ASSIGNMENT 4a

Sarthak Suresh Chandurkar (244101069)

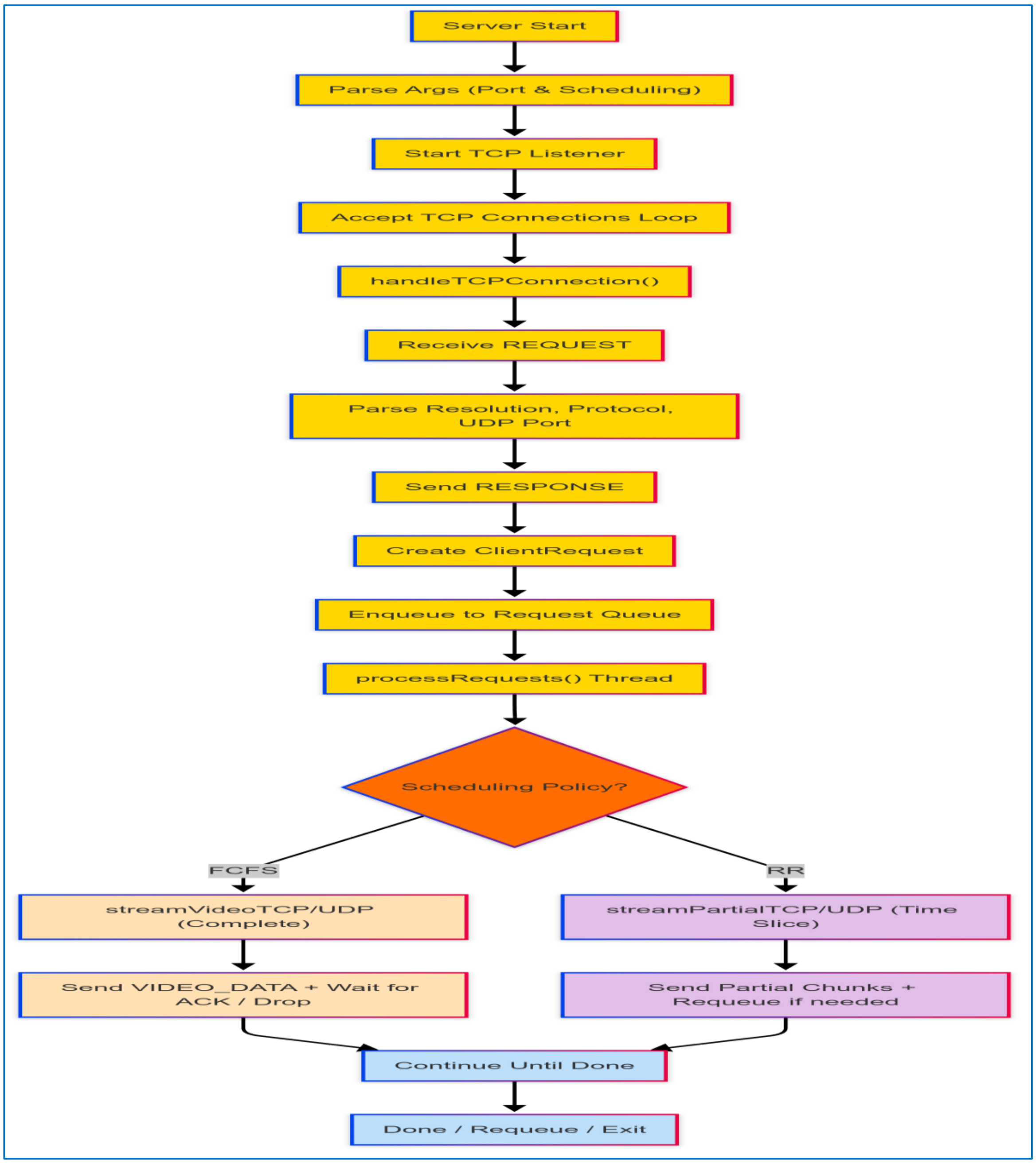
Abhishek Kumar Tiwari (244101067)

Deepjyoti Brahma (244101014)

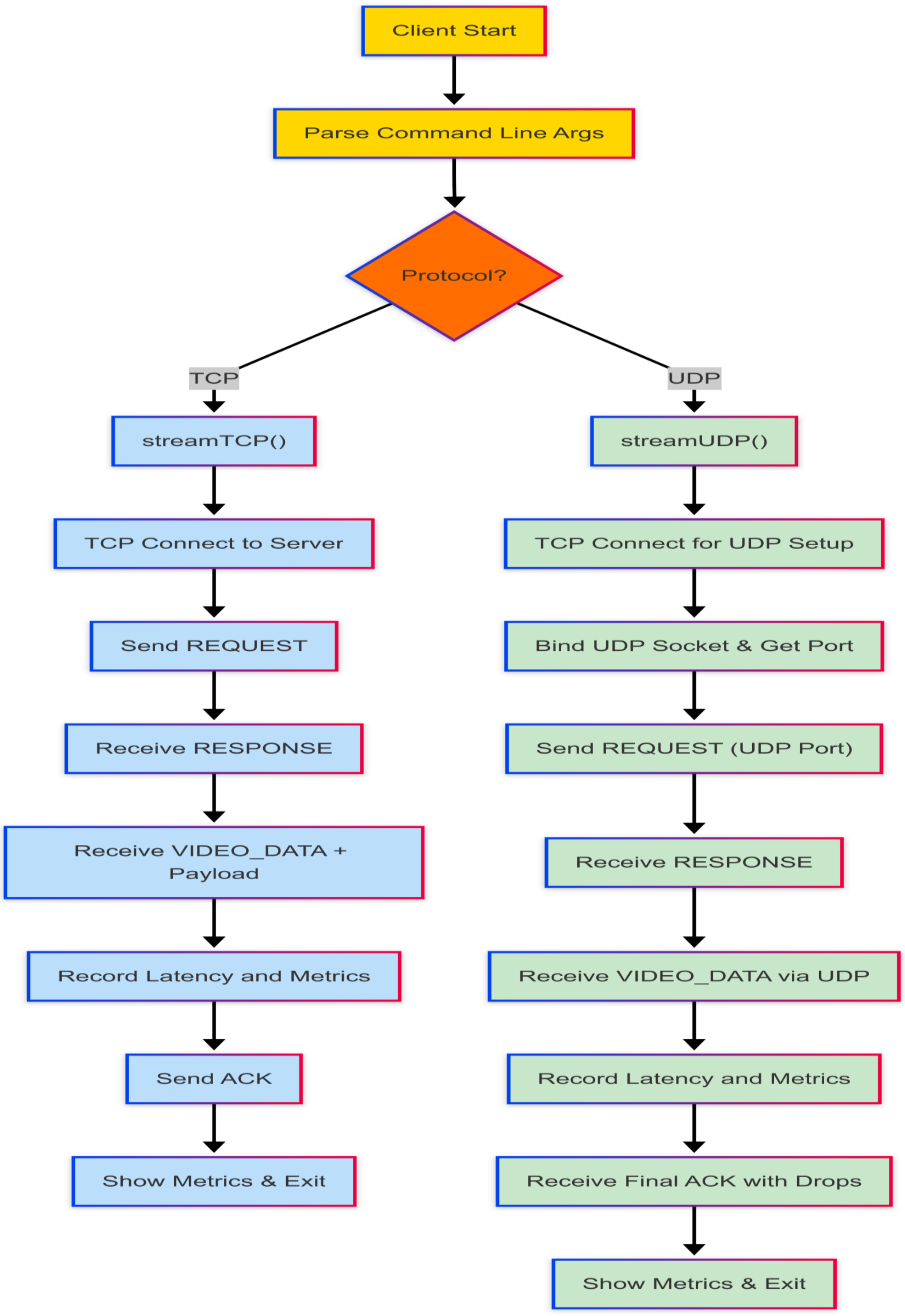
Application #2: Video Streaming Simulation using TCP, UDP, and Multithreading

Code Flow Diagram

Server:


Server Start
Parse Args (Port & Scheduling)
Start TCP Listener
Accept TCP Connections Loop
handleTCPConnection()
Receive REQUEST
Parse Resolution, Protocol, UDP Port
Send RESPONSE
Create ClientRequest
Enqueue to Request Queue
processRequests() Thread
Scheduling Policy?
FCFS
RR
streamVideoTCP/UDP (Complete)
streamPartialTCP/UDP (Time Slice)
Send VIDEO_DATA + Wait for ACK / Drop
Send Partial Chunks + Requeue if needed
Continue Until Done
Done / Requeue / Exit

Client:

