameserver <DNS-SERVER>

What packets are exchanged by DNS and what information is transported?

- DNS Query: Client sends a query to a DNS server to resolve a domain name into IP address (or vice versa). "What is the IP for www.example.com?"
- DNS Response: DNS server replies with the requested information. "The IP address for www.example.com" is 01.234.567.89.
- Information passed:
    - **Header**: Contains the query ID, flags and other sections;
    - **Question Section**: Has the domain name that was queried and the type of record requested;
    - **Answer Section:** Displays the resolved data, such as the IP address of the domain;
    - **Authority Section:** Lists authoritative name servers for the queried domain;
    - **Additional Section:** Provides extra information;

Experiment 6 - TCP connections

Network Architecture:
In this experiment, the FTP client developed earlier is tested in a multi-device network setup, involving `tux13` (FTP client), `Rc` (router with NAT enabled), and `ftp.netlab.fe.up.pt` (FTP server). The network spans multiple subnets interconnected via NAT. The goal is to analyze TCP connections, their phases, and mechanisms, including ARQ and congestion control, during a file download and in scenarios with competing traffic.

Experiment Objectives:

1. Test the FTP download application in a real-world network environment.
2. Analyze the TCP phases—connection establishment, data transfer, and termination—using Wireshark.
3. Observe and understand TCP mechanisms, including ARQ and congestion control.
4. Examine the impact of competing downloads on TCP throughput and behavior.

Wireshark Capture: TCP control and data connections are captured:

- **Connection Establishment**: The three-way handshake (`SYN`, `SYN-ACK`, `ACK`) is clearly observed, marking the initiation of the TCP connection.
- **Data Transfer**: Logs display FTP commands (e.g., `USER`, `PASS`, `RETR`) in the control connection and the subsequent data transfer.
- **Connection Termination**: The four-segment connection termination (`FIN`, `ACK`) is logged, indicating proper teardown of the TCP session.

Competing Downloads: The files that were available on ftp.netlab.fe.up.pt were either too small (crab.mp4 download finished before we could start it on the other tux) or too large (ubuntu.iso produced very large logs that we were unable to retrieve to an external device or upload for future examination) so we could not conclude anything. We tried to reproduce this at home and the conclusion we reached was that the transfer speed of the first connection gets approximately halved by the TCP congestion control mechanism.

Analysis:
This experiment highlights TCP's ability to reliably transfer data even under adverse conditions such as packet loss or congestion.

How many TCP connections are opened by your FTP application?

- There are two TCP connections on a FTP application:
    1. **Control Connection:** Used to send and receive FTP commands;
    2. **Data Connection:** Used to transfer files between the client and server;

In what connection is transported the FTP control information?

- The FTP control information is transported in the **Control Connection.** This remains open during the session and is used to handle commands such as USER,PASS,RETR,etc…

What are the phases of a TCP connection?

1. **Connection Establishment (3-way Handshake):**
    - **SYN:** Client send a synchronization packet;
    - **SYN-ACK:** Server acknowledges and sends a SYN;
    - **ACK:** Client acknowledges completing the "handshake";
2. **Data Transfer:**
    - Data is traded between the client and the server.
3. **Connection Termination (4-way Handshake):**
    - **FIN:** Request to terminate the connection;
    - **ACK:** The other side acknowledges the FIN;
    - **FIN:** The second side sends a FIN;
    - **ACK:** The first side acknowledges, completing the termination;

How does the ARQ TCP mechanism work? What are the relevant TCP fields? What relevant information can be observed in the logs?

1.

- TCP used ARQ (Automatic Repeat Request) to reliably deliver the data;
- Its sent an ACK for every packet that is received;
- If there is not ACK being received within the timeout, the sender will retransmitter the packet;

2.

Sequence Number: Identifies the order of bytes in the data stream;
Acknowledgment Number: Confirms the receipt of data;
Window Size: Indicates the buffer capacity available
Flag: ACK,FIN,SYN,etc…

3.

- Adjustments in window size during flow control;
- Duplicate ACKs signaling packet loss;
- Retransmissions due to timeout;

How does the TCP congestion control mechanism work? What are the relevant fields?

- TCP congestion control mechanism operates in four phases:
    1. **Slow Start:** The congestion window grows exponentially until it reached a threshold or a packet loss occurs;
    2. **Congestion Avoidance:** After the threshold is reached, the window increases linearly trying not to overload the network;
    3. **Fast Retransmit:** If three duplicate ACKs are received, TCP assumes a packet is lost and retransmits it;
    4. **Fast Recovery:** The congestion window is reduced, and TCP shifts to congestion avoidance mode to gradually recover;

How did the throughput of the data connection evolve along the time? Is it according to the TCP congestion control mechanism?

- At the beginning, during the Slow Start phase, throughput grows rapidly (exponentially) as the congestion window (cwnd) increases. Once the connection enters the Congestion Avoidance phase, throughput grows more slowly (linearly). If the connection detects packet loss, throughput drops sharply because the congestion window is reduced (often halved). Afterward, throughput gradually recovers. This pattern of growth and occasional drops can be observed in graphs of throughput over time.

Is the throughput of a TCP data connection disturbed by the appearance of a second TCP connection? How?

- Yes, the throughput of a TCP data connection is affected when a second connection starts sharing the same network. Both compete for the existing bandwidth, while TCP adjusts the congestion window for each to ensure fairness, typically the result is each connection being reduced to half of the bandwidth. Over time, the network is stabilized and both connections reach a steady state of share throughput as seen in performance graphs.

Conclusions

In conclusion, this lab provided a comprehensive understanding of networking principles and protocols, emphasizing the development of a simple FTP client and its interaction with TCP/IP. Through practical experiments, we analyzed network configurations, routing, and bridging while observing key TCP mechanisms such as ARQ and congestion control. The successful execution and analysis of FTP file transfers demonstrated the reliability of TCP and the importance of precise network configurations. Overall, the lab reinforced theoretical concepts with hands-on experience, enhancing our ability to diagnose and optimize network communication.

Annexes

Download Application Code:

1.1 - main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <libgen.h>
#include "ftp_client.h"

int main(int argc, char *argv[]) {
    if (argc != 2) {
```

```
        fprintf(stderr, "Usage: %s
ftp://[<user>:<password>@]<host>/<url-path>\n", argv[0]);
        return -1;
    }

    char user[64], password[64], host[256], path[256];
    if (parse_ftp_url(argv[1], user, password, host, path) < 0) {
        fprintf(stderr, "Error: Invalid FTP URL format.\n");
        return -1;
    }

    printf("Starting FTP download...\n");

    // Connect to FTP server
    int control_sock = connect_to_server(host, 21);
    if (control_sock < 0) {
        fprintf(stderr, "Error: Unable to connect to server.\n");
        return -1;
    }
    printf("Connected to server: %s\n", host);

    // Read initial greeting
    char greeting[2048];
    if (read_ftp_response(control_sock, greeting, sizeof(greeting)) <=
0) {
        fprintf(stderr, "Error: Failed to read server greeting.\n");
        close(control_sock);
        return -1;
    }
    printf("Server greeting: %s", greeting);

    // Login to FTP server
    if (ftp_login(control_sock, user, password) < 0) {
        fprintf(stderr, "Error: Login failed.\n");
        close(control_sock);
        return -1;
    }
    printf("Logged in as: %s\n", user);

