

CS 558: Computer Systems Laboratory

(January–May 2026)

Assignment 1: Network Diagnostics & Socket Programming

Submission Deadline: 11:59 PM, Thursday, 29 January 2026

General Guidelines

- All groups **must complete Questions Q1 to Q7**. A soft copy of the report (preferably in PDF format) covering these experiments must be submitted.
- In addition to Q1–Q7, **each group must implement exactly one application** assigned to them as per the Task Allocation Table provided at the end.
- A **single consolidated submission** should include solutions to Q1–Q7 along with the assigned application.
- All applications **must be developed using socket programming in C or C++ only**. Submissions using any other language will not be accepted.
- Submit the complete set of source files as a **compressed archive (ZIP/RAR/TAR.GZ)**. The file size must not exceed **1 MB**, and the archive name should match your group number (e.g., Group_4.zip).
- **Only one group member** needs to upload the submission using the provided Google Form: <https://forms.gle/fE2pjP2dwkebi3uR6>
- Evaluation will be conducted **offline or through viva voce during lab sessions**, where groups must demonstrate execution and explain their code.
- All code must be **originally written** by the group. Plagiarism checks will be performed, and any unfair practices will result in **negative marking equal to the maximum assignment marks**.
- Marks will be awarded **collectively to the entire group**.

Reference:

<https://www.geeksforgeeks.org/socket-programming-cc/>

Q1. Ping Command Exploration (Linux/Unix Only)

The ping utility is used to verify network connectivity by transmitting ICMP Echo Request packets and measuring the time taken for Echo Replies to return. It operates similarly to sonar by analysing round-trip delays.

Investigate the ping command and answer the following:

1. How can ping be used to **detect packet loss**? Identify the part of the output that provides this information.
 2. What does the **TTL (Time To Live)** field represent in a ping packet, and which option allows you to modify its value?
 3. How can you limit the **total duration** for which the ping command runs, **regardless** of the number of packets sent?
 4. Which option in the ping command allows you to set a **timeout value for waiting for an Echo Reply**? Explain how this affects the output.
-

Q2. RTT and Packet Loss Analysis

Choose **six different Internet hosts** of your choice and list them in your report. Perform the following experiments:

- Ping each host **25 times** during **three distinct time slots** in a day.
- Identify cases where **packet loss exceeds 0%** and explain possible reasons.
- Compute the **average Round Trip Time (RTT)** for each host.
- Analyze whether RTT values show **strong or weak dependence on geographical distance**.

Next, select **one host** from the above set and repeat the experiment using **packet sizes ranging from 64 bytes to 2048 bytes**. Plot the **average RTT versus packet size** and discuss:

- The effect of packet size on RTT
- The influence of time-of-day on network latency

You may use any of the following online tools:

- <http://www.spfld.com/ping.html>
 - <https://www.subnetonline.com/pages/network-tools/online-ping-ipv4.php>
-

Q3. ifconfig and route Commands

Answer the following based on practical execution:

1. Execute the ifconfig command and explain each significant field in its output.
 2. Using ifconfig, how can you assign a **static IP address and subnet mask** to an **interface**? What precautions should be taken while doing this?
 3. How can ifconfig be used to enable or disable a network interface? Demonstrate this with a command and explain the effect.
 4. How can you use the route command to route traffic through a specific network interface?
-

Q4. Network Statistics Using `netstat`

1. What is the primary purpose of the `netstat` command?
 2. How can `netstat` be used to display listening TCP ports only? Explain the significance of these ports.
 3. What is the difference between **`netstat -a`** and **`netstat -l`**? Execute both commands and compare their outputs.
 4. How can `netstat` be used to **monitor** network statistics **continuously**? Demonstrate this with an example.
 5. Identify the option that reports **UDP connection statistics**, execute it, and explain the output.
 6. How can **`netstat`** help in **troubleshooting** network performance issues or unauthorized connections?
-

Q5. Traceroute Study

The traceroute utility is used to identify the path packets take across the network.

