EE533 Lab 1 Report

Introduction

In this Lab, we explore the fundamentals of TCP socket programming in a growing manner to finally build a robust client-server communication system. First we setup two virtual machine (server and client). The server is furtherly enhanced to handle multiple clients concurrently using `fork()`, and the lab solves the zombie process problem by introducing `SIGCHLD` and `SigCatcher()`. Through this lab, we gain hands-on experience with socket creation, connection handling, inter=process communication and the server design fundamental in Linux/UNIX environment.

Part I: Setting Up Two Virtual Machine Nodes on VMWare

In this part, we set up two virtual machine using Ubuntu and name them EE533-Server and EE533-Client.

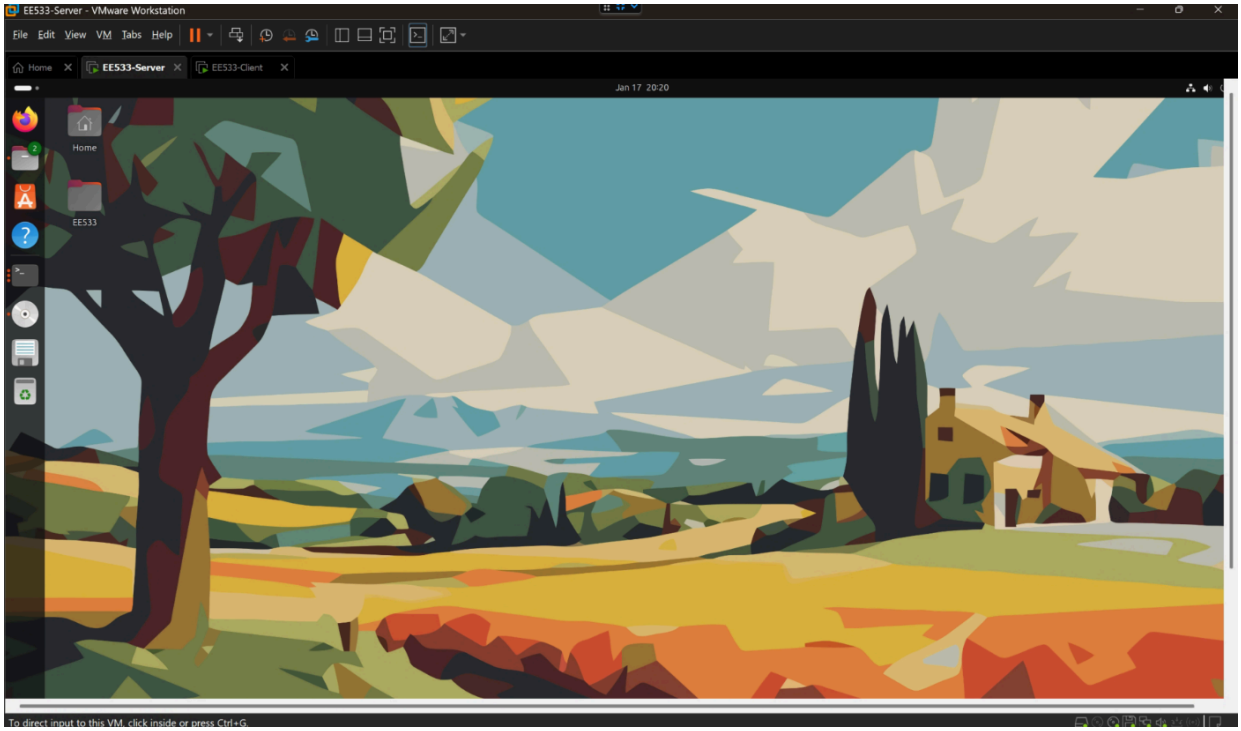

Figure 1. Creating Ubuntu VMs in VMWare Workstation

Part II: Communication Between Server and Client with Sockets

We use the existing code provided along with the lab material to test using the VMs we created in part 1. We first use `ip addr` command or `ifconfig` to get the server IP, then we ping the IP address in the client VM to check if the connection is secured.