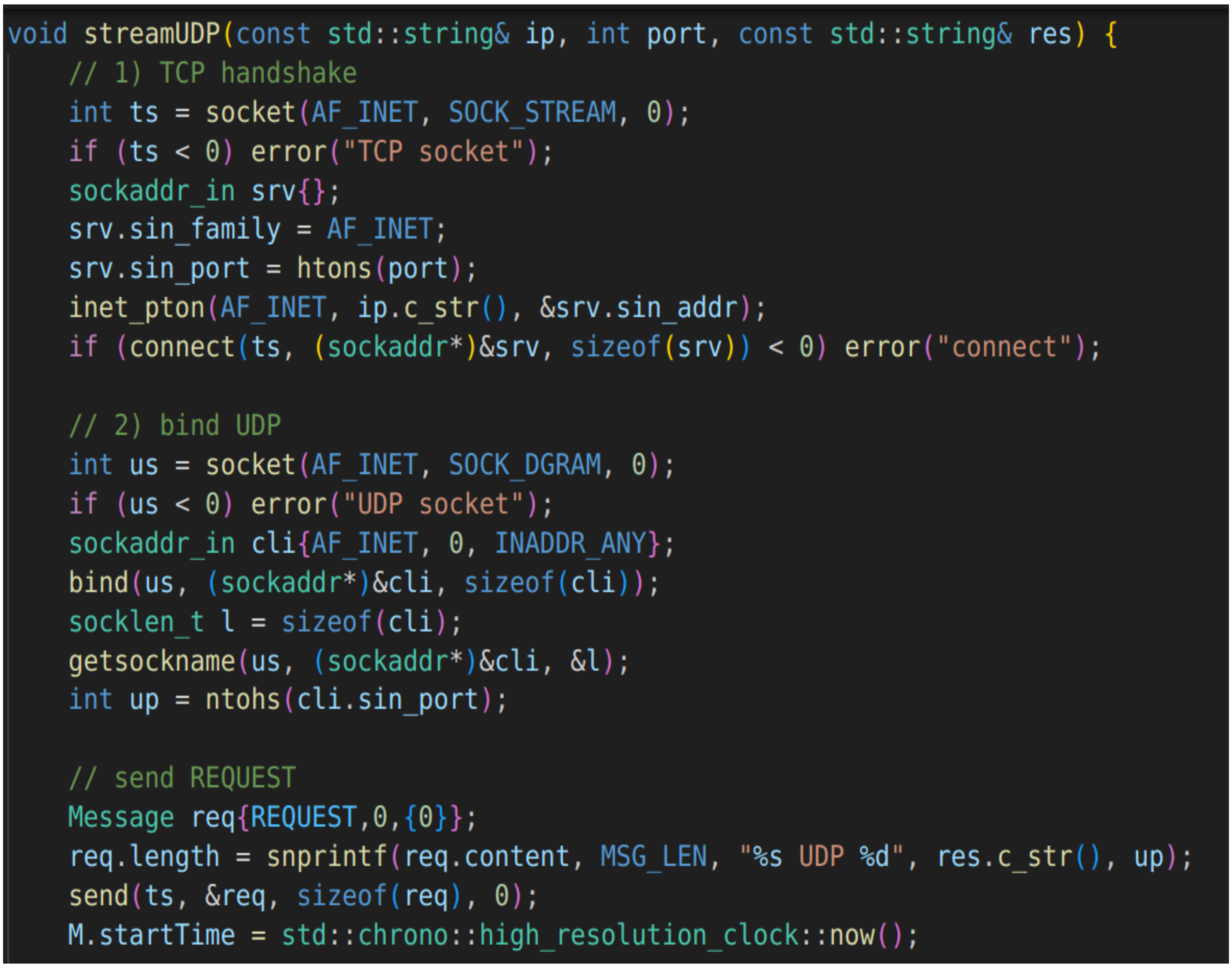


Client Start
Parse Command Line Args
Protocol?
TCP
UDP
streamTCP()
TCP Connect to Server
Send REQUEST
Receive RESPONSE
Receive VIDEO_DATA + Payload
Record Latency and Metrics
Send ACK
Show Metrics & Exit
streamUDP()
TCP Connect for UDP Setup
Bind UDP Socket & Get Port
Send REQUEST (UDP Port)
Receive RESPONSE
Receive VIDEO_DATA via UDP
Record Latency and Metrics
Receive Final ACK with Drops
Show Metrics & Exit

Connection Phase - Handshake

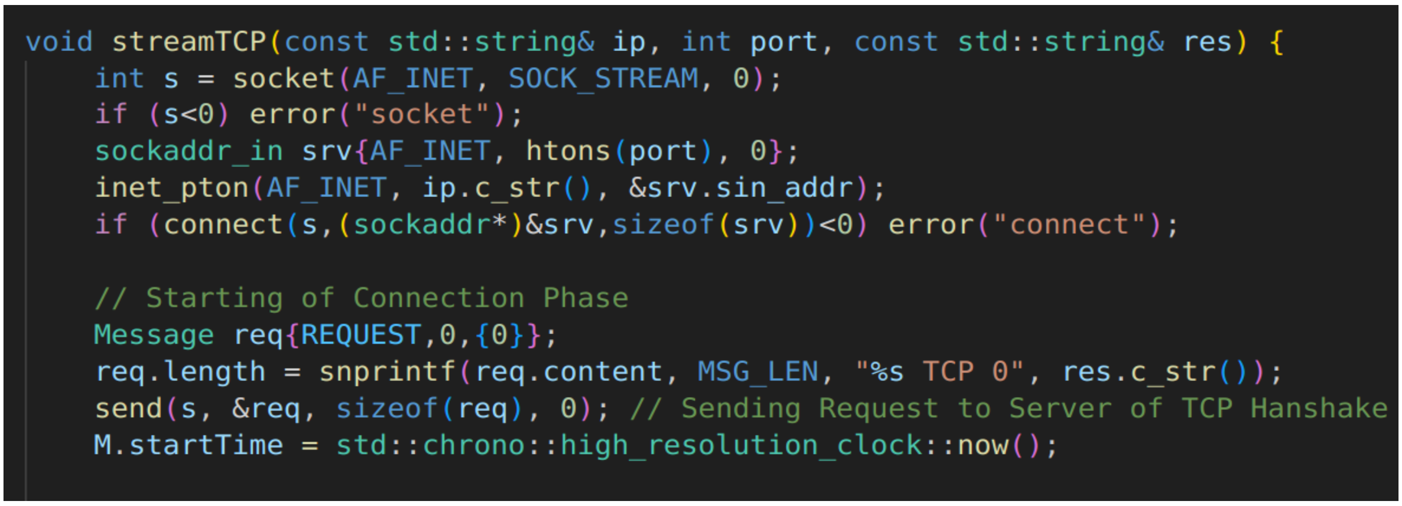
1) Request to Server

If User wants UDP Streaming



```cpp
void streamUDP(const std::string& ip, int port, const std::string& res) {
    // 1) TCP handshake
    int ts = socket(AF_INET, SOCK_STREAM, 0);
    if (ts < 0) error("TCP socket");
    sockaddr_in srv{};
    srv.sin_family = AF_INET;
    srv.sin_port = htons(port);
    inet_pton(AF_INET, ip.c_str(), &srv.sin_addr);
    if (connect(ts, (sockaddr*)&srv, sizeof(srv)) < 0) error("connect");

    // 2) bind UDP
    int us = socket(AF_INET, SOCK_DGRAM, 0);
    if (us < 0) error("UDP socket");
    sockaddr_in cli{AF_INET, 0, INADDR_ANY};
    bind(us, (sockaddr*)&cli, sizeof(cli));
    socklen_t l = sizeof(cli);
    getsockname(us, (sockaddr*)&cli, &l);
    int up = ntohs(cli.sin_port);

    // send REQUEST
    Message req{REQUEST,0,{0}};
    req.length = snprintf(req.content, MSG_LEN, "%s UDP %d", res.c_str(), up);
    send(ts, &req, sizeof(req), 0);
    M.startTime = std::chrono::high_resolution_clock::now();
```

This function first establishes a TCP connection to a server, then creates and binds a UDP socket to an available local port. It retrieves the UDP port number and sends a TCP message (req) to the server requesting a resource (res) and indicating the client's UDP port. Finally, it marks the start time for streaming.

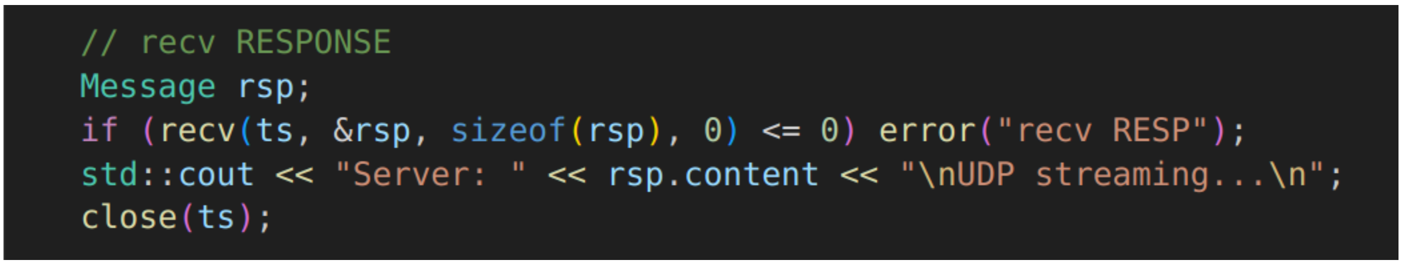


```cpp
void streamTCP(const std::string& ip, int port, const std::string& res) {
    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s<0) error("socket");
    sockaddr_in srv{AF_INET, htons(port), 0};
    inet_pton(AF_INET, ip.c_str(), &srv.sin_addr);
    if (connect(s,(sockaddr*)&srv,sizeof(srv))<0) error("connect");

    // Starting of Connection Phase
    Message req{REQUEST,0,{0}};
    req.length = snprintf(req.content, MSG_LEN, "%s TCP 0", res.c_str());
    send(s, &req, sizeof(req), 0); // Sending Request to Server of TCP Hanshake
    M.startTime = std::chrono::high_resolution_clock::now();
```

This function creates a TCP socket, connects to the server, and sends a request message (res) indicating TCP streaming. It starts the connection phase and records the start time for measuring stream performance.

