

CS558: Computer Systems Lab
(January–May 2025)
Assignment - 4a

Submission Deadline: 11:55 pm, Wednesday, 9th April, 2025

Application #1: Client-Server programming using both TCP and UDP sockets

In this assignment, you are required to implement two C programs, namely server and client to communicate with each other based on both TCP and UDP sockets. The aim is to implement a simple 2 stage communication protocol.

Initially, the server will be waiting for a TCP connection from the client. Then, the client will connect to the server using the server's TCP port already known to the client. After successful connection, the client sends a Request Message (Type 1 message) to the server via TCP port to request a UDP port number from the server for future communication. After receiving the Request Message, the server selects a UDP port number and sends this port number back to the client as a Response Message (Type 2 Message) over the TCP connection. After this negotiation phase, the TCP connection on both the server and client should be closed gracefully, releasing the socket resource.

In the second phase, the client transmits a short Data Message (Type 3 message) over the earlier negotiated UDP port. The server will display the received Data Message and send a Data Response (Type 4 message) to indicate the successful reception. After this data transfer phase, both sides close their UDP sockets.

Message Format

The messages used to communicate contain the following fields:

Message Type	Message Length	Message
Message_type : integer	Message_length : integer	Message : character [MSG_LEN]

- Message_type : integer
- Message_length : integer
- Message : character [MSG_LEN], where MSG_LEN is an integer constant

A <Data Message> in Client will be a Type 3 message with some content in its message section.

Concurrent Server Requirement

You also require implementing a **"Concurrent Server"**, i.e., a server that accepts connections from multiple clients and serves all of them concurrently.

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Multi-threaded Server with Scheduling Policies

Your server must be a multi-threaded implementation that serves multiple clients concurrently. The following enhancements are required:

1. Implement a **request queue** where client requests are stored before processing.
2. Implement **two scheduling policies** for handling client requests:
 - **First-Come-First-Serve (FCFS)**: Serve clients in the order they arrive.
 - **Round-Robin (RR)**: Serve clients cyclically, ensuring fair resource allocation.
3. The server should accept an additional command-line argument to select the scheduling policy.

Performance measurement

- (a) Measure the throughput of TCP and UDP for 1-KB, 2-KB, 3-KB, ..., 32-KB messages. Plot the measured throughput as a function of message size.
- (b) Measure the throughput of TCP by sending 1 MB of data and 10 MB of data from one host to another. Do this in a loop that sends a message of some size—for example, 1024 iterations of a loop that sends 1-KB messages. Repeat the experiment with different message sizes and plot the results.

Command Line Prototype

Prototype for command line is as follows:

Prototypes for Client and Server

- Client: <executable code> <Server IP Address> <Server Port number>
- Server: <executable code> <Server Port number>

NB: Please make necessary and valid assumptions wherever required.

Application #2: Video Streaming Simulation using TCP, UDP, and Multithreading

In this assignment, you are required to implement a simulation of a video streaming service using both TCP and UDP protocols. The objective is to analyze how each protocol handles video data under different network conditions and compare their performance.

The application consists of a **Server** and a **Client**. The communication between the two involves two distinct phases. In the **Connection Phase**, the client initiates a TCP connection with the server to request video streaming. During this phase, the client sends a Type 1 Request Message that specifies the desired video resolution (for example, 480p, 720p, or 1080p). The server responds with a Type 2 Response Message that includes information about the selected video stream and its estimated bandwidth requirements. Once this exchange is complete, the TCP connection is closed.

The next phase is the **Video Streaming Phase**, where the client chooses whether to stream the video using TCP or UDP. If the client selects **TCP Mode**, the video data is transmitted over a new reliable TCP connection, ensuring zero packet loss. If **UDP Mode** is chosen, the data is sent via a UDP connection, emulating real-time video streaming where packet loss may occur. The server must be capable of handling multiple client connections concurrently, for which a multithreaded implementation is required.

To manage client requests efficiently, the server supports two scheduling policies. The **First-Come-First-Serve (FCFS)** policy serves clients in the order they arrive, while the **Round-Robin (RR)** policy ensures fairness by streaming video to clients in a cyclic manner. These policies help simulate different strategies of load balancing and resource allocation in a real streaming environment.