    // Enter passive mode
    char ip[16];
    int port;
    if (ftp_enter_passive_mode(control_sock, ip, &port) < 0) {
```

```
        fprintf(stderr, "Error: Failed to enter passive mode.\n");
        close(control_sock);
        return -1;
    }
    printf("Entered passive mode: %s:%d\n", ip, port);

    // Connect to data socket
    int data_sock = connect_to_server(ip, port);
    if (data_sock < 0) {
        fprintf(stderr, "Error: Unable to establish data
connection.\n");
        close(control_sock);
        return -1;
    }
    printf("Data connection established.\n");
    char *filename = basename(path);

    // Retrieve file
    if (ftp_retrieve_file(control_sock, data_sock, path, filename) < 0)
{
        fprintf(stderr, "Error: Failed to retrieve file.\n");
        close(data_sock);
        close(control_sock);
        return -1;
    }
    printf("File downloaded successfully: %s\n", filename);

    // Close connections
    close(data_sock);
    ftp_quit(control_sock);
    close(control_sock);
    printf("FTP session closed.\n");

    return 0;
}
```

1.2 - ftp_client.h

```
#ifndef FTP_CLIENT_H
#define FTP_CLIENT_H
```

```c
#include <stddef.h>

int read_ftp_response(int sock, char *buf, size_t size);
int parse_ftp_url(const char *url, char *user, char *password, char
*host, char *path);
int connect_to_server(const char *hostname, int port);
int ftp_login(int control_sock, const char *user, const char
*password);
int ftp_enter_passive_mode(int control_sock, char *ip, int *port);
int ftp_retrieve_file(int control_sock, int data_sock, const char
*remote_path, const char *local_filename);
void ftp_quit(int control_sock);

#endif
```

1.3 - ftp_client.c

```c
#include "ftp_client.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netdb.h>



// Parses the FTP URL and extracts user, password, host, and path
int parse_ftp_url(const char *url, char *user, char *password, char
*host, char *path)
{
  // Try full format: ftp://user:pass@host/path
  if (sscanf(url, "ftp://%[^:]:%[^@]@%[^/]/%s", user, password, host,
path) == 4)
  {
    return 0;
  }
  // Try anonymous format: ftp://host/path
  if (sscanf(url, "ftp://%[^/]/%s", host, path) == 2)
  {
```

```c
    strcpy(user, "anonymous");
    strcpy(password, "guest");
    return 0;
  }
  return -1;
}

// Connects to the server and returns the socket descriptor
int connect_to_server(const char *hostname, int port)
{
  struct sockaddr_in server_addr;
  struct hostent *host_entry;

  if ((host_entry = gethostbyname(hostname)) == NULL)
  {
    perror("gethostbyname");
    return -1;
  }

  int sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
  {
    perror("socket");
    return -1;
  }

  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;
  memcpy(&server_addr.sin_addr, host_entry->h_addr,
host_entry->h_length);
  server_addr.sin_port = htons(port);

  if (connect(sockfd, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0)
  {
    perror("connect");
    close(sockfd);
    return -1;
  }

  return sockfd;
}
```

```c
// Reads a complete FTP response, handling multi-line replies.
// FTP multi-line replies format:
// xyz-... (one or more lines)
// xyz ... (final line)
int read_ftp_response(int sock, char *buf, size_t size)
{
  memset(buf, 0, size);
  int total = 0;
  char line[1024];
  bool multiline = false;
  char code[4] = {0};

  while (1)
  {
    memset(line, 0, sizeof(line));
    int n = 0;
    // Read one line at a time
    while (n < (int)sizeof(line) - 1)
    {
      int r = read(sock, &line[n], 1);
      if (r <= 0)
      {
        // Connection closed or error
        break;
      }
      if (line[n] == '\n')
      {
        n++;
        break;
      }
      n++;
    }
    if (n == 0)
    {
      // No data read, possibly connection closed
      break;
    }
    line[n] = '\0';

    // Append this line to buf if space allows
    if ((int)strlen(buf) + n < (int)size - 1)
    {
      strcat(buf, line);
```

```
    }

    // If this is the first line, determine if it's multiline
    if (total == 0 && strlen(line) >= 4)
    {
      strncpy(code, line, 3);
      code[3] = '\0';
      // Check if multiline
      if (line[3] == '-')
      {
        multiline = true;
      }
    }

    total += n;

    // Check for end of multiline
    if (strlen(code) == 3 && !multiline)
    {
      // Single line response
      break;
    }

    if (multiline && strncmp(line, code, 3) == 0 && line[3] == ' ')
    {
      // End of multiline response
      break;
    }

    if (!multiline)
    {
      // If not multiline, we read only one line
      break;
    }
  }
  return total;
}

// Logs in to the FTP server
int ftp_login(int control_sock, const char *user, const char *password)
{
  char buffer[2048];
```

```c
  // If user is anonymous and password is "guest", try a common
anonymous password
  char actual_password[256];
  if (strcmp(user, "anonymous") == 0 && strcmp(password, "guest") == 0)
  {
    strcpy(actual_password, "anonymous@example.com");
  }
  else
  {
    strcpy(actual_password, password);
  }

  snprintf(buffer, sizeof(buffer), "USER %s\r\n", user);
  write(control_sock, buffer, strlen(buffer));
  read_ftp_response(control_sock, buffer, sizeof(buffer));

  // Expect a 331 code here
  if (strncmp(buffer, "331", 3) != 0)
  {
    fprintf(stderr, "Login failed (USER step): %s", buffer);
    return -1;
  }

  snprintf(buffer, sizeof(buffer), "PASS %s\r\n", actual_password);
  write(control_sock, buffer, strlen(buffer));
  read_ftp_response(control_sock, buffer, sizeof(buffer));

  // Expect a 230 code here for success
  if (strncmp(buffer, "230", 3) != 0)
  {
    fprintf(stderr, "Login failed (PASS step): %s", buffer);
    return -1;
  }