2) Response from server

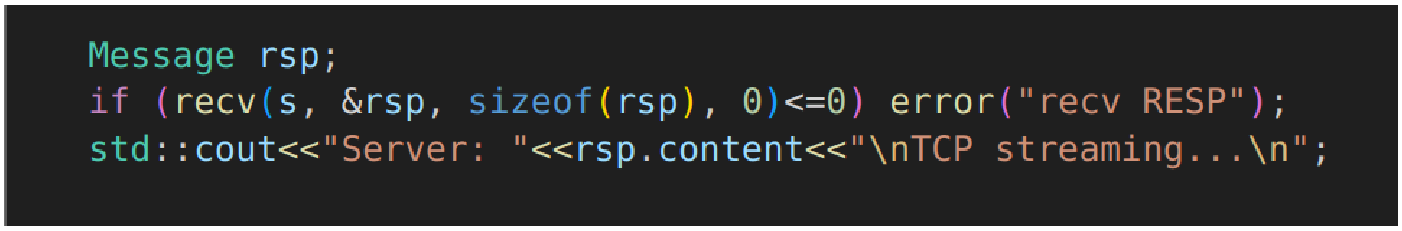
In UDP Streaming



```cpp
    // recv RESPONSE
    Message rsp;
    if (recv(ts, &rsp, sizeof(rsp), 0) <= 0) error("recv RESP");
    std::cout << "Server: " << rsp.content << "\nUDP streaming...\n";
    close(ts);
```

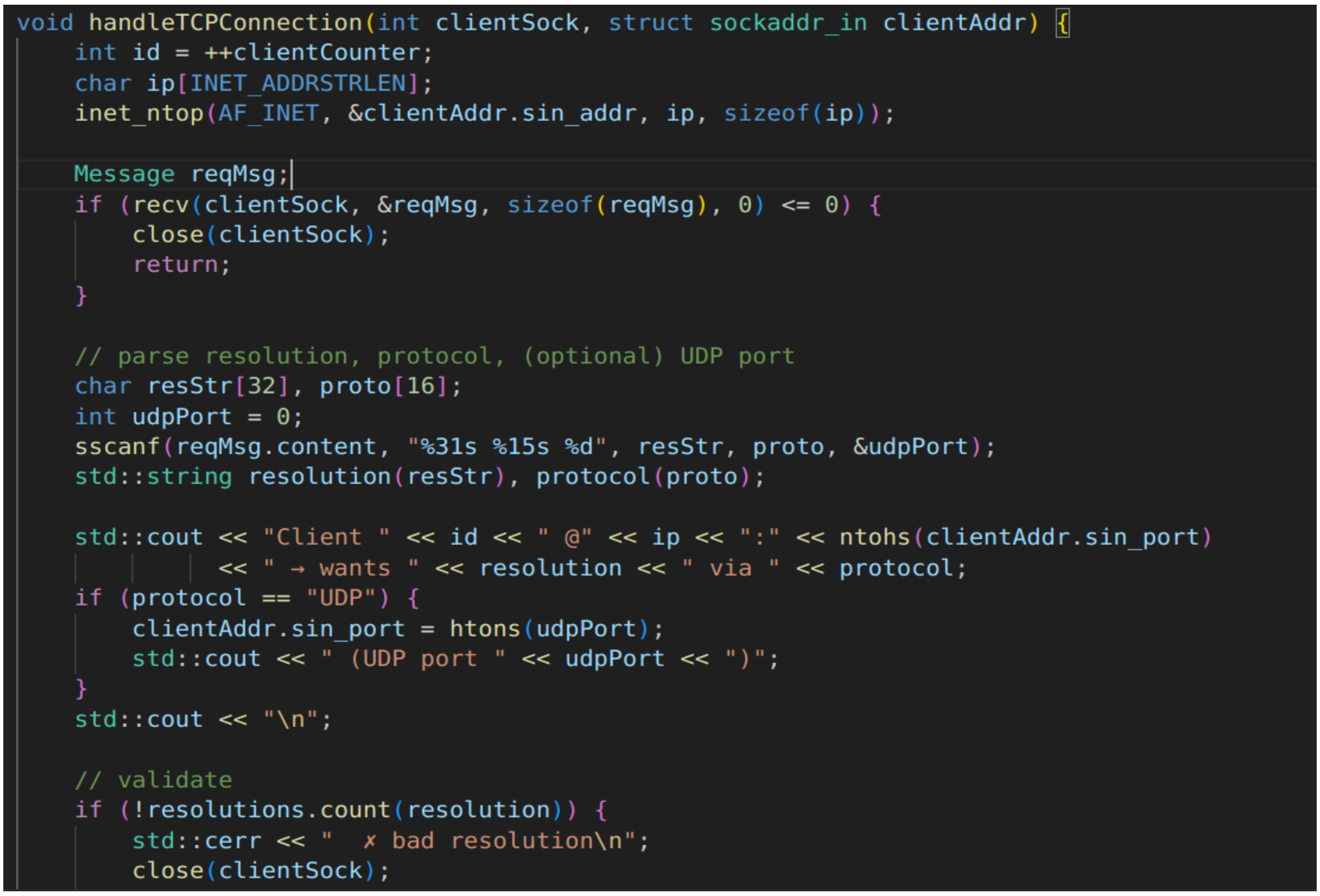
The client receives the server's response over the TCP socket (ts). If the reception fails, it throws an error. On success, it prints the server's response, indicates the start of UDP streaming, and closes the TCP connection.

In TCP Streaming

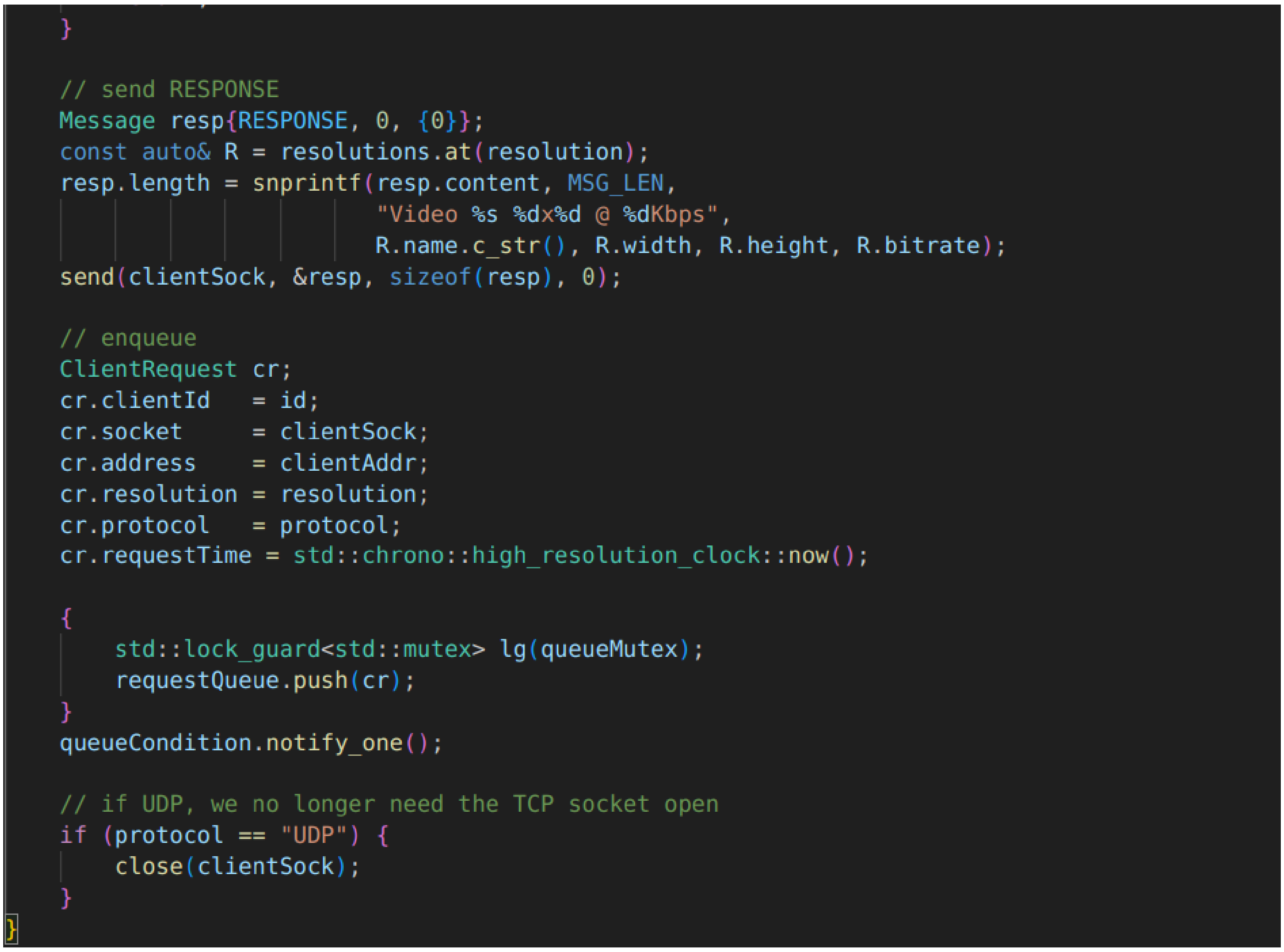


```cpp
    Message rsp;
    if (recv(s, &rsp, sizeof(rsp), 0)<=0) error("recv RESP");
    std::cout<<"Server: "<<rsp.content<<"\nTCP streaming...\n";
```