Using the **same hosts selected in Q2**, perform traceroute experiments at **three different times of the day** using any one of the following tools:

- ping.eu
- [Cogent Looking Glass](http://CogentLookingGlass.com)
- network-tools.com

Answer the following:

1. How does traceroute determine the **sequence** of routers between the source and destination? Explain the role of the **TTL field**.
 2. What is the significance of **round-trip time** values displayed for each hop in traceroute output?
 3. How do traceroute paths differ when using **ICMP-based traceroute versus UDP- or TCP-based traceroute**?
 4. Explain how **load balancing** across multiple paths can affect traceroute output.
-

Q6. Address Resolution Protocol (ARP)

1. What is the purpose of ARP in the TCP/IP protocol stack, and at which layer does it operate?
2. What happens when an **ARP request** is sent for an IP address that **does not exist** on the local network?
3. How can **duplicate IP address** detection be performed using ARP?
4. What is ARP spoofing? How can **static ARP entries** be used to prevent **ARP spoofing**? What are their limitations?

Q7. Local Area Network Scanning Using Nmap

Use nmap to scan your local network and identify active hosts. For example:

```
nmap -n -sP <Subnet Range>
```

- You may choose any subnet, but clearly specify it in your report.
 - Perform the scan **at least six times** at different hours of the day.
 - Record the number of active hosts each time.
 - Plot a **time vs number of active hosts graph** and analyze daily usage trends in your LAN.
- 1) What types of probes does nmap use to determine whether a host is active?
 - 2) What is the purpose of the -n option in the nmap command, and how does it affect scan performance?

Application Assignments

Application 1: Simple FTP Using TCP Sockets

In this assignment, you are required to implement a simple File Transfer Protocol (FTP) using TCP socket programming in C. Two programs must be developed: a server and a client. The server starts first and waits for incoming TCP connections, while the client connects to the server using the server's IP address and port number provided through the command line.

After a successful connection, the client should support the following file transfer operations:

- **PUT**: The client uploads a user-specified file to the server, which stores it on disk. If the file already exists on the server, the server informs the client. The client then asks the user whether to overwrite the file, and the server performs the corresponding action based on the user's choice.
- **GET**: The client downloads a specified file from the server and saves it locally. If the file already exists on the client side, the client prompts the user for overwrite permission and proceeds accordingly.
- **MPUT and MGET**: These commands extend PUT and GET to multiple files. MPUT uploads all files with a specified extension (such as .c or .txt) from the client to the server, while MGET downloads all files with the given extension from the server to the client. Both client and server must maintain a list of files available on their respective disks, and file overwrite handling must be implemented for each file transfer.

For simplicity, assume that only .c and .txt files are transferred. Appropriate message types should be designed to distinguish between commands, control messages, and file data.

Execution

The programs must not use hard-coded port numbers.

Client: <executable> <Server IP Address> <Server Port Number> ,

Server: <executable> <Server Port Number> .

You may make reasonable and valid assumptions wherever required. The implementation should be clean, modular, and well documented to clearly explain the working of the protocol.

Application 2: Error Detection Using CRC-8

Implement a **Stop-and-Wait data link layer protocol** between two nodes using **socket programming**:

- **Node A:** Client (Sender)
- **Node B:** Server (Receiver)

The protocol must support **error detection using CRC-8** and handle retransmissions using ACK/NACK and timers.

Error Detection Technique

- Use **Cyclic Redundancy Check (CRC-8)** with generator polynomial:
$$G(x) = x^8 + x^2 + x + 1$$
- The client constructs the transmitted frame $T(x)$ from the raw message by appending CRC bits.
- At the receiver:
 - if $T(x)$ is divisible by $G(x) \rightarrow$ **No error** \rightarrow **Send ACK**
 - Otherwise \rightarrow **Error detected** \rightarrow **Send NACK**

Error Simulation

- Implement **error generation** based on a user-provided **Bit Error Rate (BER)** or probability (random value between 0 and 1).
- Errors must be randomly injected into:
 - Data frames $T(x)$
 - Control frames (ACK / NACK)

Retransmission Logic

- If the sender receives a **NACK**, it retransmits T(x).
- If the sender does not receive ACK/NACK within a timeout period:
 - A **timer** triggers retransmission (to handle lost or corrupted ACK/NACK).
- Continue retransmissions until a valid ACK is received.

