COMPUTER NETWORKS

LAB ASSIGNMENT-09

REDDIPALLI SAI CHARISH

CS22B1095

TCP Server:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <time.h>

#define SERVER_PORT 8080
#define MTU_SIZE 1500             // Maximum packet size (MTU)
#define PACKET_LOSS_PROB 0.05     // Probability of packet loss (5%)
#define MAX_LATENCY 100           // Max simulated latency per packet in ms
#define DISPLAY_PACKET_COUNT 20   // Packet count interval for display purposes

// Structure to store each packet's latency data
typedef struct {
   int packet_id;
   int latency;
} PacketLatency;

void receive_file(int client_sock) {
   FILE *file = fopen("received_audio.mp3", "wb");
   if (!file) {
      perror("File open failed");
      close(client_sock);
```

```
    return;
}

char buffer[MTU_SIZE];
int bytes_received;
int packets_received = 0, packets_lost = 0, retransmissions = 0;
int total_latency = 0;
float loss_chance;
srand(time(NULL));  // Seed for random number generator

printf("--- Starting file reception ---\n");

// Receive packets from client
while ((bytes_received = recv(client_sock, buffer, MTU_SIZE, 0)) > 0) {
    packets_received++;
    loss_chance = (float)rand() / RAND_MAX;  // Simulate packet loss

    // Simulate packet loss
    if (loss_chance < PACKET_LOSS_PROB) {
        packets_lost++;
        retransmissions++;
        printf("Packet %d lost, simulating retransmission...\n", packets_received);
        continue; // Skip writing this packet to simulate loss
    }

    // Simulate latency
    int latency = rand() % MAX_LATENCY + 1;
    total_latency += latency;
    //usleep(latency * 1000);  // Convert latency to microseconds

    fwrite(buffer, sizeof(char), bytes_received, file);
```

```
        // Optional: Display every N packets received
        if (packets_received % DISPLAY_PACKET_COUNT == 0) {
            printf("Received %d packets so far...\n", packets_received);
        }
    }

    if (bytes_received < 0) {
        perror("Error receiving data");
    } else {
        printf("File received successfully.\n");
    }

    // Calculate and print summary statistics
    float average_latency = packets_received > 0 ? (float)total_latency / packets_received : 0;
    printf("\n--- Transmission Summary ---\n");
    printf("Total Packets Received: %d\n", packets_received);
    printf("Total Packets Lost: %d\n", packets_lost);
    printf("Total Retransmissions: %d\n", retransmissions);
    printf("Average Latency per Packet: %.2f ms\n", average_latency);
    printf("Total Transmission Latency: %d ms\n", total_latency);
    printf("Efficiency : %.2f%%\n",(float)(packets_received-packets_lost)/packets_received*100);
    printf("Packet Loss Rate: %.2f%%\n", (float)packets_lost / packets_received * 100);
    printf("Reassemble reliability: %.2f%%\n", (float)(packets_received-
packets_lost)/packets_received*100);
    fclose(file);
}

int main() {
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
```

```
socklen_t addr_size = sizeof(client_addr);

// Create TCP socket
server_sock = socket(AF_INET, SOCK_STREAM, 0);
if (server_sock < 0) {
   perror("Socket creation failed");
   return 1;
}

// Set up server address structure
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(SERVER_PORT);
server_addr.sin_addr.s_addr = INADDR_ANY;

// Bind the socket to the specified IP and port
if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
   perror("Bind failed");
   close(server_sock);
   return 1;
}

// Listen for incoming connections
if (listen(server_sock, 1) < 0) {
   perror("Listen failed");
   close(server_sock);
   return 1;
}
printf("Server listening on port %d...\n", SERVER_PORT);

// Accept a client connection
client_sock = accept(server_sock, (struct sockaddr *)&client_addr, &addr_size);
```

```
    if (client_sock < 0) {
        perror("Accept failed");
        close(server_sock);
        return 1;
    }
    printf("Client connected.\n");

    // Start receiving file from the client
    receive_file(client_sock);