After sending the request, the client waits to receive a response (rsp) from the server. If receiving fails, it throws an error. Otherwise, it prints the server's response and a message indicating that TCP streaming is starting.



```cpp
void handleTCPConnection(int clientSock, struct sockaddr_in clientAddr) {
    int id = ++clientCounter;
    char ip[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &clientAddr.sin_addr, ip, sizeof(ip));

    Message reqMsg;
    if (recv(clientSock, &reqMsg, sizeof(reqMsg), 0) <= 0) {
        close(clientSock);
        return;
    }

    // parse resolution, protocol, (optional) UDP port
    char resStr[32], proto[16];
    int udpPort = 0;
    sscanf(reqMsg.content, "%31s %15s %d", resStr, proto, &udpPort);
    std::string resolution(resStr), protocol(proto);

    std::cout << "Client " << id << " @" << ip << ":" << ntohs(clientAddr.sin_port)
              << " → wants " << resolution << " via " << protocol;
    if (protocol == "UDP") {
        clientAddr.sin_port = htons(udpPort);
        std::cout << " (UDP port " << udpPort << ")";
    }
    std::cout << "\n";

    // validate
    if (!resolutions.count(resolution)) {
        std::cerr << "  ✗ bad resolution\n";
        close(clientSock);
```



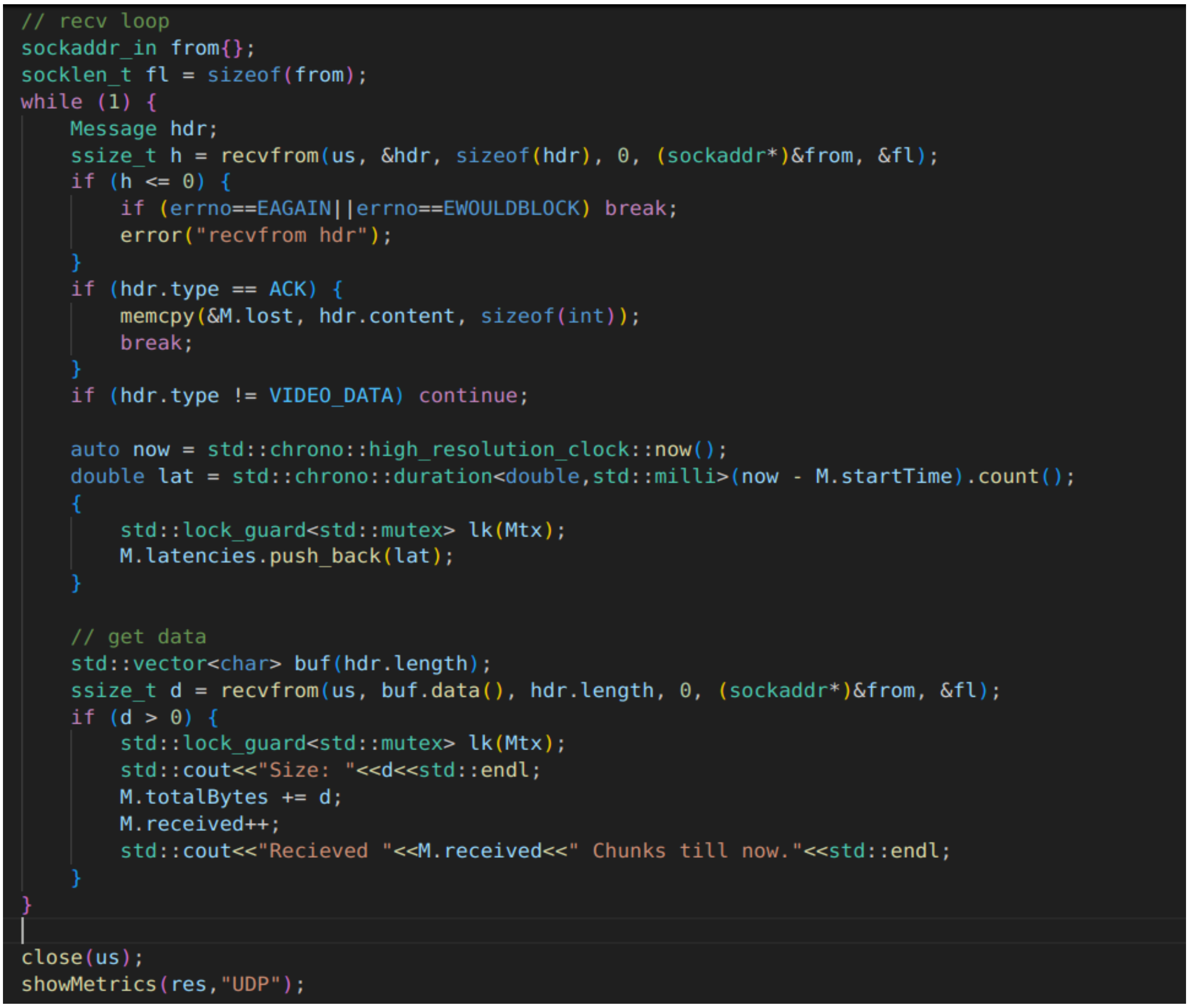
```cpp
    }

    // send RESPONSE
    Message resp{RESPONSE, 0, {0}};
    const auto& R = resolutions.at(resolution);
    resp.length = snprintf(resp.content, MSG_LEN,
                           "Video %s %dx%d @ %dKbps",
                           R.name.c_str(), R.width, R.height, R.bitrate);
    send(clientSock, &resp, sizeof(resp), 0);

    // enqueue
    ClientRequest cr;
    cr.clientId   = id;
    cr.socket     = clientSock;
    cr.address    = clientAddr;
    cr.resolution = resolution;
    cr.protocol   = protocol;
    cr.requestTime = std::chrono::high_resolution_clock::now();

    {
        std::lock_guard<std::mutex> lg(queueMutex);
        requestQueue.push(cr);
    }
    queueCondition.notify_one();

    // if UDP, we no longer need the TCP socket open
    if (protocol == "UDP") {
        close(clientSock);
    }
}
```

This function handles an incoming TCP client connection:

- It receives a request message and parses the desired video resolution, protocol (TCP/UDP), and optional UDP port.
- If the resolution is invalid, the connection is closed.
- It sends a response with video details back to the client.
- A ClientRequest is created and added to the global queue for processing.
- If the client chose UDP, the TCP socket is closed immediately; otherwise, it stays open for TCP streaming.