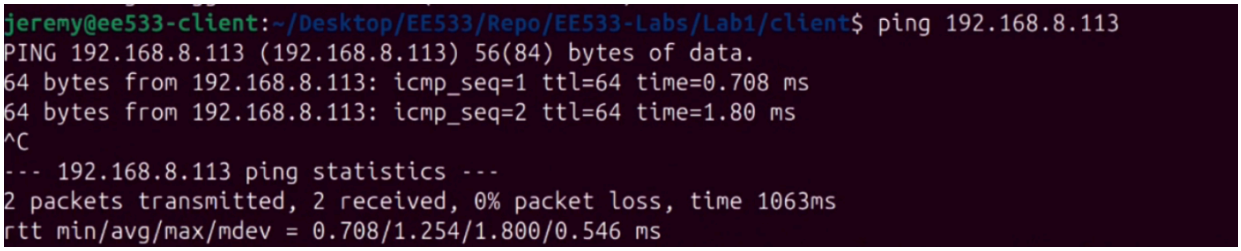

Figure 2. Ping the Server IP from Client VM

Then we use `gcc server.c -o server` and `gcc client.c -o client` to compile given code in server and client VM separately. Run the server code using `/server 51717` and client code using `./client 192.168.8.113 51717.` In client VM, we are asked to enter the message and we can get the message printed in the server VM.

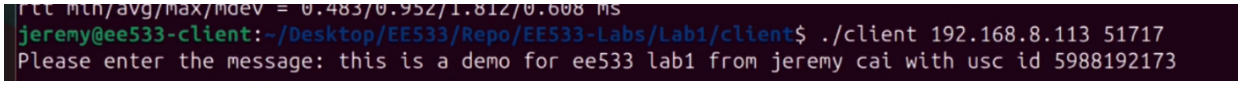

Figure 3. Client VM Asking to Enter Message

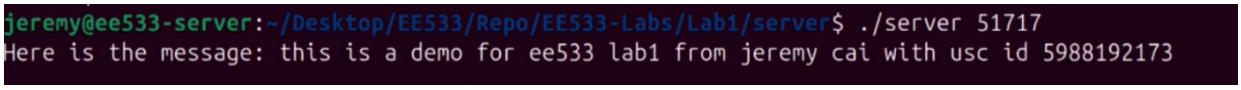

Figure 4. Server VM Getting Message

Part III: Enhancements to The Server Code

In the enhancement section, the lab modifies the original server code to allow it to handle **multiple client connections continuously**. This is done by placing the `accept()` and message-handling logic inside a `while(1)` loop so the server runs indefinitely. To support concurrent clients, the server uses fork() to create a child process for each new connection, allowing the parent to return immediately to listen for more clients. Additionally, the lab introduces a solution to the **zombie process problem**: a `SIGCHLD` signal handler (SigCatcher) is installed to clean up terminated child processes using `wait3(NULL, WNOHANG, NULL)`. This ensures the server remains stable without accumulating defunct (zombie) processes.

```
...
void dostuff(int sock)
{
    char buffer[256];
    int n;
    bzero(buffer,256);
    n = read(sock,buffer,255);
    if (n < 0) error("ERROR reading from socket");
    printf("Here is the message: %s\n",buffer);
    n = write(sock,"I got your message",18);
    if (n < 0) error("ERROR writing to socket");
}
...

     /* Enhancement to the server code with while loop */
     int pid;
```

```
     while (1)
     {
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0)
            error("ERROR on accept");
        pid = fork();
        if (pid < 0)
            error("ERROR on fork");
        if (pid == 0)  /* child process */
        {
            close(sockfd);
            dostuff(newsockfd);
            exit(0);
        }
        else  /* parent process */
        {
            close(newsockfd);
        }
     }
     close(sockfd);
...
```