  return 0;
}

// Enters passive mode and retrieves IP and port for data connection
int ftp_enter_passive_mode(int control_sock, char *ip, int *port)
{
  char buffer[2048];
  char *start;
  int ip1, ip2, ip3, ip4, p1, p2;
```

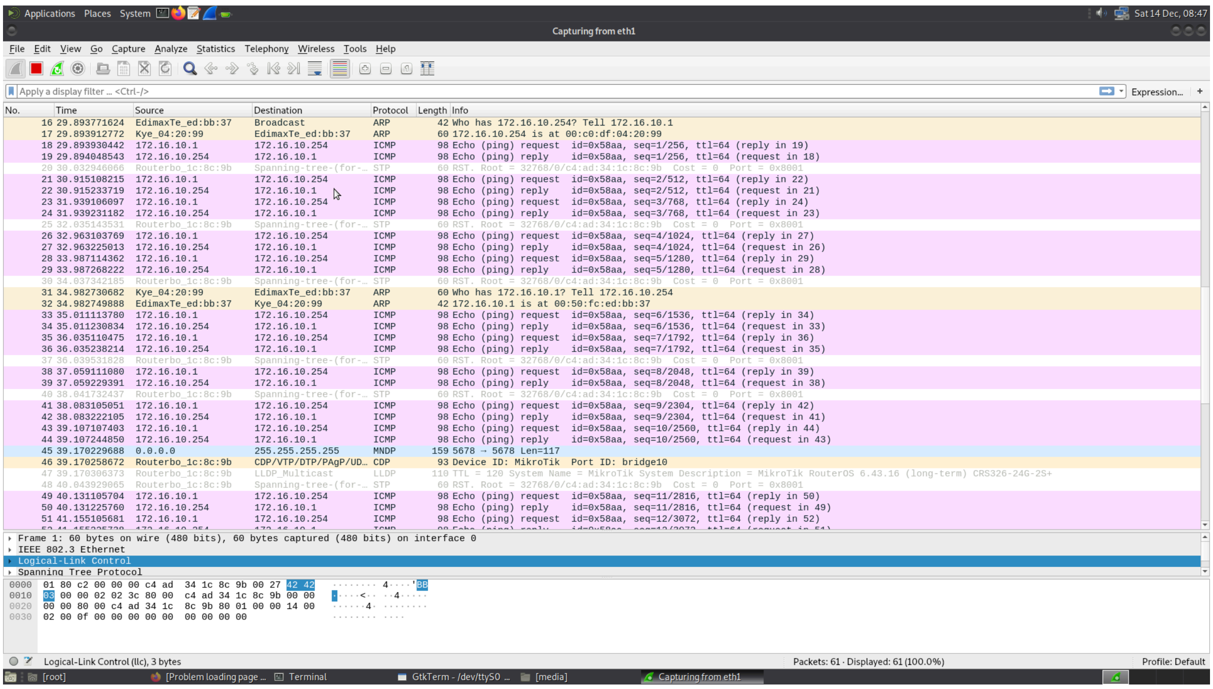
```
  snprintf(buffer, sizeof(buffer), "PASV\r\n");
  write(control_sock, buffer, strlen(buffer));
  read_ftp_response(control_sock, buffer, sizeof(buffer));
  if (strncmp(buffer, "227", 3) != 0)
  {
    fprintf(stderr, "Passive mode failed: %s", buffer);
    return -1;
  }

  // Parse the response to extract IP and port
  start = strchr(buffer, '(');
  if (!start || sscanf(start, "(%d,%d,%d,%d,%d,%d)", &ip1, &ip2, &ip3,
&ip4, &p1, &p2) != 6)
  {
    fprintf(stderr, "Failed to parse passive mode response: %s",
buffer);
    return -1;
  }

  snprintf(ip, 16, "%d.%d.%d.%d", ip1, ip2, ip3, ip4);
  *port = p1 * 256 + p2;
  return 0;
}

// Retrieves a file from the server
int ftp_retrieve_file(int control_sock, int data_sock, const char
*remote_path, const char *local_filename)
{
  char buffer[2048];
  FILE *file;

  // Set binary mode using TYPE I command
  snprintf(buffer, sizeof(buffer), "TYPE I\r\n");
  write(control_sock, buffer, strlen(buffer));
  read_ftp_response(control_sock, buffer, sizeof(buffer));
  if (strncmp(buffer, "200", 3) != 0)
  {
    fprintf(stderr, "Failed to set binary mode: %s", buffer);
    return -1;
  }

  // Send RETR command to retrieve the file
```

```c
  snprintf(buffer, sizeof(buffer), "RETR %s\r\n", remote_path);
  write(control_sock, buffer, strlen(buffer));
  read_ftp_response(control_sock, buffer, sizeof(buffer));

  // Expecting 150 (File status okay; about to open data connection)
  if (strncmp(buffer, "150", 3) != 0 && strncmp(buffer, "125", 3) != 0)
  {
    fprintf(stderr, "Failed to retrieve file: %s", buffer);
    return -1;
  }

  // Open file for writing
  if ((file = fopen(local_filename, "wb")) == NULL)
  {
    perror("fopen");
    return -1;
  }

  // Read data from the data socket and write to file
  int bytes;
  char data_buf[1024];
  while ((bytes = read(data_sock, data_buf, sizeof(data_buf))) > 0)
  {
    if (fwrite(data_buf, 1, bytes, file) != (size_t)bytes)
    {
      perror("fwrite");
      fclose(file);
      return -1;
    }
  }

  // Check if reading from data socket encountered an error
  if (bytes < 0)
  {
    perror("read");
    fclose(file);
    return -1;
  }

  fclose(file);

  // After data transfer, server should send a 226 response (Transfer
complete)
```

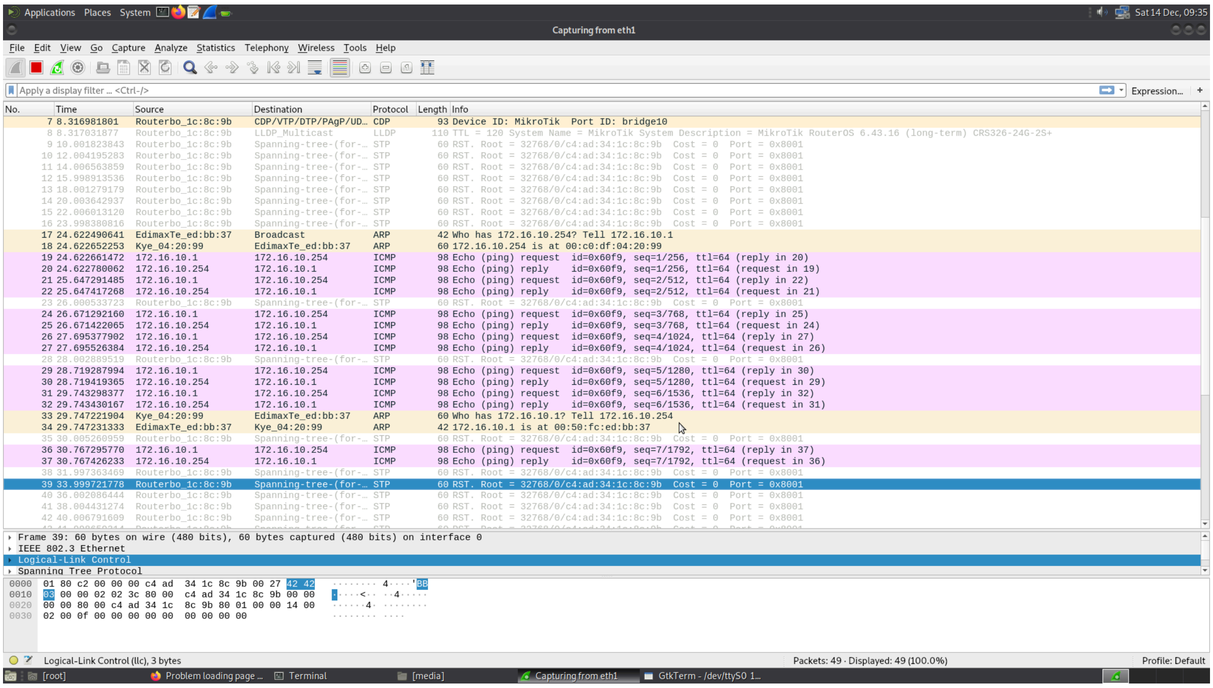
```
  read_ftp_response(control_sock, buffer, sizeof(buffer));
  if (strncmp(buffer, "226", 3) != 0)
  {
    fprintf(stderr, "File transfer incomplete: %s", buffer);
    return -1;
  }