Server Requirements

- Implement a **concurrent server** capable of handling **multiple clients simultaneously**.
- The server should continue running as clients connect and disconnect.

Command-Line Interface

Do **not** use hard-coded port numbers.

Client: <executable> <Server IP Address> <Server Port Number>

Server: <executable> <Server Port Number>

Graceful Termination

- Client and server must **close sockets properly**.
- On pressing **Ctrl+C**, the server should:
 - Handle the signal using a **signal handler**
 - Gracefully release all open sockets before exiting

Notes

- Make **reasonable assumptions** where required.
- Code should be **clean, modular, and well-documented**.

Application 3: Base64 Encoding Communication System

This application requires the development of **two C programs—a client and a server—that communicate using TCP sockets**. The objective is to design a **simple communication protocol based on Base64 encoding**.

At the start, the server runs in a listening mode, waiting for incoming TCP connection requests. The client then initiates a connection to the server using a TCP port number supplied at runtime. Once the connection is established, the client accepts textual input from

the user and converts the input into its **Base64-encoded representation**. The encoded data is then transmitted to the server as a **Type 1 message** over the TCP connection.

Upon receiving the message, the server decodes the Base64 content to recover the original text, displays both the encoded and decoded messages, and sends an **acknowledgment response (Type 2 message)** back to the client. The client and server continue exchanging messages in this manner, allowing multiple messages to be sent during a single session.

When the client decides to terminate the communication, it sends a **Type 3 termination message** to the server. Following this, both the client and server must **close the TCP connection gracefully**, ensuring that all socket resources are properly released.

All communication between the client and server follows a predefined message structure consisting of the following fields:

1. **Message_Type** – an integer indicating the type of message
2. **Message** – a character array of fixed length (MSG_LEN)
3. The content of the message field in a **Type 3 message** can be anything.

In addition to the basic functionality, the server must be implemented as a **concurrent server**, capable of handling multiple client connections simultaneously.

The server's IP address and port number must be provided **through command-line arguments**, and the use of hard-coded network parameters is strictly prohibited.

Command-line execution format:

- **Client:** <executable> <Server_IP_Address> <Server_Port_Number>
- **Server:** <executable> <Server_Port_Number>

Any assumptions made during development should be **reasonable, clearly stated, and justified**.

Overview of Base64 Encoding

Base64 encoding is commonly used to transmit binary data over text-based communication channels. In this encoding scheme, data is processed in **24-bit blocks**, which are divided into **four 6-bit segments**. Each 6-bit value is mapped to a corresponding ASCII character.

- Values **0–25** are represented by uppercase letters **A–Z**
- Values **26–51** map to lowercase letters **a–z**
- Values **52–61** correspond to digits **0–9**
- Values **62** and **63** are encoded using the characters **‘+’** and **‘/’**, respectively

If the final data block contains fewer than 24 bits, padding is applied using **‘=’** or **‘==’** characters to complete the encoded output.

Application 4: File Transfer Protocol (FTP)

In this application, you are required to develop **two separate C programs—a client and a server—that communicate using TCP socket connections**. The objective is to design and implement a **basic File Transfer Protocol (FTP)** system.

Initially, the server remains in a listening state, waiting for incoming TCP connection requests. The client then establishes a connection with the server using a TCP port number that is provided at runtime. Once the connection is successfully established, the client can perform the following operations:

- **PUT Operation:** The client uploads a user-specified file to the server. Upon receiving the file, the server stores it on its local disk. If a file with the same name already exists on the server, the server notifies the client. The client then prompts the user to decide whether the existing file should be overwritten, and the server acts accordingly based on the user's choice.
- **GET Operation:** The client requests a file from the server and saves the received file to its local storage. If a file with the same name is already present on the client's system, the user is asked whether to overwrite it, and the operation proceeds based on the response.
- **MPUT and MGET Operations:** These commands extend the functionality of PUT and GET by allowing the transfer of **multiple files sharing a common extension** (such as .c or .txt). To support these operations, both the client and the server must maintain a list of available files on their respective disks. File overwrite handling must also be implemented for these bulk transfer commands.