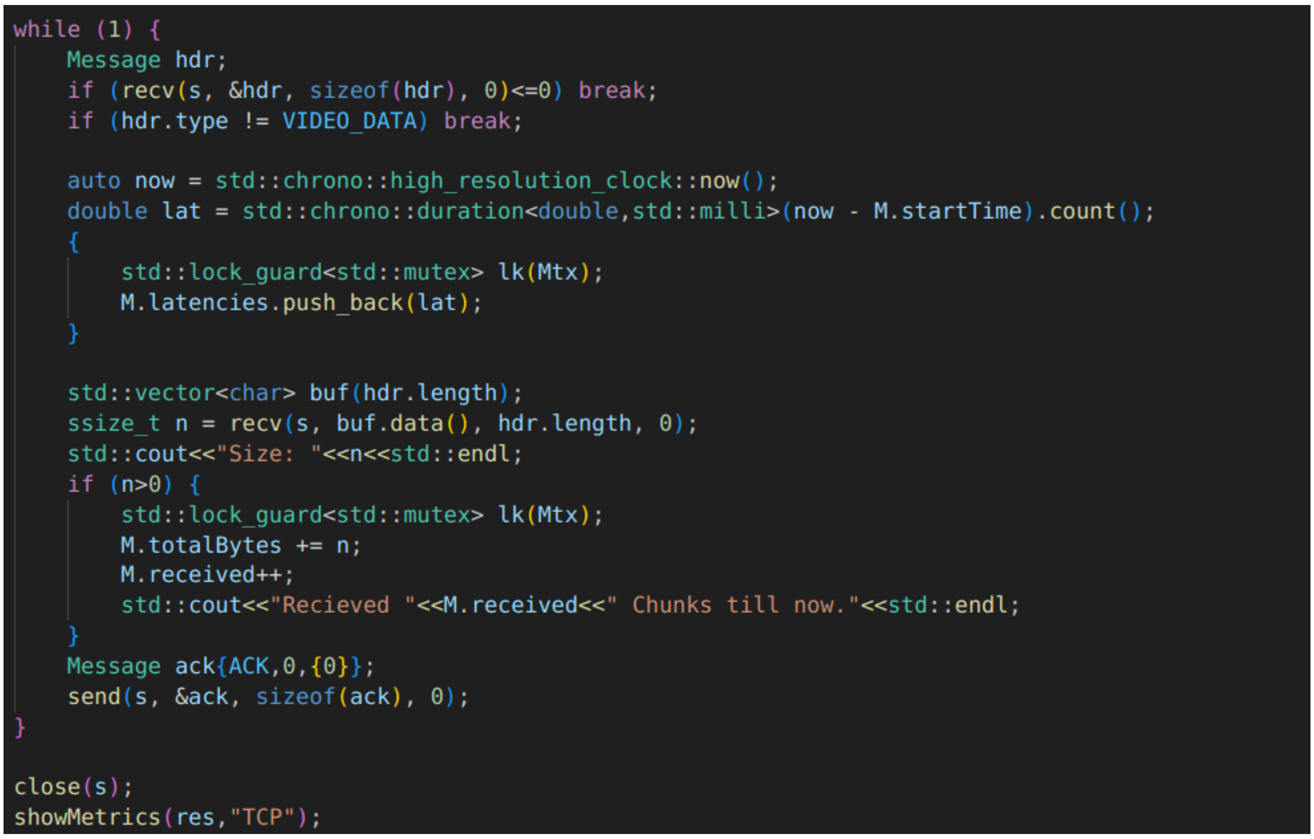
Video Streaming Phase

On Client Side



```cpp
// recv loop
sockaddr_in from{};
socklen_t fl = sizeof(from);
while (1) {
    Message hdr;
    ssize_t h = recvfrom(us, &hdr, sizeof(hdr), 0, (sockaddr*)&from, &fl);
    if (h <= 0) {
        if (errno==EAGAIN||errno==EWOULDBLOCK) break;
        error("recvfrom hdr");
    }
    if (hdr.type == ACK) {
        memcpy(&M.lost, hdr.content, sizeof(int));
        break;
    }
    if (hdr.type != VIDEO_DATA) continue;

    auto now = std::chrono::high_resolution_clock::now();
    double lat = std::chrono::duration<double,std::milli>(now - M.startTime).count();
    {
        std::lock_guard<std::mutex> lk(Mtx);
        M.latencies.push_back(lat);
    }

    // get data
    std::vector<char> buf(hdr.length);
    ssize_t d = recvfrom(us, buf.data(), hdr.length, 0, (sockaddr*)&from, &fl);
    if (d > 0) {
        std::lock_guard<std::mutex> lk(Mtx);
        std::cout<<"Size: "<<d<<std::endl;
        M.totalBytes += d;
        M.received++;
        std::cout<<"Recieved "<<M.received<<" Chunks till now."<<std::endl;
    }
}

close(us);
showMetrics(res,"UDP");
```

This loop continuously receives UDP packets. If a packet is an ACK, it updates the lost packet count and exits. If it's VIDEO_DATA, it records latency, then receives the actual data chunk. It updates metrics like total bytes and chunk count. The loop ends on timeout or error, then closes the UDP socket and displays performance metrics.

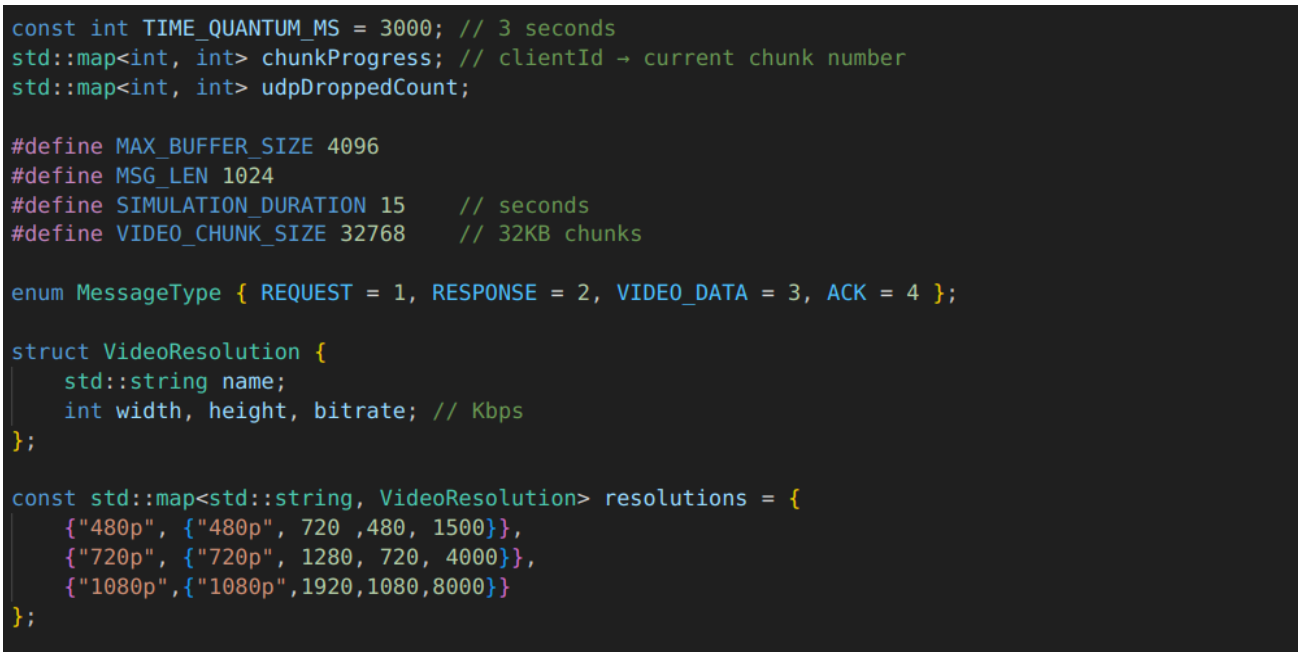


```
while (1) {
    Message hdr;
    if (recv(s, &hdr, sizeof(hdr), 0)<=0) break;
    if (hdr.type != VIDEO_DATA) break;

    auto now = std::chrono::high_resolution_clock::now();
    double lat = std::chrono::duration<double,std::milli>(now - M.startTime).count();
    {
        std::lock_guard<std::mutex> lk(Mtx);
        M.latencies.push_back(lat);
    }

    std::vector<char> buf(hdr.length);
    ssize_t n = recv(s, buf.data(), hdr.length, 0);
    std::cout<<"Size: "<<n<<std::endl;
    if (n>0) {
        std::lock_guard<std::mutex> lk(Mtx);
        M.totalBytes += n;
        M.received++;
        std::cout<<"Recieved "<<M.received<<" Chunks till now."<<std::endl;
    }
    Message ack{ACK,0,{0}};
    send(s, &ack, sizeof(ack), 0);
}

close(s);
showMetrics(res,"TCP");
```

This loop handles TCP video streaming. It receives a VIDEO_DATA header, records latency, then receives the actual chunk. Metrics like total bytes and chunk count are updated. An ACK is sent back after each chunk. The loop ends on error or non-VIDEO_DATA, then the socket is closed and performance metrics are shown.