Code 1. Code Change for Server Concurrent Connections

We observe that the concurrent connection is maintained when client code finish execute. However, as described in the document, this concurrent connection maintenance is creating a new issue about zombie process. The lab material suggest a complete method of solving this issue using

```
...
#include <sys/wait.h>
#include <signal.h>

...

void *SigCatcher (int n)
{
    wait3(NULL, WNOHANG, NULL);
}

...
     /* Used to catch SIGCHLD and prevent zombie processes */
     signal(SIGCHLD, SigCatcher);

...
```

Code 2. Code Change for Server Concurrent Connections without Zombie

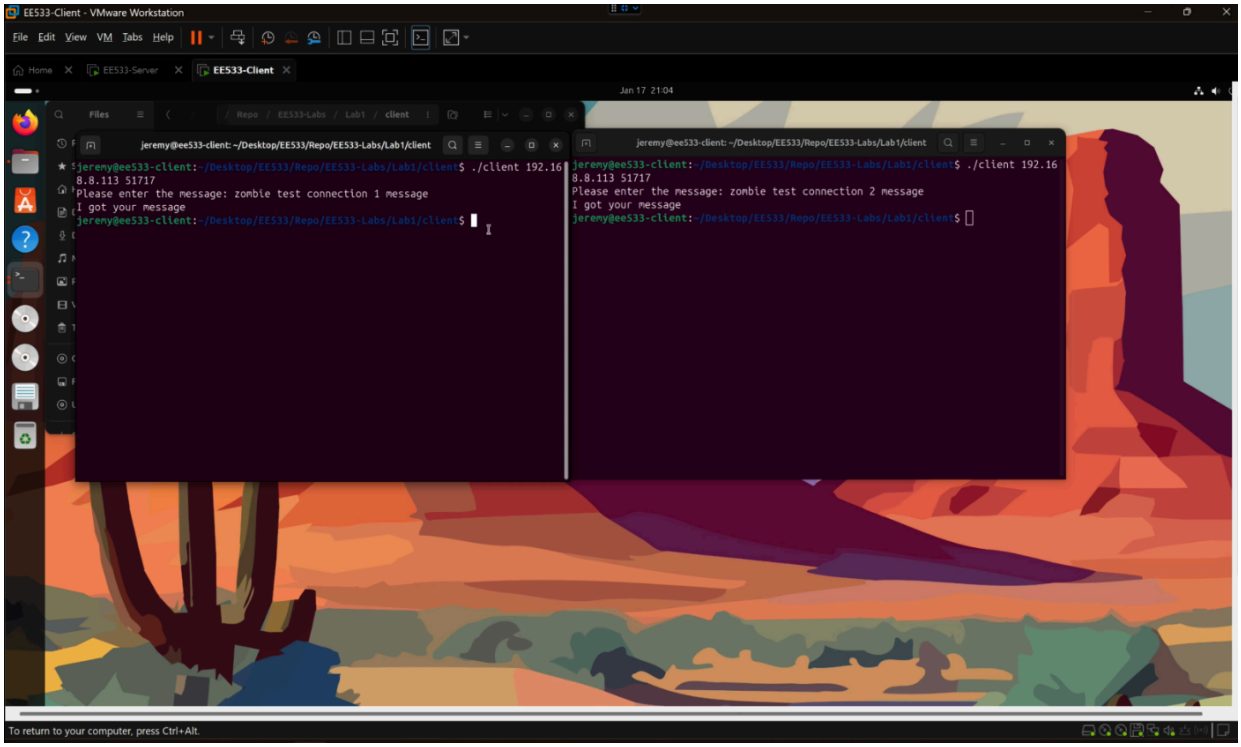

Figure 5. Client Concurrent Connection for Zombie Testing

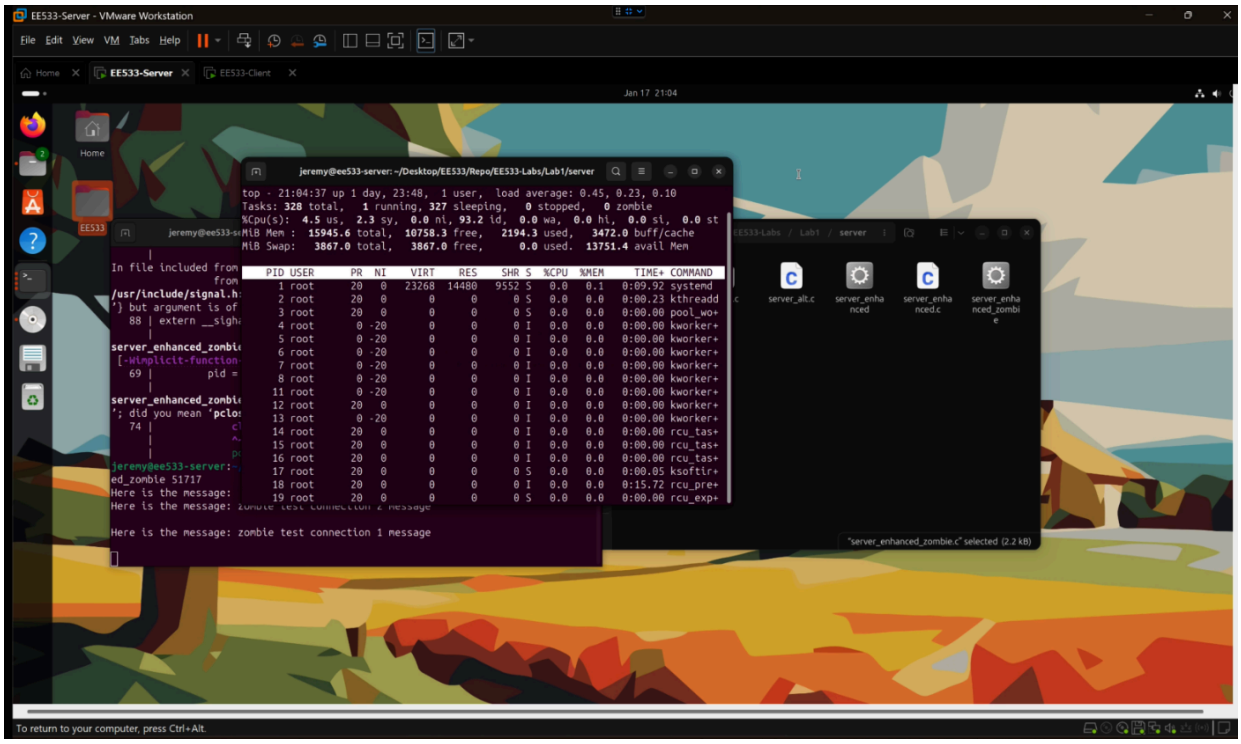

Figure 6. Server Getting Message and No Zombie Shown

Part IV: Alternative Types of Sockets (UDP)

In previous parts, we are using TCP as the main method of a tram socket in the Internet domain. In this part, we implement UDP as an alternative type to enhance the understanding of the various connection server and client code design.

```c
/* Alternative method for server code design */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
    int sock;
    int portno;

    struct sockaddr_in serv_addr;
    struct sockaddr_in from;
    socklen_t fromlen;

    int n;
    char buf[1024];

    if (argc < 2) {
        fprintf(stderr, "ERROR, no port provided\n");
        exit(1);
    }

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
        error("opening socket");

    bzero((char *)&serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    if (bind(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
        error("binding");

    fromlen = sizeof(struct sockaddr_in);

    while (1) {
```

```
        bzero(buf, 1024);

        n = recvfrom(sock, buf, 1024, 0, (struct sockaddr *)&from, &fromlen);
        if (n < 0)
            error("recvfrom");

        printf("Here is the message: %s\n", buf);

        n = sendto(sock, "Got your message", 17, 0, (struct sockaddr *)&from, fromlen)
        if (n < 0)
            error("sendto");
    }

    return 0;
}
```