  return 0;
}

// Sends QUIT command to close the session
void ftp_quit(int control_sock)
{
  char buffer[2048];
  snprintf(buffer, sizeof(buffer), "QUIT\r\n");
  write(control_sock, buffer, strlen(buffer));
  read_ftp_response(control_sock, buffer, sizeof(buffer));
  printf("Server response: %s", buffer);
}
```

Configuration and Study of a Network Experience Logs and Screenshots

2.1.1 - Tux13 pinging Tux14

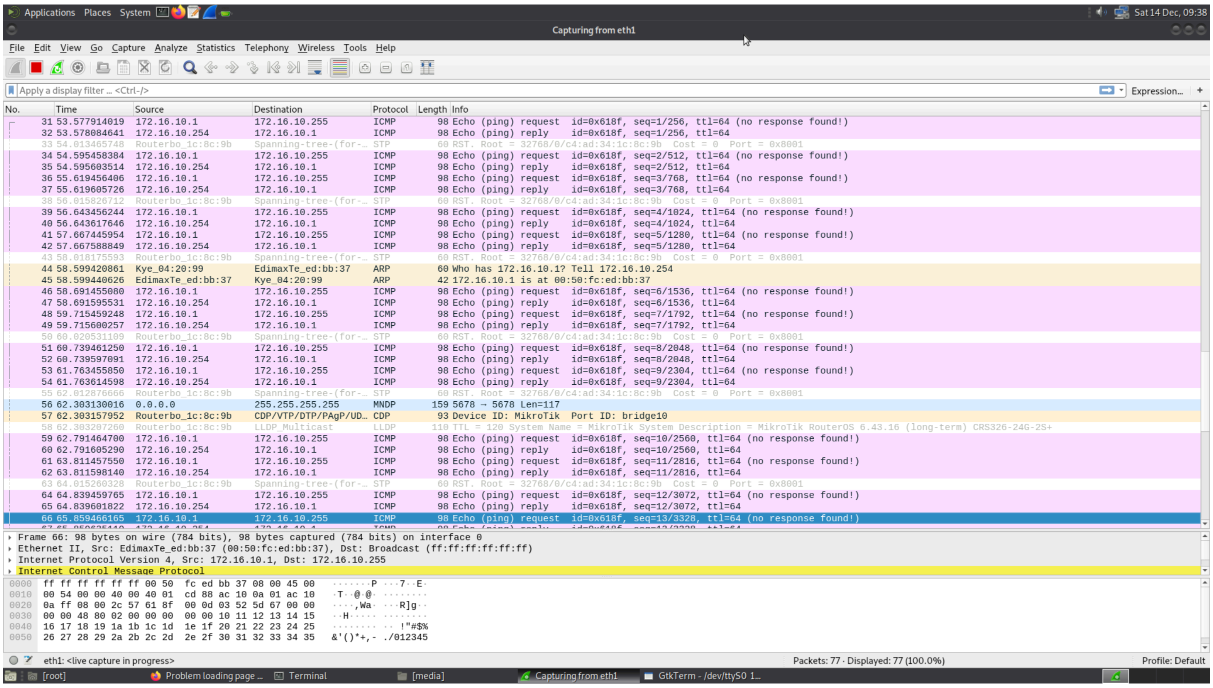


2.2.1 - Tux13 pinging Tux14

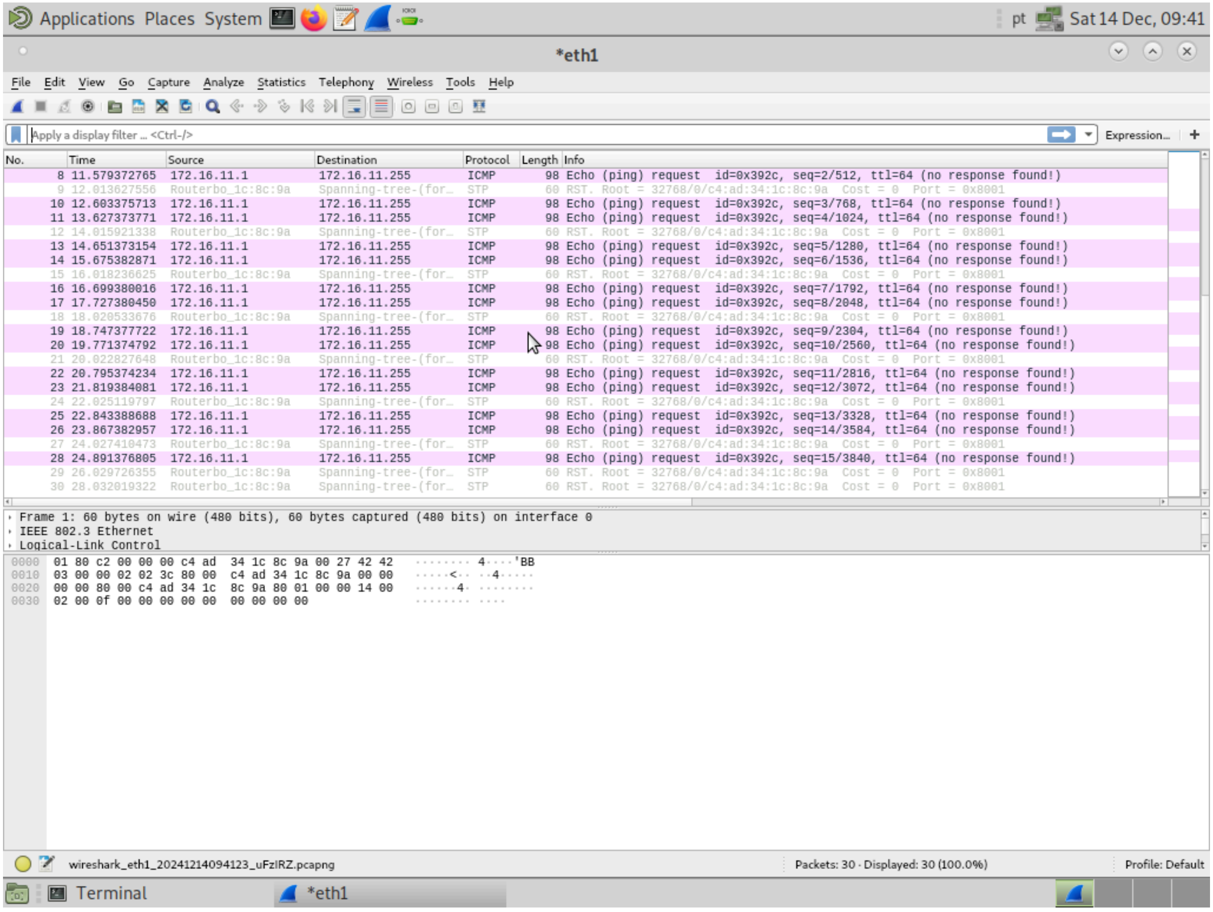


2.2.2 - Tux12 unreachable

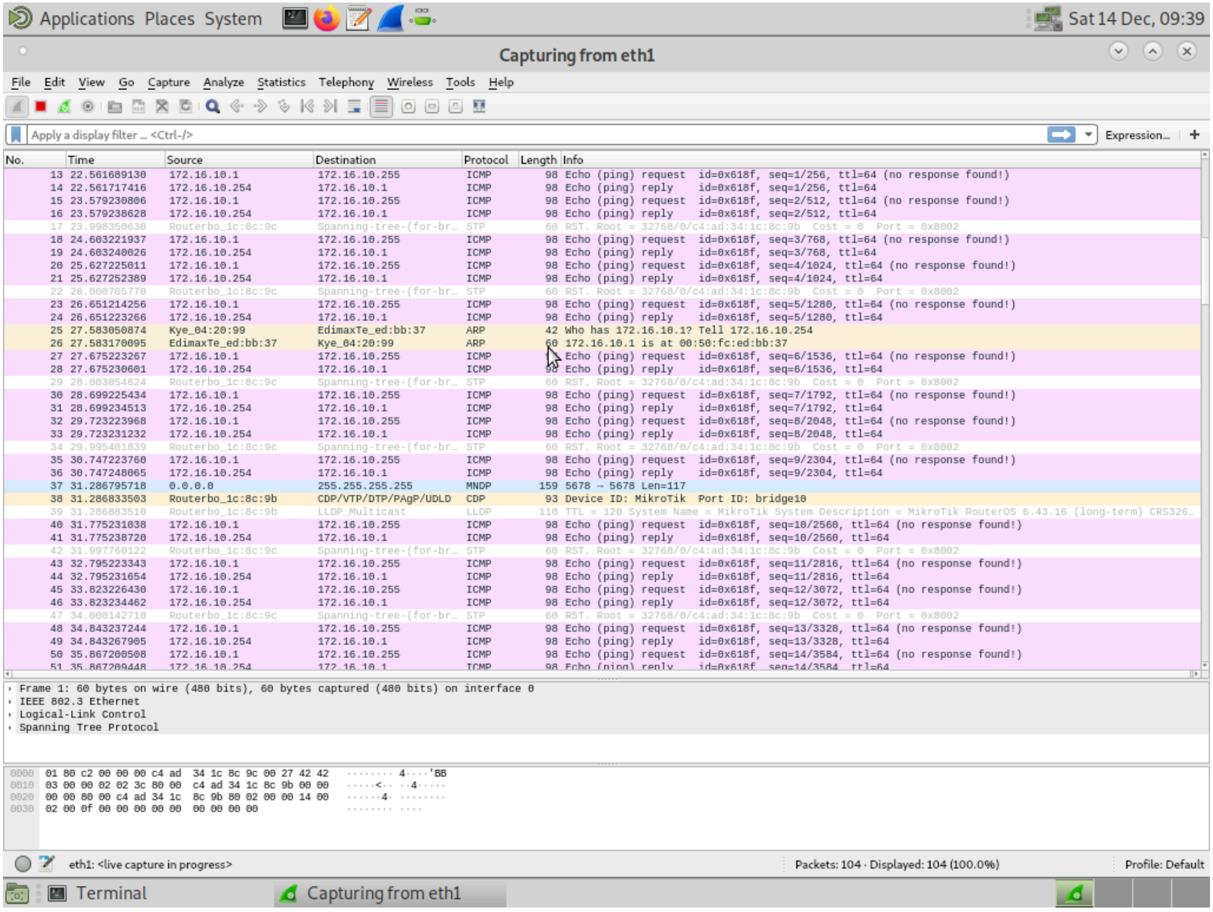