    // Close sockets
    close(client_sock);
    close(server_sock);
    return 0;
}
```

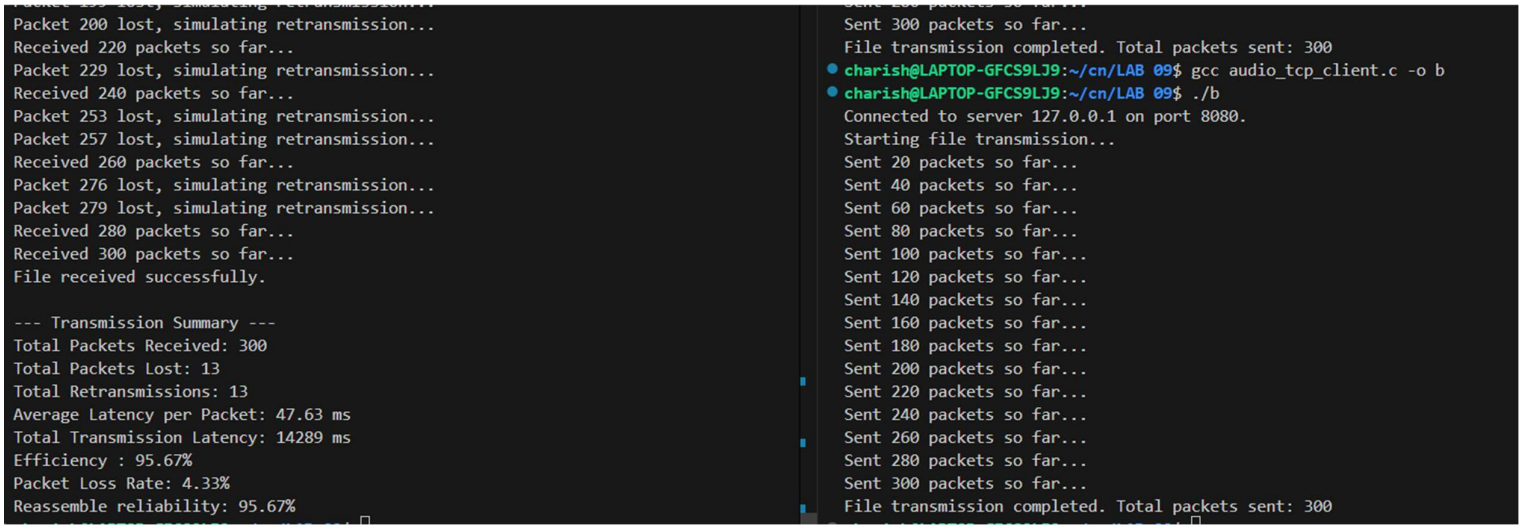
TCP Client:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <errno.h>

#define SERVER_IP "127.0.0.1"    // Change to server's IP if not running locally
#define SERVER_PORT 8080
#define MTU_SIZE 1500            // Maximum Transmission Unit (packet size)

void send_file(int server_sock, const char *filename) {
```

```
FILE *file = fopen(filename, "rb");
if (!file) {
    perror("File open failed");
    close(server_sock);
    return;
}

char buffer[MTU_SIZE];
int bytes_read;
int total_packets = 0;

printf("Starting file transmission...\n");

// Read from file and send data in chunks
while ((bytes_read = fread(buffer, sizeof(char), MTU_SIZE, file)) > 0) {
    if (send(server_sock, buffer, bytes_read, 0) < 0) {
        perror("Failed to send packet");
        fclose(file);
        close(server_sock);
        return;
    }
    total_packets++;

    // Optional: Display progress every few packets
    if (total_packets % 20 == 0) {
        printf("Sent %d packets so far...\n", total_packets);
    }
}

printf("File transmission completed. Total packets sent: %d\n", total_packets);
```

```
    fclose(file);
}

int main() {
    int client_sock;
    struct sockaddr_in server_addr;

    // Create TCP socket
    client_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (client_sock < 0) {
        perror("Socket creation failed");
        return 1;
    }

    // Set up server address structure
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
        perror("Invalid address/ Address not supported");
        close(client_sock);
        return 1;
    }

    // Connect to the server
    if (connect(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Connection to server failed");
        close(client_sock);
        return 1;
    }
    printf("Connected to server %s on port %d.\n", SERVER_IP, SERVER_PORT);
```

```
    // Send file to the server
    send_file(client_sock, "birds-chirping-75156.mp3");  // Replace with your audio file path

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



UDP Server:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include<sys/time.h>
#include <arpa/inet.h>
#include <errno.h>
#include <time.h>

#define SERVER_PORT 8081
#define MTU_SIZE 1500
#define PACKET_LOSS_PROB 0.05

typedef struct {
    int packet_id;
    int latency;
} PacketLatency;

void receive_file(int udp_sock, int total_packets) {
    FILE *file = fopen("received_audio_udp.mp3", "wb");
    if (!file) {
        perror("File open failed");
        close(udp_sock);
        return;
    }

    struct sockaddr_in client_addr;
```

```
    socklen_t addr_size = sizeof(client_addr);
    char buffer[MTU_SIZE];