On Server Side

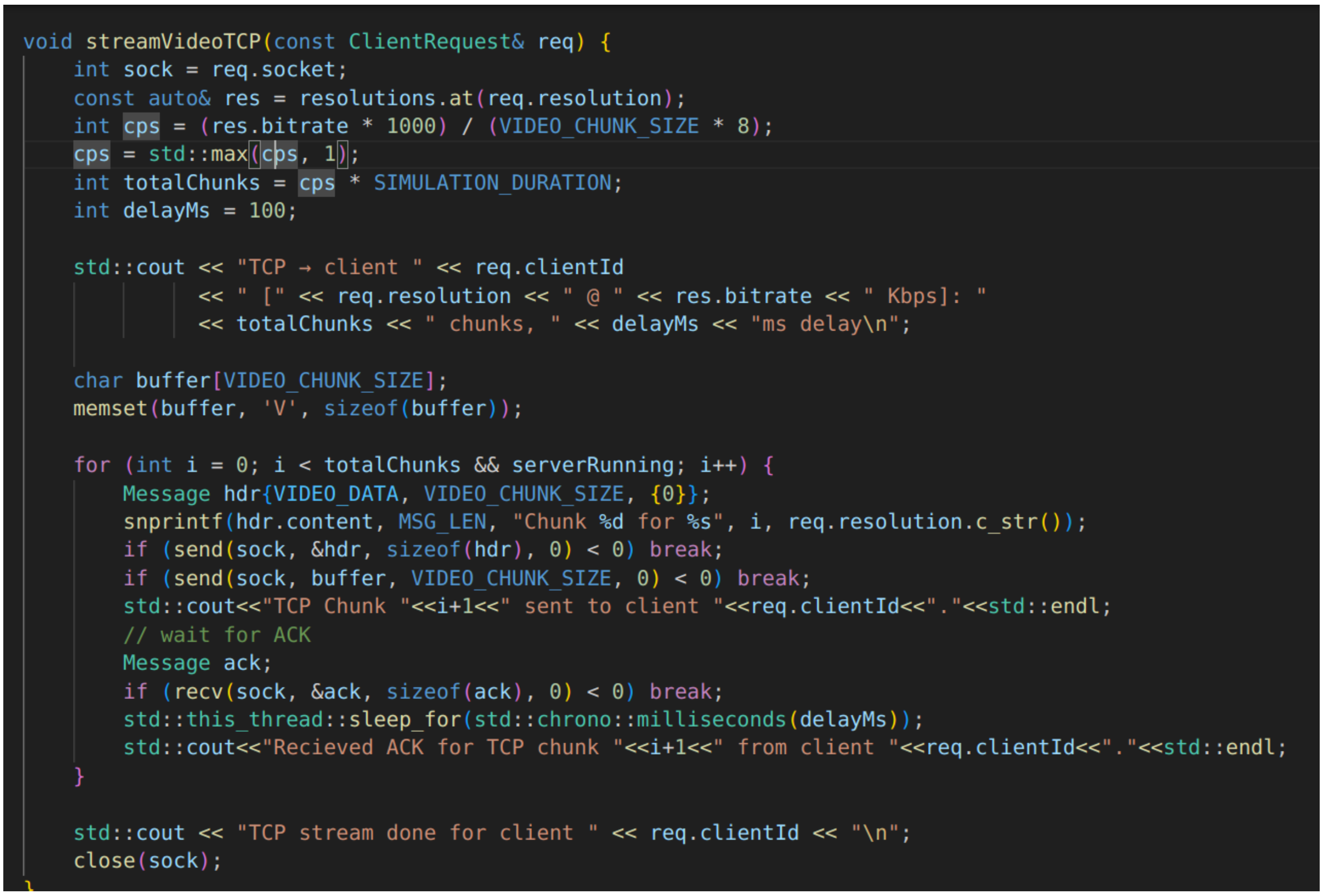
Assumptions:



```
const int TIME_QUANTUM_MS = 3000; // 3 seconds
std::map<int, int> chunkProgress; // clientId → current chunk number
std::map<int, int> udpDroppedCount;

#define MAX_BUFFER_SIZE 4096
#define MSG_LEN 1024
#define SIMULATION_DURATION 15    // seconds
#define VIDEO_CHUNK_SIZE 32768    // 32KB chunks

enum MessageType { REQUEST = 1, RESPONSE = 2, VIDEO_DATA = 3, ACK = 4 };

struct VideoResolution {
    std::string name;
    int width, height, bitrate; // Kbps
};

const std::map<std::string, VideoResolution> resolutions = {
    {"480p", {"480p", 720 ,480, 1500}},
    {"720p", {"720p", 1280, 720, 4000}},
    {"1080p",{"1080p",1920,1080,8000}}
};
```

This setup defines constants and data structures for video streaming:

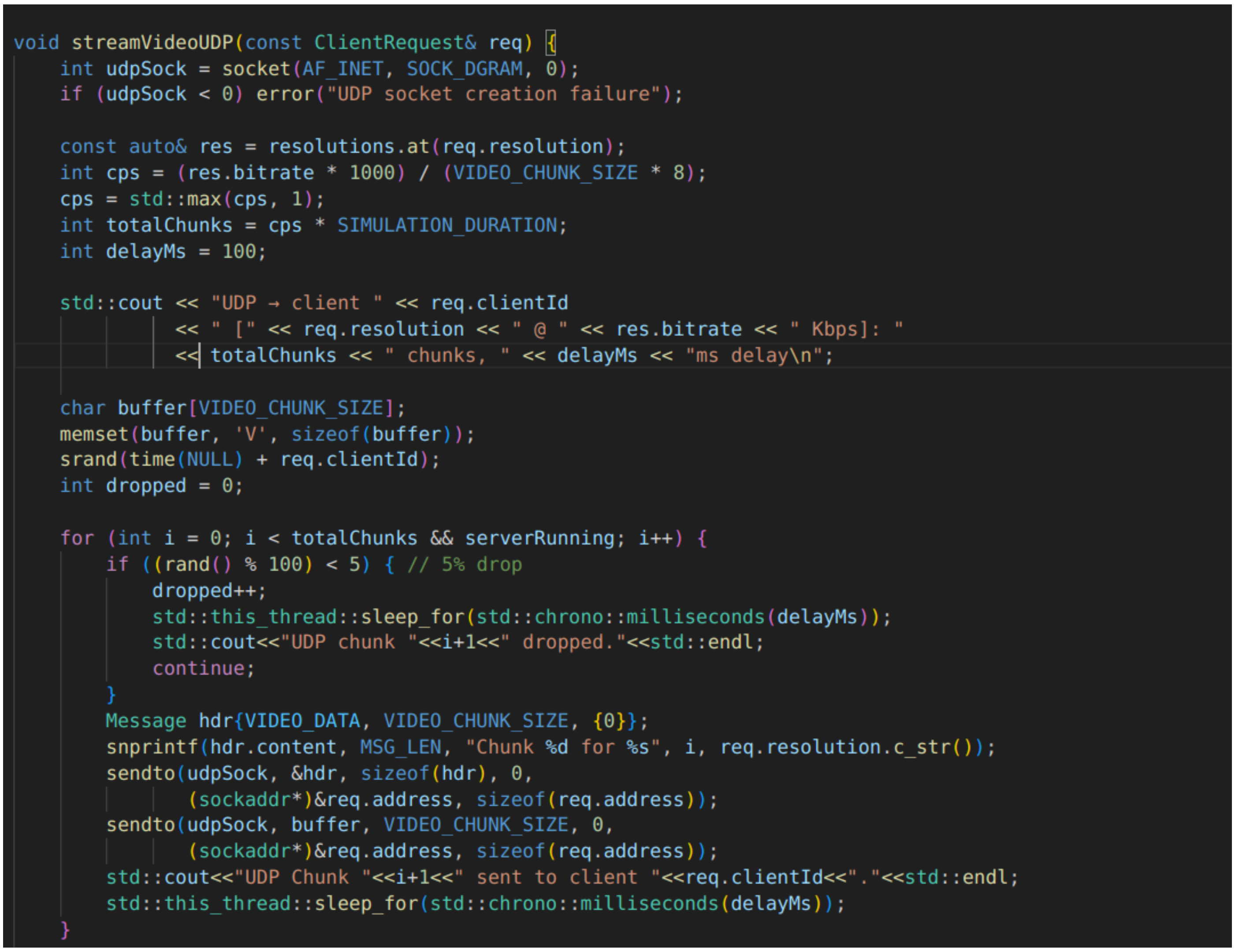
- TIME_QUANTUM_MS: Time slice per client in RR (3 sec).
- chunkProgress: Tracks current chunk for each client.
- udpDroppedCount: Tracks dropped UDP chunks per client.
- MessageType: Enum for message types (request, response, data, ACK).
- resolutions: Map of video resolution names to their properties (dimensions and bitrate).



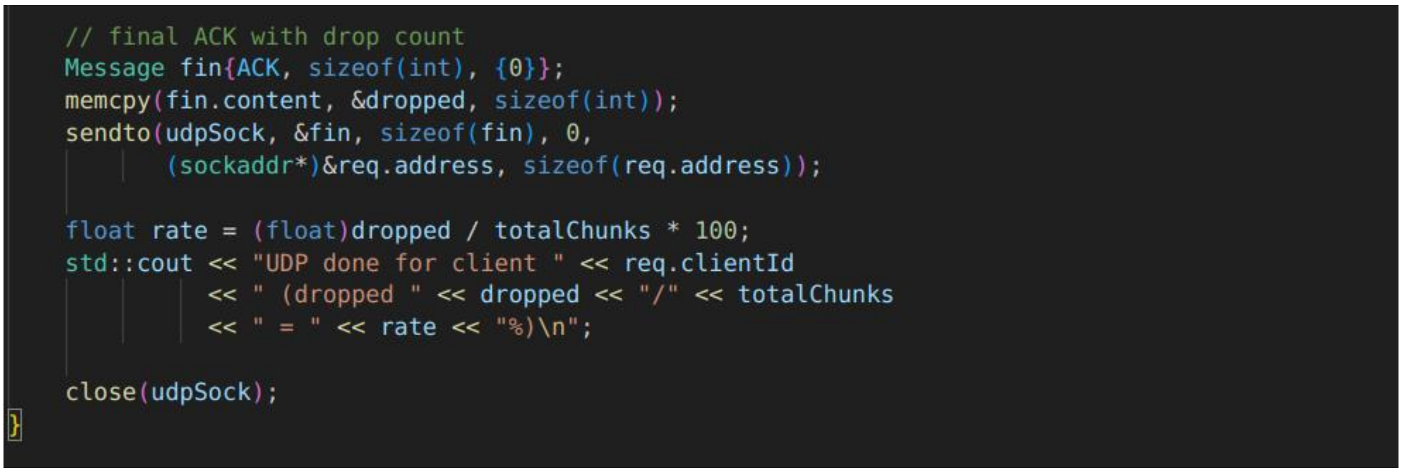
```
void streamVideoTCP(const ClientRequest& req) {
    int sock = req.socket;
    const auto& res = resolutions.at(req.resolution);
    int cps = (res.bitrate * 1000) / (VIDEO_CHUNK_SIZE * 8);
    cps = std::max(cps, 1);
    int totalChunks = cps * SIMULATION_DURATION;
    int delayMs = 100;

    std::cout << "TCP → client " << req.clientId
              << " [" << req.resolution << " @ " << res.bitrate << " Kbps]: "
              << totalChunks << " chunks, " << delayMs << "ms delay\n";

    char buffer[VIDEO_CHUNK_SIZE];
    memset(buffer, 'V', sizeof(buffer));

    for (int i = 0; i < totalChunks && serverRunning; i++) {
        Message hdr{VIDEO_DATA, VIDEO_CHUNK_SIZE, {0}};
        snprintf(hdr.content, MSG_LEN, "Chunk %d for %s", i, req.resolution.c_str());
        if (send(sock, &hdr, sizeof(hdr), 0) < 0) break;
        if (send(sock, buffer, VIDEO_CHUNK_SIZE, 0) < 0) break;
        std::cout<<"TCP Chunk "<<i+1<<" sent to client "<<req.clientId<<"."<<std::endl;
        // wait for ACK
        Message ack;
        if (recv(sock, &ack, sizeof(ack), 0) < 0) break;
        std::this_thread::sleep_for(std::chrono::milliseconds(delayMs));
        std::cout<<"Recieved ACK for TCP chunk "<<i+1<<" from client "<<req.clientId<<"."<<std::endl;
    }

    std::cout << "TCP stream done for client " << req.clientId << "\n";
    close(sock);
```