2.2.3 - Tux13 ping broadcast logs



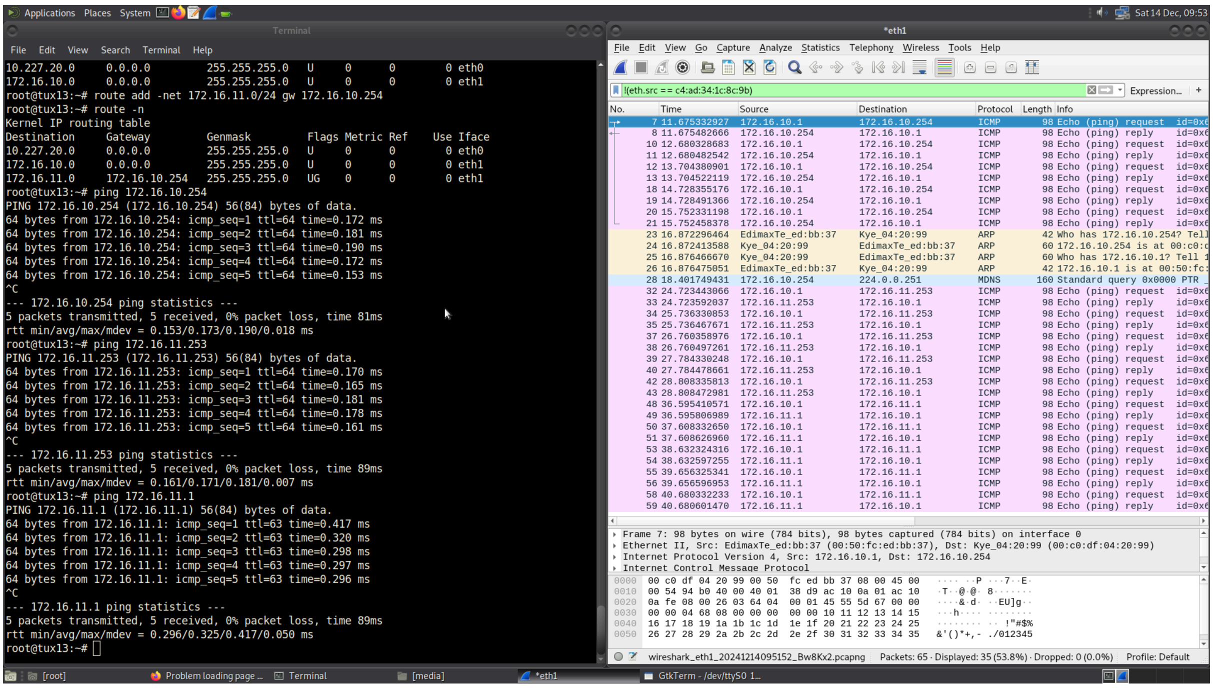
2.2.4 - Tux12 ping broadcast logs



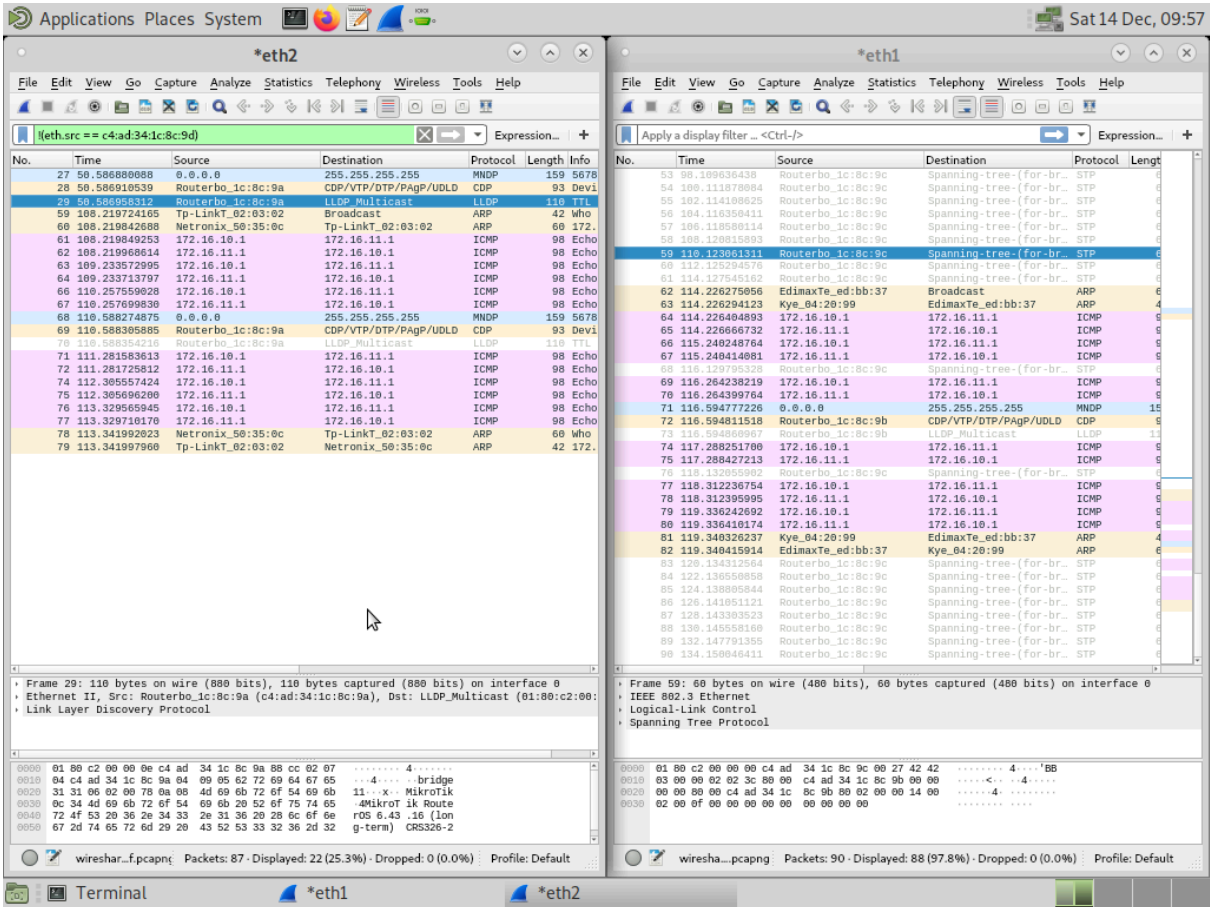
2.2.5 - Tux13 ping broadcast logs received in Tux14



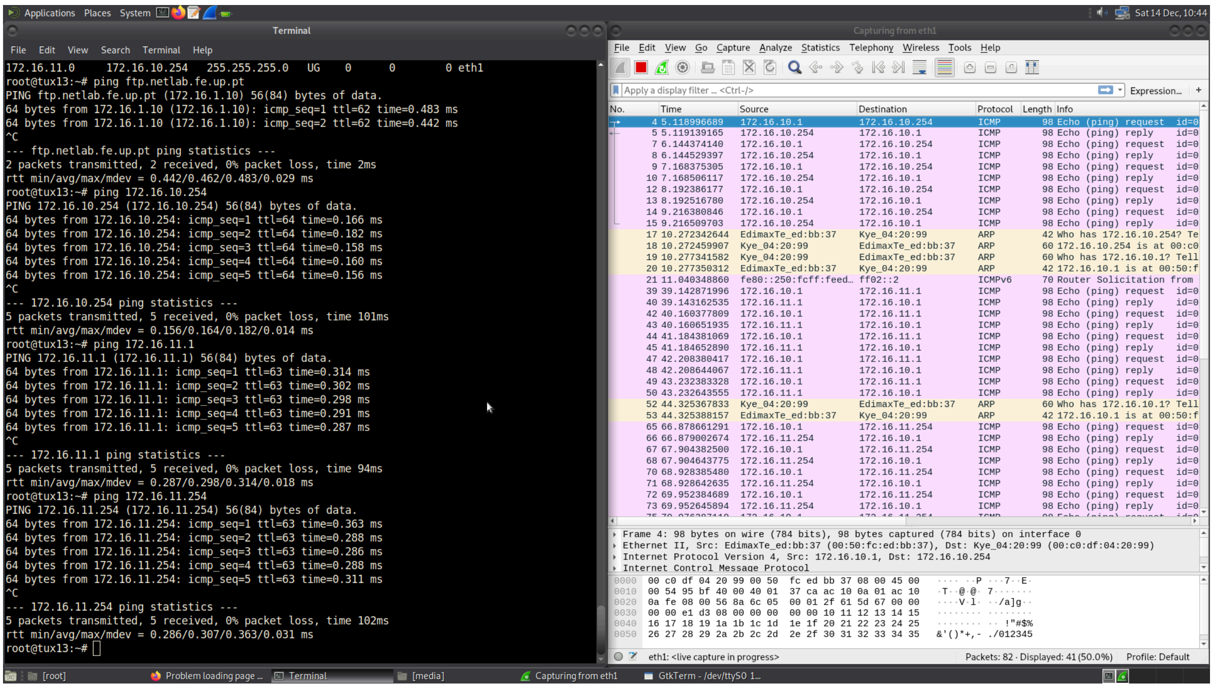
2.3.1 - Tux13 pinging other interfaces