Code 3. Alternative Server Design Using UDP

```
/* Alternative method for client code design */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

static void error(const char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
    int sock, portno, n;
    socklen_t serverlen, fromlen;

    struct sockaddr_in serveraddr, from;
    struct hostent *server;

    char buf[1024];

    if (argc < 3) {
        fprintf(stderr, "usage %s hostname port\n", argv[0]);
        exit(0);
    }

    portno = atoi(argv[2]);

    sock = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    if (sock < 0) error("opening socket");

    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(0);
    }

    bzero((char *)&serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *)&serveraddr.sin_addr.s_addr, server->h_lengt
    serveraddr.sin_port = htons(portno);

    serverlen = sizeof(serveraddr);

    printf("Please enter the message: ");
    bzero(buf, 1024);
    fgets(buf, 1023, stdin);

    n = sendto(sock, buf, strlen(buf), 0, (struct sockaddr *)&serveraddr, serverlen);
    if (n < 0) error("sendto");

    fromlen = sizeof(from);
    bzero(buf, 1024);
    n = recvfrom(sock, buf, 1024, 0, (struct sockaddr *)&from, &fromlen);
    if (n < 0) error("recvfrom");

    printf("Server reply: %s\n", buf);

    close(sock);
    return 0;
}
```

Code 4. Alternative Client Design Using UDP

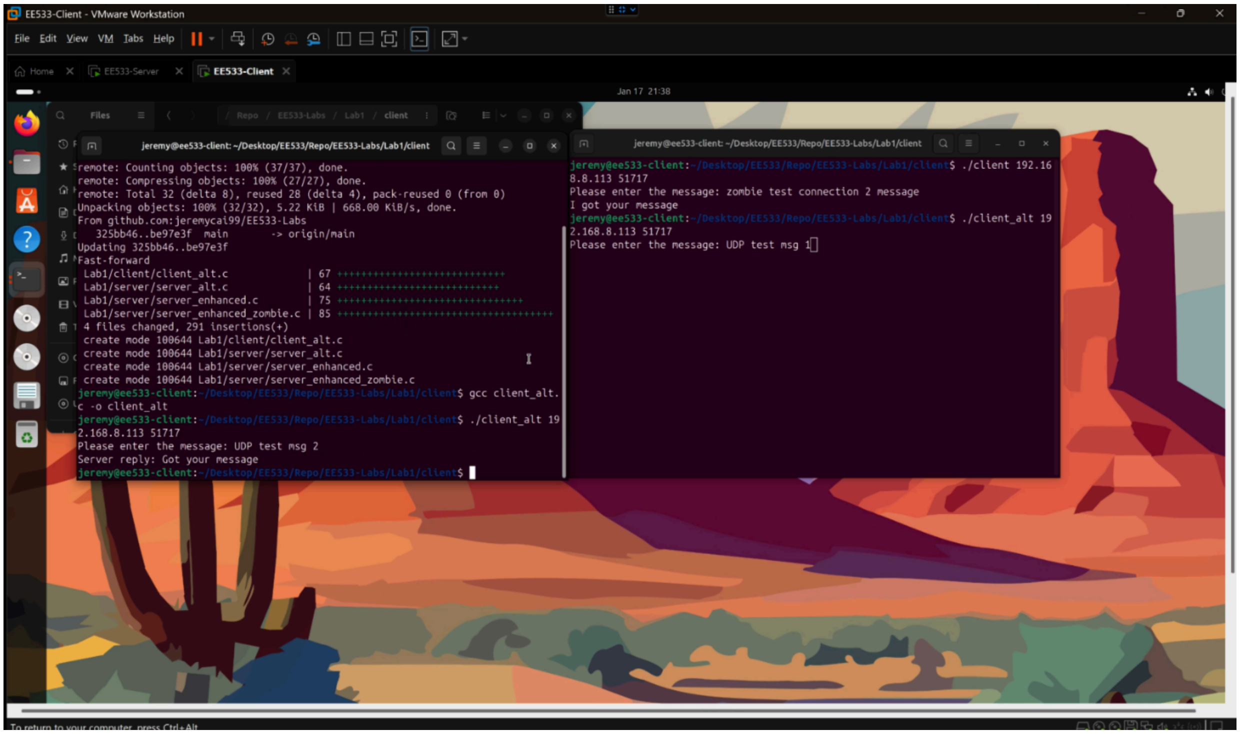

Figure 7. Alternative Method Client Setup

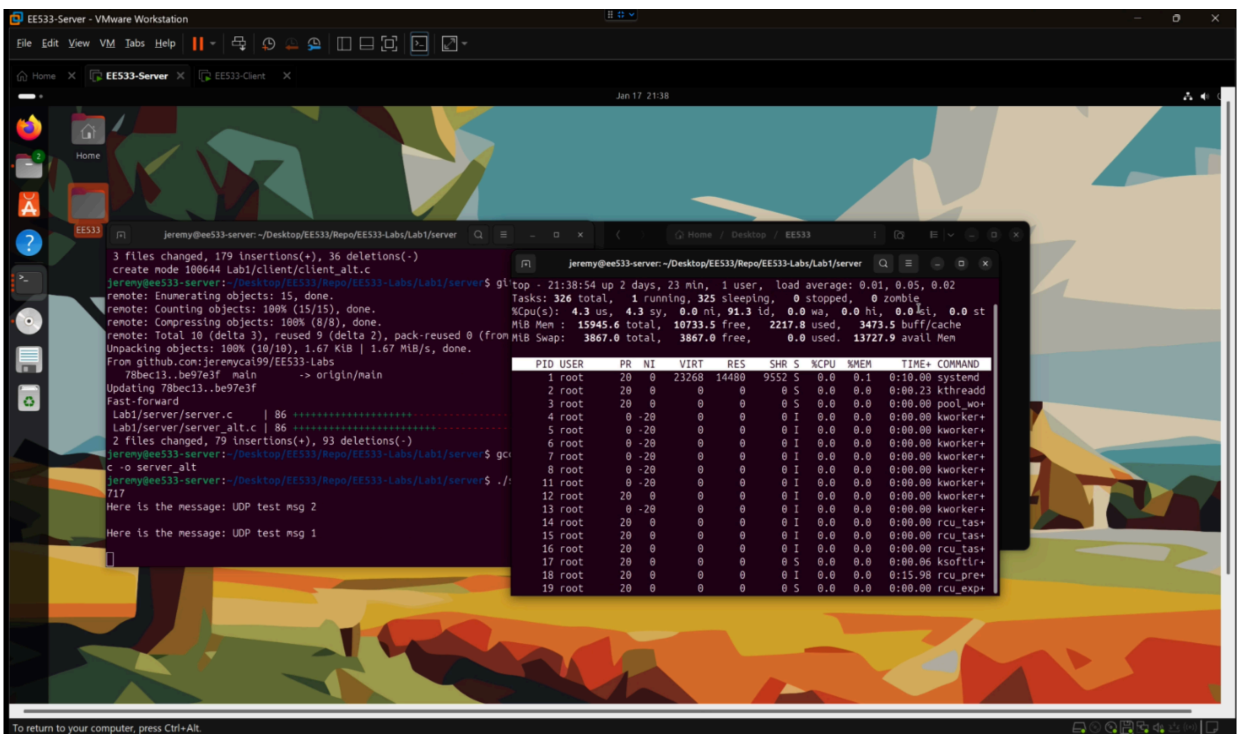

Figure 7. Alternative Method Server Setup and Connection Check

Part V: Sockets in the Unix Domain

In this part, we are using the existing code for testing the connection in UNIX domain. Since this test should be done within a single VM, we focus on server VM for this test. First, we need to run the server code with an argument of the path to the socket and observe the file created at that path