All the above operations should be realized using **clearly defined message types** exchanged between the client and server. For simplicity, restrict file transfers to **text (.txt) and C source (.c) files only**.

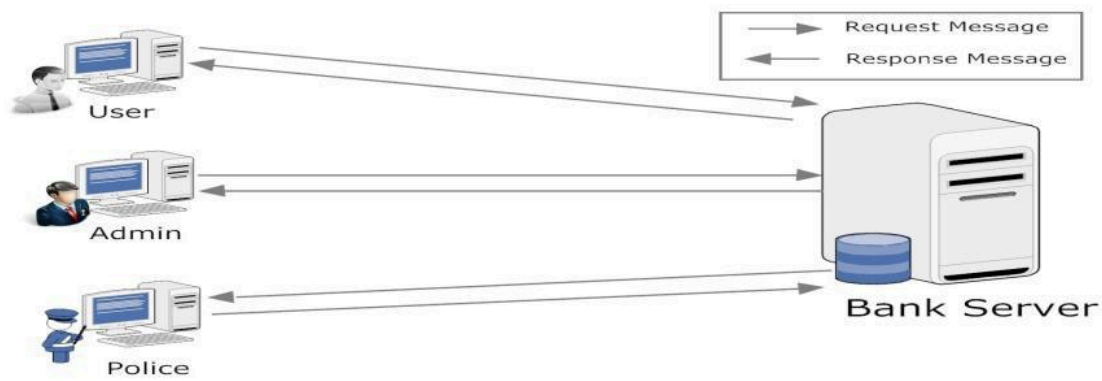
The server IP address and port number must be supplied **through command-line arguments**, and hard-coded values must be avoided. Any assumptions made during implementation should be **reasonable and explicitly stated**.

Command-line execution format:

- **Client:** <executable> <Server_IP_Address> <Server_Port_Number>
- **Server:** <executable> <Server_Port_Number>

Application 5: Client–Server Banking System (TCP)

Design a banking system using TCP sockets with separate **client and bank server** programs. Implement authentication and role-based access for **Customer, Admin, and Police** users, handling account files and transaction history securely.



This application involves developing **two C programs**—a **Client** and a **Bank Server**—that **interact using TCP socket communication**. The objective is to design a **basic client-server banking system**.

At startup, the client establishes a TCP connection with the bank server using a server port number provided at runtime. Once the connection is successfully created, the client transmits a **login request** containing a username and password. The system supports **three categories of users** on the client side: **Bank Customer, Bank Administrator, and Police**.

The bank server maintains two types of files:

- A **Login file**, which stores a fixed set of user credentials along with their corresponding roles.
- **Customer account files**, where each customer has a separate file that records their transaction history. For simplicity, assume the bank manages **exactly ten customers**, with one account file per customer.

After receiving a login request, the server authenticates the user and enables functionality based on the authenticated user role, as described below:

- **Bank Customer:** A customer can view the **current account balance** and retrieve a **mini statement** showing recent transactions.
- **Bank Administrator:** The administrator is authorized to **credit or debit funds** from any customer's account, similar to operations performed at a bank service counter. Every transaction must be recorded by appending details to the corresponding customer account file. Special care must be taken to prevent account balance underflow.
- **Police:** The police role is restricted to **view-only access**. Users in this category can inspect the **available balance of all customers** and may view the mini statement of any customer by specifying the customer's ID (i.e., a user with customer role).

The file formats used by the system are defined as follows:

- **Login File Format:**

User ID	Password	User_Type (C / A / P)
---------	----------	-----------------------

- **Customer Account File Format:**

Transaction_Date	Transaction Type (Credit/Debit) Available	Account_Balance
------------------	----------------------------------------------	-----------------

All interactions between the client and server must be carried out using **well-defined request and response message structures**. After completing each interaction, the TCP connection should be **closed gracefully on both ends**, ensuring proper release of socket resources.

The server IP address and port number must be supplied **via command-line arguments**, and no values should be hard-coded.