This function streams video over TCP to a client:

- It calculates the number of chunks based on resolution bitrate and sets a delay between chunks.
- For each chunk, it sends a VIDEO_DATA header and 32KB data buffer.
- It waits for an ACK from the client after each chunk and sleeps for a short delay.
- The loop continues until all chunks are sent or an error occurs.
- Finally, it closes the socket and ends the stream.



```cpp
void streamVideoUDP(const ClientRequest& req) {
    int udpSock = socket(AF_INET, SOCK_DGRAM, 0);
    if (udpSock < 0) error("UDP socket creation failure");

    const auto& res = resolutions.at(req.resolution);
    int cps = (res.bitrate * 1000) / (VIDEO_CHUNK_SIZE * 8);
    cps = std::max(cps, 1);
    int totalChunks = cps * SIMULATION_DURATION;
    int delayMs = 100;

    std::cout << "UDP → client " << req.clientId
              << " [" << req.resolution << " @ " << res.bitrate << " Kbps]: "
              << totalChunks << " chunks, " << delayMs << "ms delay\n";

    char buffer[VIDEO_CHUNK_SIZE];
    memset(buffer, 'V', sizeof(buffer));
    srand(time(NULL) + req.clientId);
    int dropped = 0;

    for (int i = 0; i < totalChunks && serverRunning; i++) {
        if ((rand() % 100) < 5) { // 5% drop
            dropped++;
            std::this_thread::sleep_for(std::chrono::milliseconds(delayMs));
            std::cout<<"UDP chunk "<<i+1<<" dropped."<<std::endl;
            continue;
        }
        Message hdr{VIDEO_DATA, VIDEO_CHUNK_SIZE, {0}};
        snprintf(hdr.content, MSG_LEN, "Chunk %d for %s", i, req.resolution.c_str());
        sendto(udpSock, &hdr, sizeof(hdr), 0,
               (sockaddr*)&req.address, sizeof(req.address));
        sendto(udpSock, buffer, VIDEO_CHUNK_SIZE, 0,
               (sockaddr*)&req.address, sizeof(req.address));
        std::cout<<"UDP Chunk "<<i+1<<" sent to client "<<req.clientId<<"."<<std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(delayMs));
    }
```



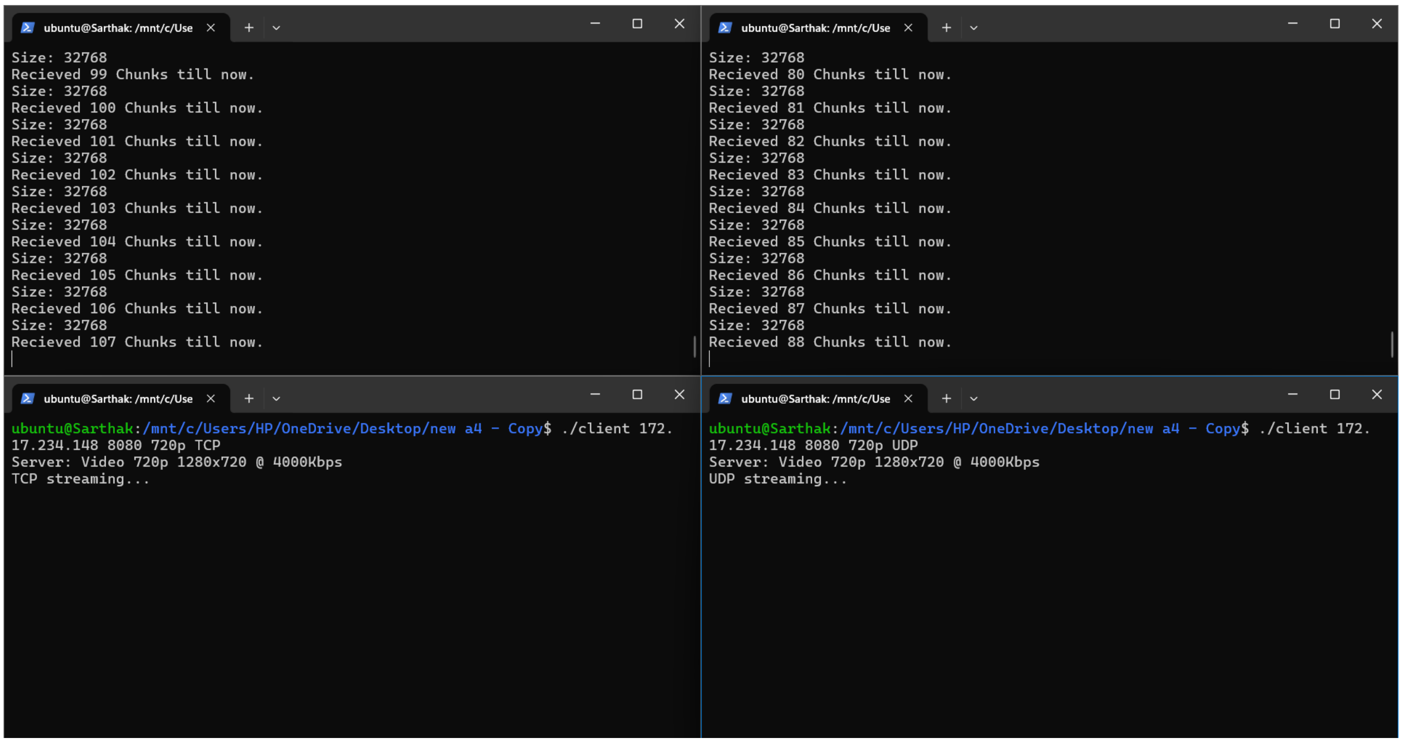
```cpp
    // final ACK with drop count
    Message fin{ACK, sizeof(int), {0}};
    memcpy(fin.content, &dropped, sizeof(int));
    sendto(udpSock, &fin, sizeof(fin), 0,
           (sockaddr*)&req.address, sizeof(req.address));

    float rate = (float)dropped / totalChunks * 100;
    std::cout << "UDP done for client " << req.clientId
              << " (dropped " << dropped << "/" << totalChunks
              << " = " << rate << "%)\n";

    close(udpSock);
}
```

This function streams video over UDP:

- It calculates the total chunks and delay based on video bitrate.
- For each chunk, there's a 5% chance it's "dropped" (simulated loss).
- If not dropped, it sends a VIDEO_DATA header and the data chunk via sendto.
- After all chunks, it sends a final ACK with the total number of dropped chunks.
- It prints the drop rate and closes the UDP socket.