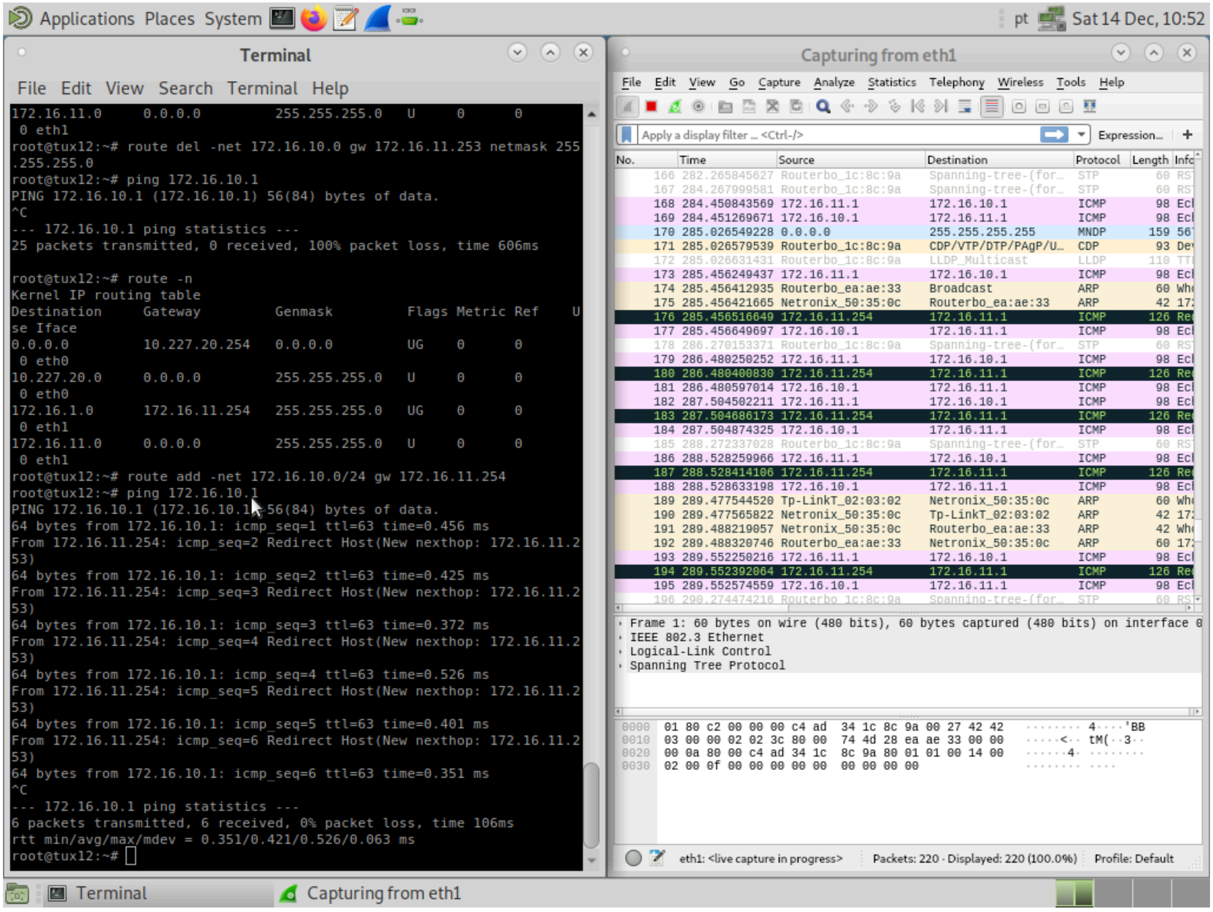
2.3.2 - Tux 14 capturing both network interfaces



2.4.1 - Tux13 pinging all network interfaces



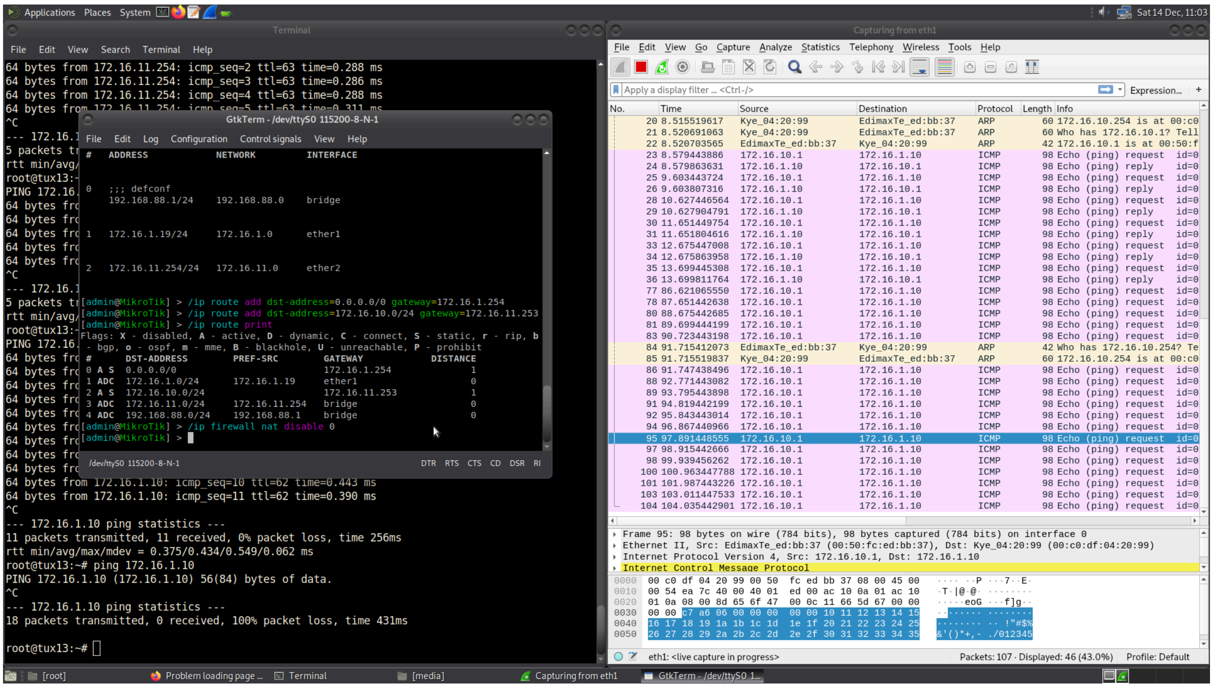
2.4.2 - Tux12 pinging Tux13 through Rc



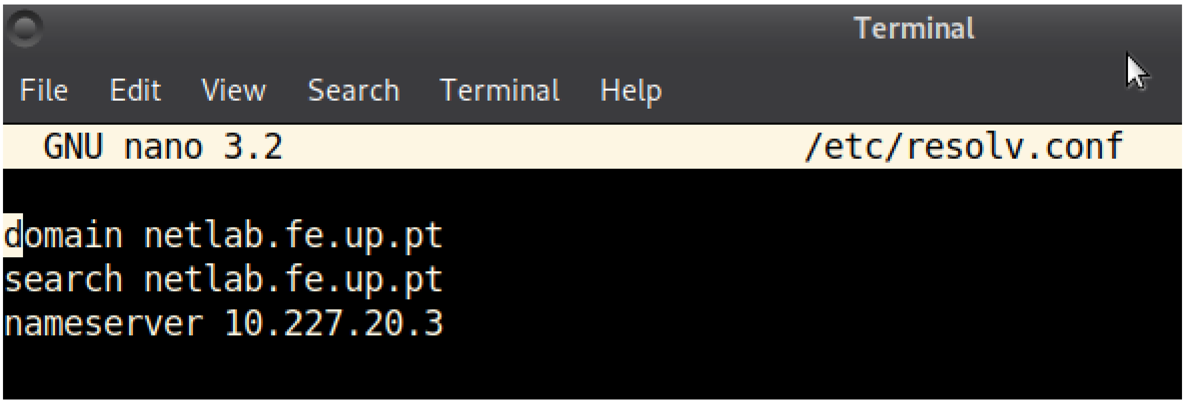
2.4.3 - Tux12 traceroutes