Performance Measurement

This simulation includes detailed performance analysis. Throughput and latency will be measured for both TCP and UDP modes under varying video resolutions (480p, 720p, and 1080p). For UDP, the simulation will introduce packet loss to study its impact on video quality. Resource utilization in terms of CPU and memory consumption will also be monitored and compared for both protocols. The following table outlines the key metrics:

Metric	Description
Throughput	Amount of video data transmitted per second (Mbps)
Latency	Time delay between request and first packet received (ms)
Packet Loss (UDP)	Percentage of packets lost during transmission
CPU Usage	Percentage of CPU consumed during streaming
Memory Usage	Amount of RAM used during streaming

Table 1: Performance Metrics for Evaluation

The collected results will be visualized using appropriate graphs, such as Throughput vs. Resolution and Latency vs. Resolution, to highlight the differences in protocol performance under various conditions.

Command Line Interface

The simulation must support the following command-line interface for both the client and server programs:

- **Client:** <executable_code> <Server_IP> <Server_Port> <Mode: TCP/UDP> <Resolution: 480p/720p/1080p>
- **Server:** <executable_code> <Server_Port> <Scheduling_Policy: FCFS/RR>

Additional Notes

The implementation must include proper error handling and socket management. Packet loss, network latency, and bandwidth limitations should be simulated programmatically

to reflect real-world networking conditions. Since this is a simulation, actual video files are not required—instead, mock data should be used to represent the video stream for simplicity and to reduce resource usage.

The server should be capable of supporting concurrent clients through multithreading, and the client should be able to switch between TCP and UDP streaming modes easily. Code should be well-documented with clear comments to support future modifications or enhancements.

Application #3: TCP and UDP Communication using Base64 Encoding in C

In this assignment, you are required to implement two C programs: a server and a client. The server must be capable of handling multiple clients concurrently and should support communication over both TCP and UDP sockets. The client will connect to the server using either TCP or UDP, based on user input, and will transmit Base64-encoded messages.

The primary objective is to create a simple communication protocol using Base64 encoding that supports both TCP (connection-oriented) and UDP (connectionless) communication methods. Additionally, the server must support concurrent communication with multiple clients using multithreading.

Functionality

1. Server

The server program should start by listening for incoming connections on a specified port, which should be passed via command-line arguments. It must support both TCP and UDP connections.

When a client connects via TCP, the server should create a new thread dedicated to handling that client. This ensures that multiple TCP clients can be served concurrently. On the other hand, since UDP is connectionless, the server handles incoming UDP messages in the main thread.

Upon receiving a message (from either TCP or UDP), the server decodes the Base64-encoded content and prints the original message. It then sends an acknowledgment (ACK) message back to the client. The server should remain active, able to serve multiple clients simultaneously. When a termination message (Type 3) is received from a client, the server should gracefully close that connection.

2. Client

The client connects to the server using either TCP or UDP, as specified by the user. It accepts text input from the user, encodes it using Base64 encoding, and sends the encoded message to the server.

After sending the message, the client waits for an acknowledgment (ACK) from the server. The client can continue sending messages until it chooses to terminate the session by sending a Type 3 (termination) message. The client should also handle session termination gracefully.

Message Format

Each message exchanged between the client and server consists of the following fields:

Field	Description
Message Type	Integer indicating the type of message (1, 2, or 3)
Message Content	Character array of fixed size (e.g., <code>MSG_LEN</code>)

Table 2: Structure of a Message

Message Types

There are three distinct types of messages that can be sent between client and server:

Message Type	Description
1	Base64-encoded message sent from client to server
2	Acknowledgment (ACK) message sent from server to client
3	Termination message to close the connection

Table 3: Types of Messages

Assumptions and Additional Notes

The server must be implemented in a way that allows it to handle multiple TCP clients simultaneously using multithreading. For UDP messages, since no persistent connection is maintained, all processing should be handled in the main thread.

Base64 encoding and decoding must follow the standard format. This includes splitting 24-bit groups of data into four 6-bit groups, using characters A–Z, a–z, 0–9, +, and / for encoding. Padding with = or == should be used for data that does not align to 24-bit boundaries.

Both the client and server must handle connection closures gracefully. Additionally, IP addresses and port numbers must be passed as command-line arguments to avoid hard-coded values.

Task Allocation Table

Group No.	Application
1, 4, 7, 11, 14, 18, 21, 24, 27	Application #1: Client-Server programming using both TCP and UDP sockets
2, 5, 9, 12, 15, 19, 22, 25	Application #2: Video Streaming Simulation using TCP, UDP, and Multithreading
3, 6, 10, 13, 16, 20, 23, 26	Application #3: TCP and UDP Communication using Base64 Encoding in C

Table 4: Group-wise task allocation for socket programming assignments