Command-line execution format:

- **Client:** <executable> <Server_IP_Address> <Server_Port_Number>
- **Server:** <executable> <Server_Port_Number>

Any additional assumptions required for implementation should be **clearly stated and logically justified**.

Application 6: Math Client–Server System

In this assignment, you will build a simple **client–server application** where a client sends arithmetic expressions to a server, and the server evaluates and returns the result.

System Overview

- The **server** runs on a specified port.
- The **client** connects to the server using IP and port provided via command line.
- The user enters arithmetic expressions (e.g., $1 + 2$, $3 * 4$).
- The client sends the expression to the server.
- The server evaluates the expression and sends the result back.
- The client displays the result and continues until terminated with **Ctrl+C**.

Supported operations (minimum requirement):

- Addition (+)
- Subtraction (-)
- Multiplication (*)

- Division (/)

Assumptions:

- Two integer operands separated by spaces.
- Handling more complex expressions is optional.

Provided Code

You are given the **client source code**. You must implement **three versions of the server** in C/C++.

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

#define MAX_INPUT_SIZE 256

int main(int argc, char *argv[])
{
    int sockfd, portnum, n;
    struct sockaddr_in server_addr;

    char inputbuf[MAX_INPUT_SIZE];
    if (argc < 3) {
        fprintf(stderr, "usage %s <server-ip-addr> <server-port>\n", argv[0]);
        exit(0);
    }

    portnum = atoi(argv[2]);

    /* Create client socket */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        fprintf(stderr, "ERROR opening socket\n");
        exit(1);
    }

    /* Fill in server address */
    bzero((char *) &server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    if (!inet_aton(argv[1], &server_addr.sin_addr))
    {
        fprintf(stderr, "ERROR invalid server IP address\n");
        exit(1);
    }
    server_addr.sin_port = htons(portnum);

    /* Connect to server */
    if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
    {
        fprintf(stderr, "ERROR connecting\n");
        exit(1);
    }
    printf("Connected to server\n");

    do
    {
        /* Ask user for message to send to server */
        printf("Please enter the message to the server: ");
```

```

bzero(inputbuf,MAX_INPUT_SIZE);
fgets(inputbuf,MAX_INPUT_SIZE-1,stdin);

/* Write to server */
n = write(sockfd,inputbuf,strlen(inputbuf));
if (n < 0)
{
    fprintf(stderr, "ERROR writing to socket\n");
    exit(1);
}

/* Read reply */
bzero(inputbuf,MAX_INPUT_SIZE);
n = read(sockfd,inputbuf,(MAX_INPUT_SIZE-1));
if (n < 0)
{
    fprintf(stderr, "ERROR reading from socket\n");
    exit(1);
}
printf("Server replied: %s\n",inputbuf);

} while(1);

return 0;
}

```

Server Implementations

Part 1: server1.c — Single-Process Server

- Handles **only one client at a time**.
- While a client is connected, additional clients must fail to connect.
- Once the client disconnects, the server accepts a new client.
- Server runs until terminated with Ctrl+C.

Part 2: server2.c — Multi-Process Server (Fork-based)

- Uses fork() to create a **new process per client**.
- Supports multiple concurrent clients.
- Server continues running regardless of client connections/disconnections.

Part 3: server3.c — Concurrent Server Using select()

- Single-process server using the select() system call.
- Manages multiple clients concurrently without forking.
- Behavior should match server2 from the client's perspective.

Submission Requirements

Submit a **tar.gz** file named with roll numbers (e.g., 15000001_15000002.tar.gz) containing:

- server1.c

- server2.c
- server3.c
- (Optional) Makefile or build script
- testcases.txt describing tested scenarios

Code should be **well-documented and cleanly written**.

Testing Criteria

Your servers will be tested for:

1. Correct arithmetic results.
 2. Proper client connection and reconnection.
 3. Concurrent client handling (server2 & server3).
 4. Rejection of multiple clients in server1.
 5. Process creation in server2 and no forking in server3.
-

Task Allocation Summary

Questions	Groups
Q1-Q7	All Groups
Application 1	1,7,13
Application 2	2,8
Application 3	3,9
Application 4	4,10
Application 5	5,11
Application 5	6,12