```
root@tux12:~# traceroute -n 172.16.10.1
traceroute to 172.16.10.1 (172.16.10.1), 30 hops max, 60 byte packets
 1  172.16.11.254  0.178 ms  0.172 ms  0.172 ms
 2  172.16.11.253  0.271 ms  0.264 ms  0.266 ms
 3  172.16.10.1  0.410 ms  0.411 ms  0.393 ms
root@tux12:~# route add -net 172.16.10.0/24 gw 172.16.11.253
root@tux12:~# traceroute -n 172.16.10.1
traceroute to 172.16.10.1 (172.16.10.1), 30 hops max, 60 byte packets
 1  172.16.11.253  0.180 ms  0.170 ms  0.153 ms
 2  172.16.10.1  0.273 ms  0.266 ms  0.271 ms
```

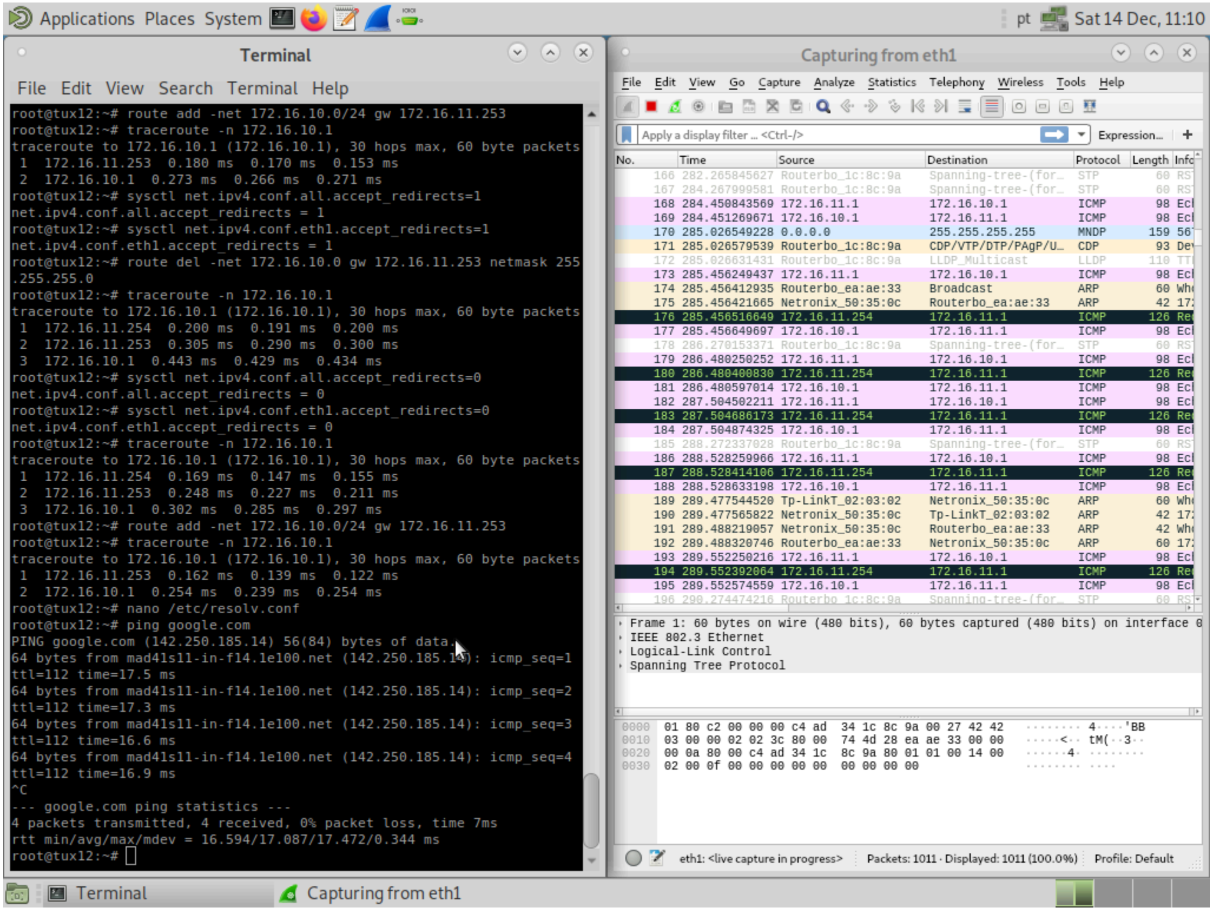
2.4.4 - Exp 4 step 7



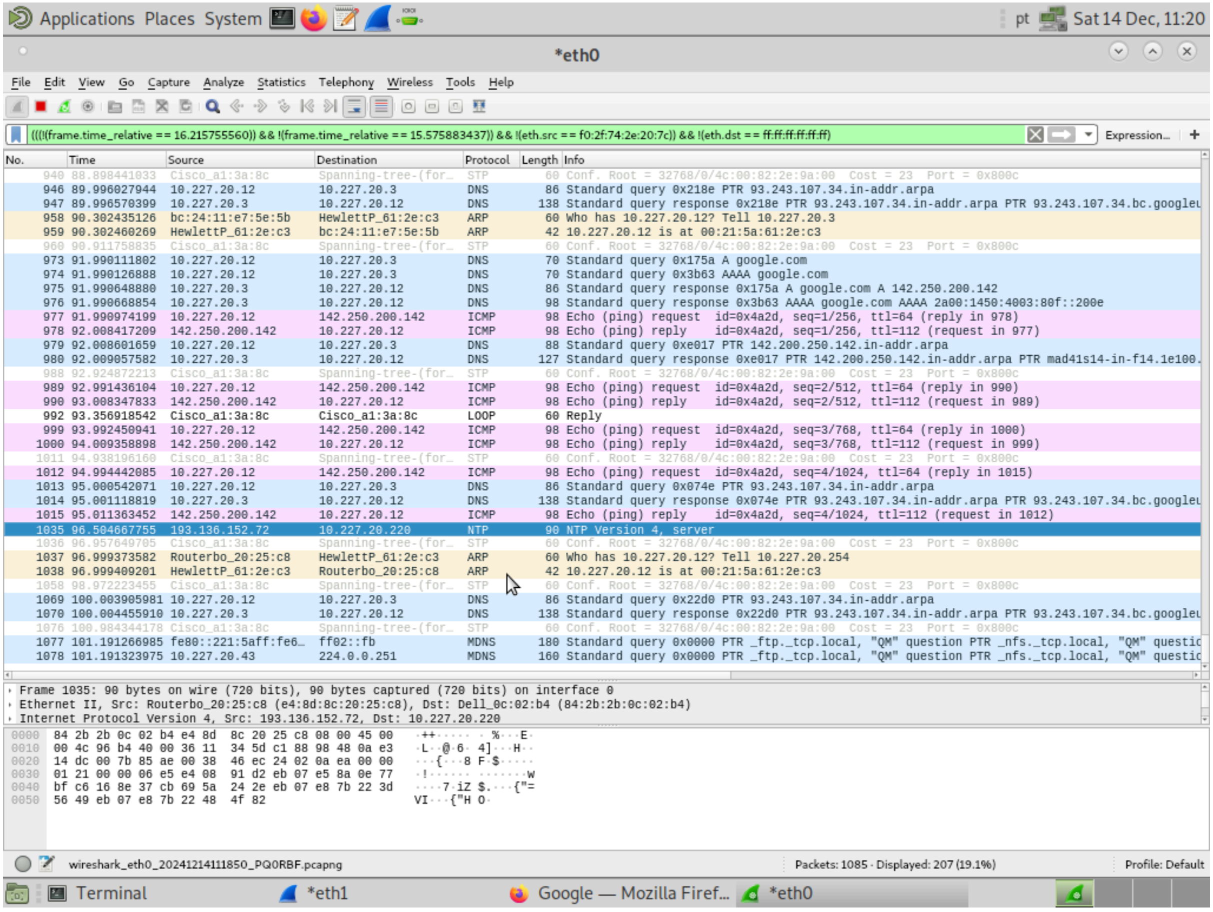
2.5.1 - DNS configuration



2.5.2 - Tux12 testing if names can be used in hosts



2.5.3 - Tux12 pinging google.com logs



2.6.1 - Exp 6 Wireshark logs