    int bytes_received;
    int packets_received = 0, packets_lost = 0, retransmissions = 0;
    int total_latency = 0;

    srand(time(NULL));

    printf("Waiting for UDP packets...\n");

    // Set socket timeout
    struct timeval timeout;
    timeout.tv_sec = 5;  // Wait for 5 seconds of inactivity before timing out
    timeout.tv_usec = 0;
    setsockopt(udp_sock, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));

    while ((bytes_received = recvfrom(udp_sock, buffer, MTU_SIZE, 0, (struct sockaddr *)&client_addr,
&addr_size)) > 0) {
        packets_received++;
        printf("Received packet %d with %d bytes\n", packets_received, bytes_received);

        float loss_chance = (float)rand() / RAND_MAX;

        if (loss_chance < PACKET_LOSS_PROB) {
            printf("Simulated packet loss for packet %d\n", packets_received);
            packets_lost++;
            retransmissions++;
            continue;
        }

        int latency = rand() % 100 + 1;
```

```
        total_latency += latency;

        fwrite(buffer, sizeof(char), bytes_received, file);
    }

    if (bytes_received < 0 && errno == EWOULDBLOCK) {
        printf("Timeout reached. No more packets are being received.\n");
    }

    // Calculate average latency
    float average_latency = packets_received > 0 ? (float)total_latency / packets_received : 0;

    // Print the summary
    printf("\n--- Transmission Summary ---\n");
    printf("Total Packets Received: %d\n", packets_received);
    printf("Total Packets Lost: %d\n", packets_lost);
    printf("Total Retransmissions: %d\n", retransmissions);
    printf("Average Latency per Packet: %.2f ms\n", average_latency);
    printf("Total Transmission Latency: %d ms\n", total_latency);
    printf("Efficiency : %.2f%%\n",(float)(packets_received-packets_lost)/packets_received*100);
    printf("Packet Loss Rate: %.2f%%\n", (float)packets_lost / packets_received * 100);
    printf("Reassemble reliability: %.2f%%\n", (float)(packets_received-
packets_lost)/packets_received*100);

    fclose(file);
}

int main() {
    int udp_sock;
    struct sockaddr_in server_addr;
```

```
    udp_sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (udp_sock < 0) {
        perror("Socket creation failed");
        return 1;
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    if (bind(udp_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Bind failed");
        close(udp_sock);
        return 1;
    }

    printf("UDP Server listening on port %d...\n", SERVER_PORT);

    int total_packets = 1000;
    receive_file(udp_sock, total_packets);

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

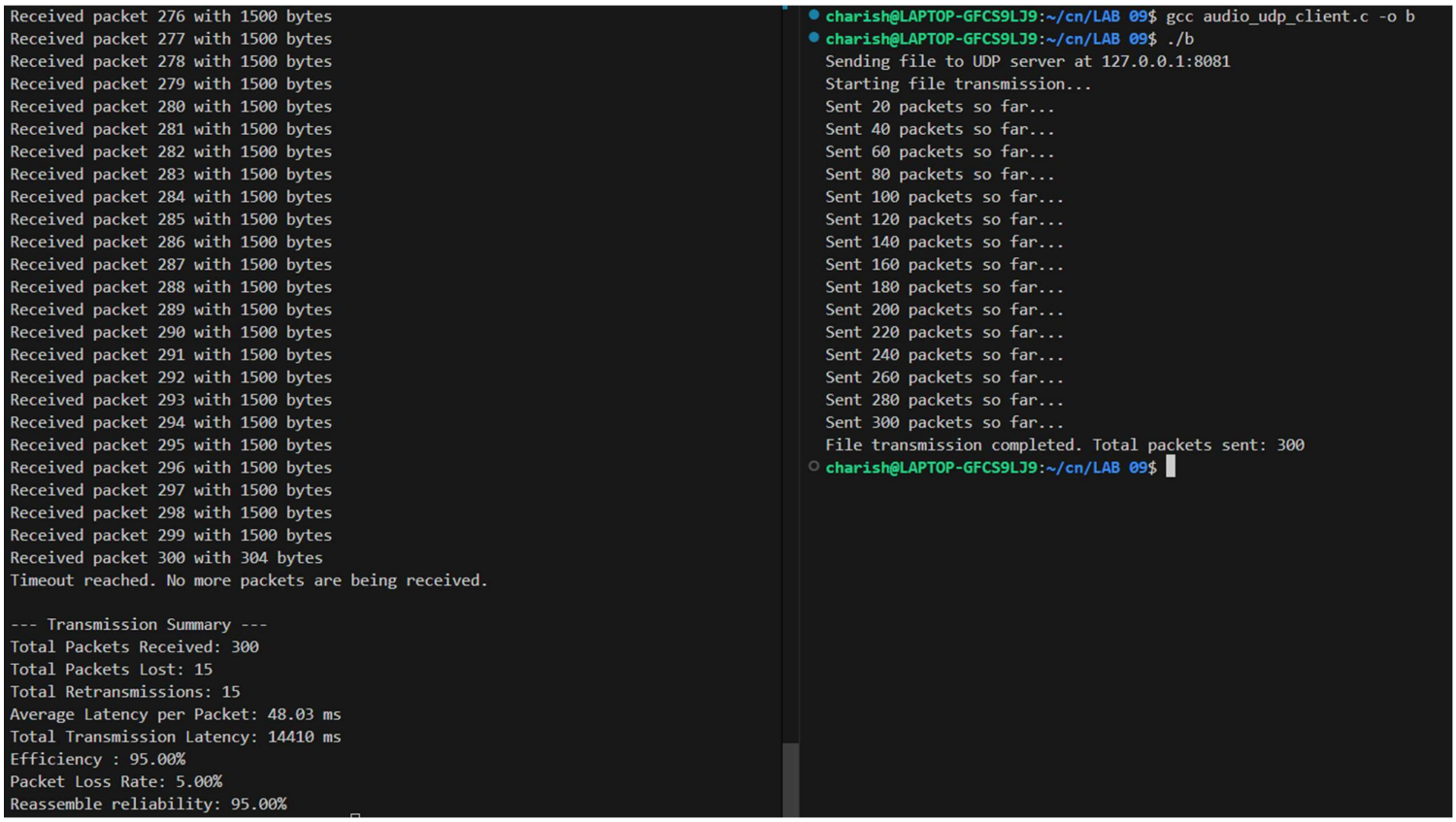
UDP Client:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