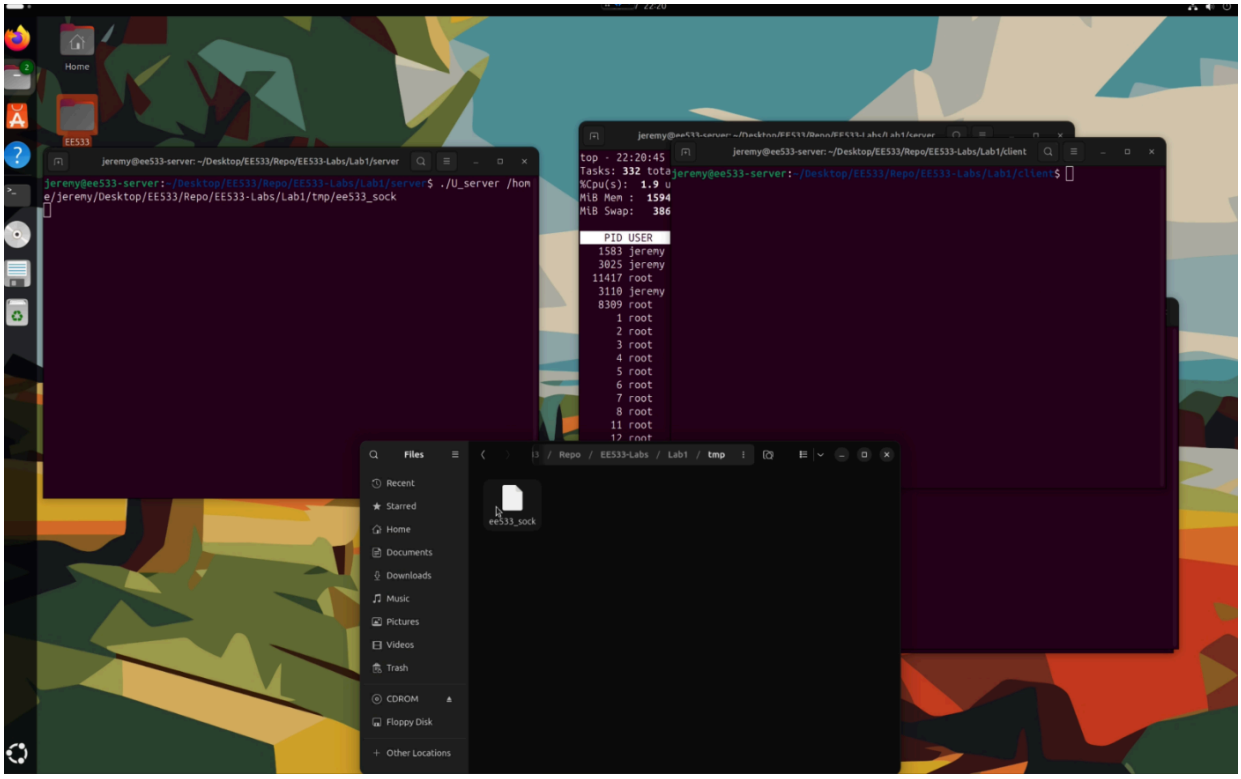

Figure 8. Running Server Code to Create Socket First

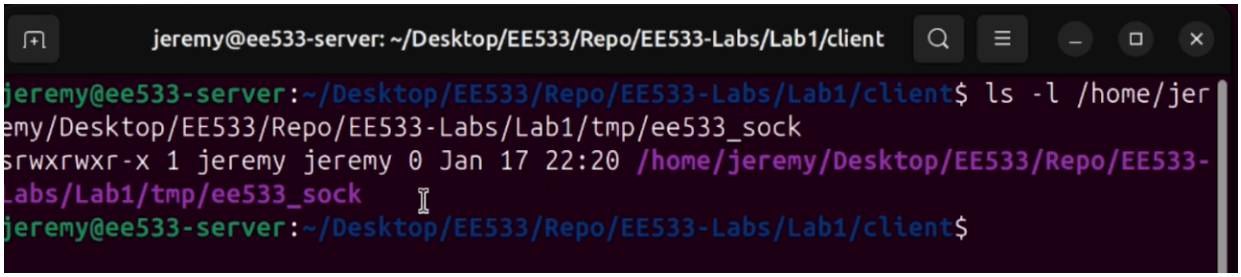

Figure 9. Verify the Socket Existence

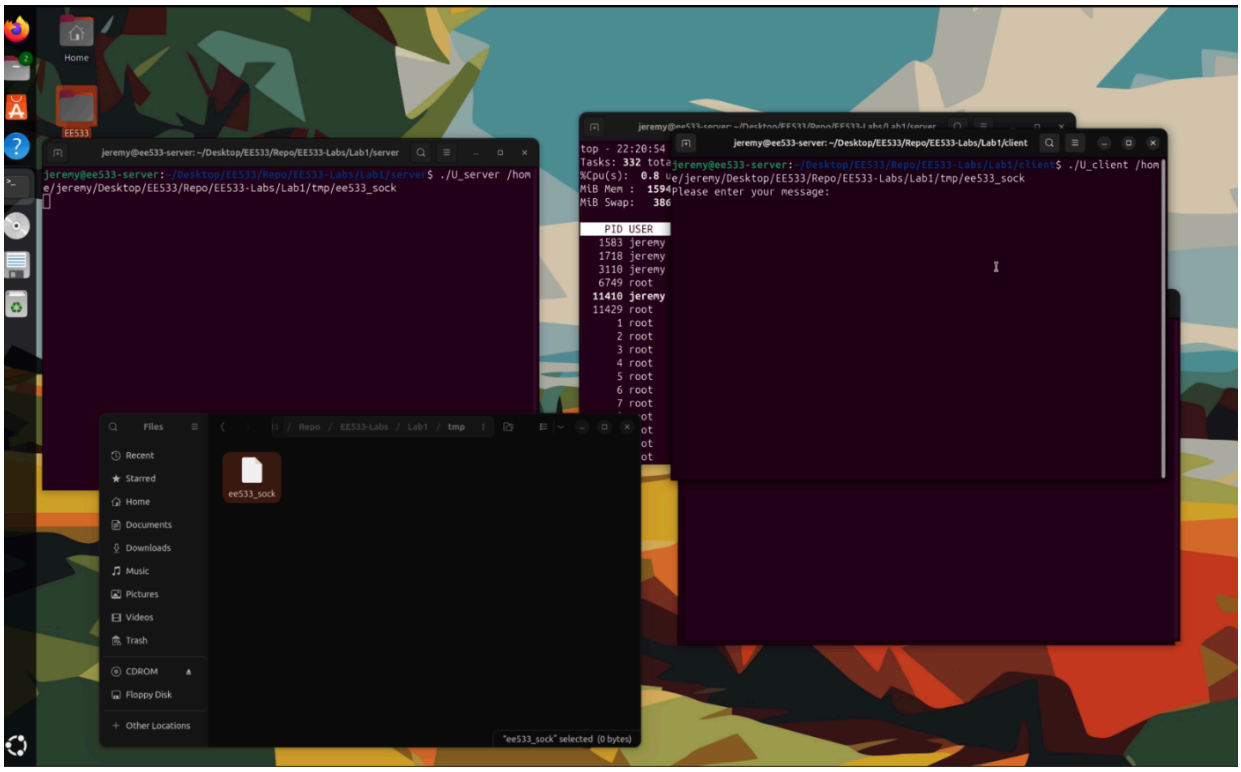

Figure 10. Run the Client Code to Connect to the Socket Created by Server

Conclusion

In this lab, we successfully built and tested socket-based communication programs in C to understand core client–server networking concepts. We first implemented a basic TCP client and server, then enhanced the server to run continuously and handle multiple clients by using a loop and fork(), and addressed the zombie process issue by properly reaping terminated children as described in the lab material. We also implemented and validated the UDP “datagram” alternative using recvfrom()/sendto() (without listen()/accept()), and explored Unix-domain sockets where processes communicate through a filesystem pathname on the same VM, confirming socket-file creation and cleanup behavior. Overall, this lab strengthened our understanding of stream vs. datagram semantics, process-based server design, and practical debugging/testing workflows in a Linux virtual environment.

Project Links

Github Link: https://github.com/jeremycai99/EE533-Labs.git
YouTube Link: https://youtu.be/y-0NcgyoBKY