```
#include <errno.h>

#define SERVER_IP "127.0.0.1"    // Change to the server's IP if not running locally
#define SERVER_PORT 8081
#define MTU_SIZE 1500            // Maximum Transmission Unit (packet size)

void send_file(int udp_sock, const char *filename, struct sockaddr_in server_addr) {
   FILE *file = fopen(filename, "rb");
   if (!file) {
      perror("File open failed");
      close(udp_sock);
      return;
   }

   char buffer[MTU_SIZE];
   int bytes_read;
   int packet_id = 0;

   printf("Starting file transmission...\n");

   // Read from file and send data in packets
   while ((bytes_read = fread(buffer, sizeof(char), MTU_SIZE, file)) > 0) {
      // Add packet ID to buffer for identification (optional)
      memcpy(buffer, &packet_id, sizeof(packet_id));

      if (sendto(udp_sock, buffer, bytes_read + sizeof(packet_id), 0, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
         perror("Failed to send packet");
         fclose(file);
         close(udp_sock);
         return;
```

```
        }

        packet_id++;

        // Optional: Display progress every few packets
        if (packet_id % 20 == 0) {
            printf("Sent %d packets so far...\n", packet_id);
        }
    }

    printf("File transmission completed. Total packets sent: %d\n", packet_id);

    fclose(file);
}

int main() {
    int udp_sock;
    struct sockaddr_in server_addr;

    // Create UDP socket
    udp_sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (udp_sock < 0) {
        perror("Socket creation failed");
        return 1;
    }

    // Set up server address structure
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVER_PORT);
    if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
        perror("Invalid address/ Address not supported");
```

```
    close(udp_sock);
    return 1;
  }

  printf("Sending file to UDP server at %s:%d\n", SERVER_IP, SERVER_PORT);

  // Send file to the server
  send_file(udp_sock, "birds-chirping-75156.mp3", server_addr);  // Replace with your audio file path

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



Network fragmentation:

Smaller MTU sizes lead to more packets and thus increased fragmentation, which could impact transmission performance and reassembly.

Impact on TCP and UDP:

TCP - handles packet reassembly with error correction.

- Smaller MTU sizes may lead to more frequent acknowledgments and re-transmissions in the case of packet loss, affecting overall throughput.

UDP - does not guarantee packet delivery or reassembly.

- Smaller MTU sizes mean more packets are likely to be lost, especially if network conditions are suboptimal, leading to reduced reliability in UDP-based transmissions.

Larger MTUs result in fewer packets, which can be more efficient but may encounter issues if a packet is too large to be sent in one piece, especially in networks with lower MTU limits.

Smaller packets might reduce the impact of any single packet loss in TCP but could lead to delays in UDP where lost packets aren't re-transmitted. TCP's reassembly mechanism ensures that lost packets are recovered, while UDP's lack of such guarantees may lead to data loss in audio streams.

Packet Loss: Smaller MTUs mean more packets, increasing the risk of individual packet loss but potentially reducing the impact on overall data, as each packet carries less.

Latency: More packets can introduce latency, especially with TCP's need for acknowledgment.

Re-Transmissions: TCP will attempt to resend lost packets, while UDP will not. With smaller MTUs, TCP re-transmissions may occur more frequently.

Real-Time Audio: UDP is preferred due to lower latency, but small MTU sizes could lead to packet loss, which impacts audio quality. Balancing MTU size is critical for minimizing latency while ensuring data is transmitted reliably.

Non-Real-Time Audio: TCP may be better suited for ensuring complete data delivery, even with a higher MTU size, as latency is less critical.