Output:

In FCFS mode for both TCP and UDP

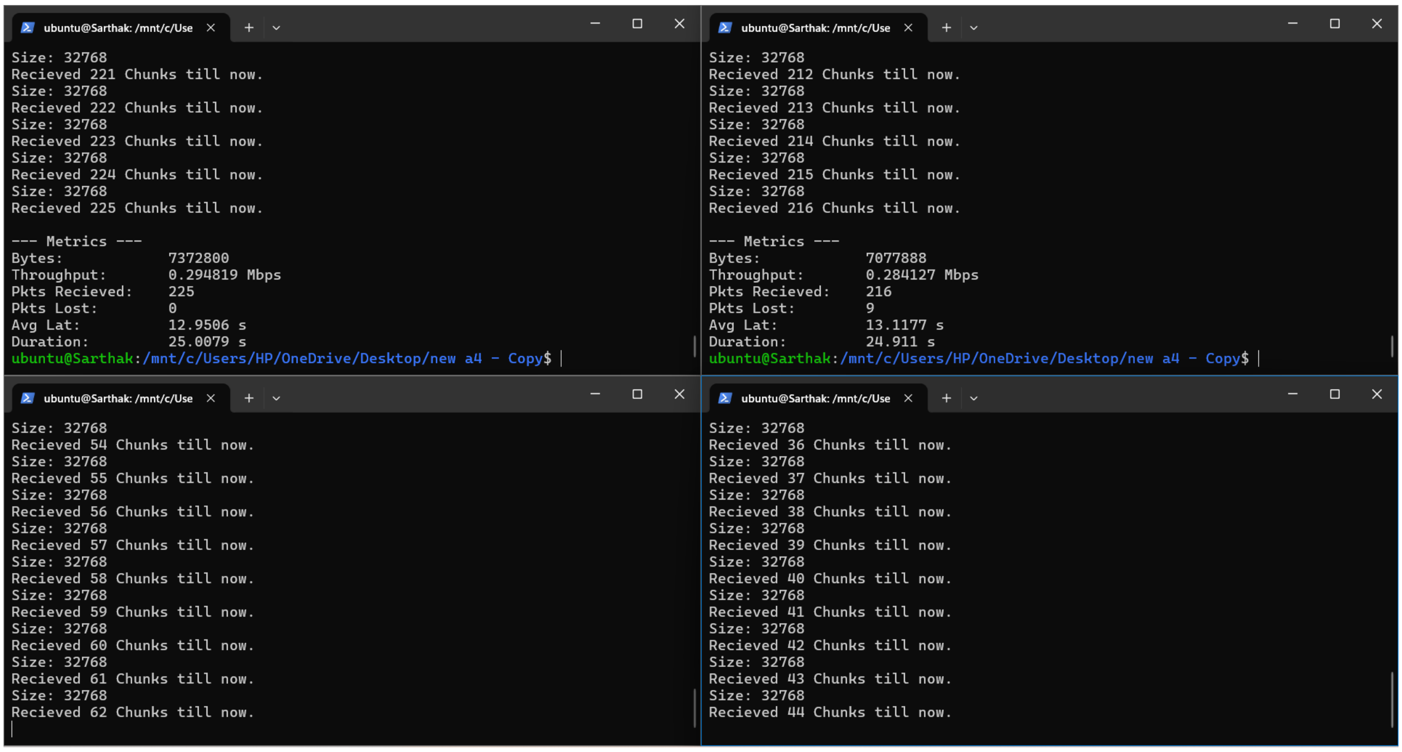


```
Size: 32768
Recieved 99 Chunks till now.
Size: 32768
Recieved 100 Chunks till now.
Size: 32768
Recieved 101 Chunks till now.
Size: 32768
Recieved 102 Chunks till now.
Size: 32768
Recieved 103 Chunks till now.
Size: 32768
Recieved 104 Chunks till now.
Size: 32768
Recieved 105 Chunks till now.
Size: 32768
Recieved 106 Chunks till now.
Size: 32768
Recieved 107 Chunks till now.
```

```
Size: 32768
Recieved 80 Chunks till now.
Size: 32768
Recieved 81 Chunks till now.
Size: 32768
Recieved 82 Chunks till now.
Size: 32768
Recieved 83 Chunks till now.
Size: 32768
Recieved 84 Chunks till now.
Size: 32768
Recieved 85 Chunks till now.
Size: 32768
Recieved 86 Chunks till now.
Size: 32768
Recieved 87 Chunks till now.
Size: 32768
Recieved 88 Chunks till now.
```

```
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$ ./client 172.
17.234.148 8080 720p TCP
Server: Video 720p 1280x720 @ 4000Kbps
TCP streaming...
```

```
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$ ./client 172.
17.234.148 8080 720p UDP
Server: Video 720p 1280x720 @ 4000Kbps
UDP streaming...
```



```
Size: 32768
Recieved 221 Chunks till now.
Size: 32768
Recieved 222 Chunks till now.
Size: 32768
Recieved 223 Chunks till now.
Size: 32768
Recieved 224 Chunks till now.
Size: 32768
Recieved 225 Chunks till now.

--- Metrics ---
Bytes:            7372800
Throughput:       0.294819 Mbps
Pkts Recieved:    225
Pkts Lost:        0
Avg Lat:          12.9506 s
Duration:         25.0079 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

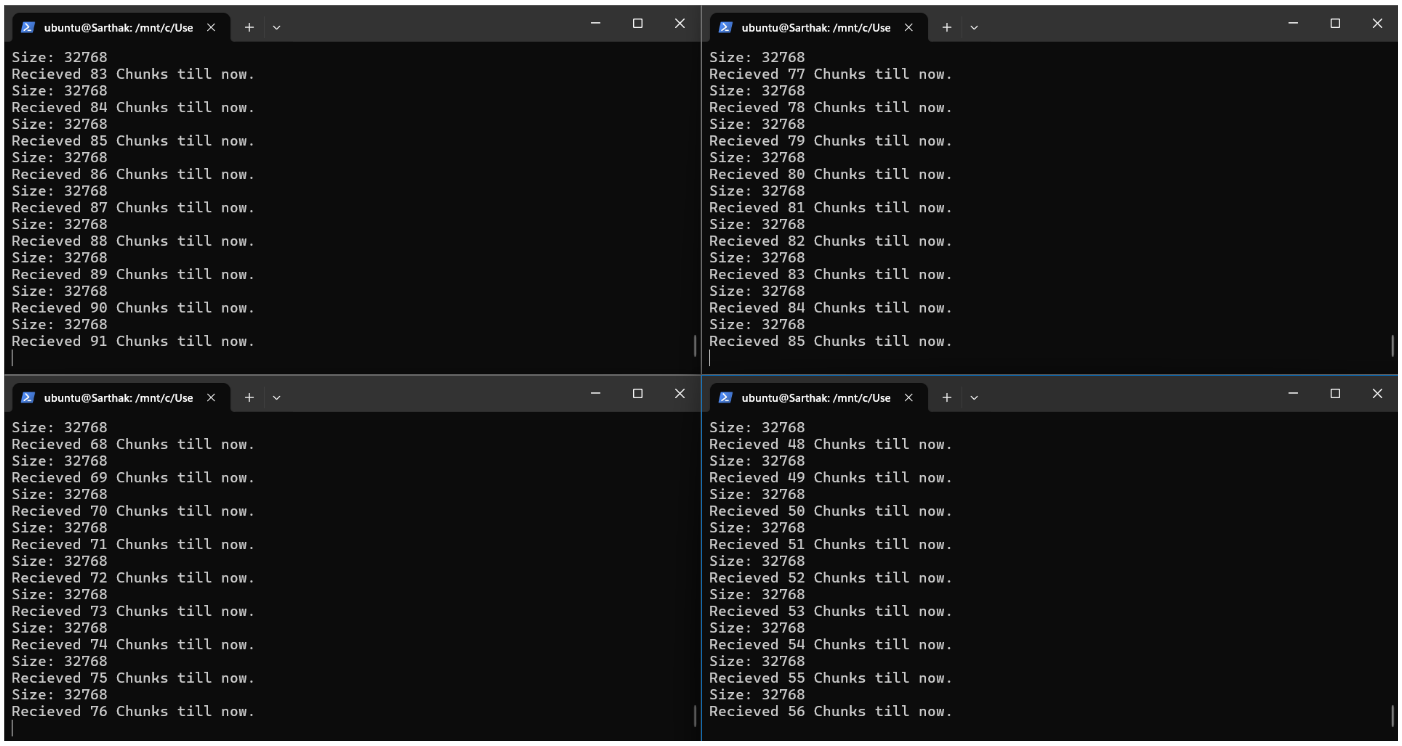
```
Size: 32768
Recieved 212 Chunks till now.
Size: 32768
Recieved 213 Chunks till now.
Size: 32768
Recieved 214 Chunks till now.
Size: 32768
Recieved 215 Chunks till now.
Size: 32768
Recieved 216 Chunks till now.

--- Metrics ---
Bytes:            7077888
Throughput:       0.284127 Mbps
Pkts Recieved:    216
Pkts Lost:        9
Avg Lat:          13.1177 s
Duration:         24.911 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

```
Size: 32768
Recieved 54 Chunks till now.
Size: 32768
Recieved 55 Chunks till now.
Size: 32768
Recieved 56 Chunks till now.
Size: 32768
Recieved 57 Chunks till now.
Size: 32768
Recieved 58 Chunks till now.
Size: 32768
Recieved 59 Chunks till now.
Size: 32768
Recieved 60 Chunks till now.
Size: 32768
Recieved 61 Chunks till now.
Size: 32768
Recieved 62 Chunks till now.
```

```
Size: 32768
Recieved 36 Chunks till now.
Size: 32768
Recieved 37 Chunks till now.
Size: 32768
Recieved 38 Chunks till now.
Size: 32768
Recieved 39 Chunks till now.
Size: 32768
Recieved 40 Chunks till now.
Size: 32768
Recieved 41 Chunks till now.
Size: 32768
Recieved 42 Chunks till now.
Size: 32768
Recieved 43 Chunks till now.
Size: 32768
Recieved 44 Chunks till now.
```

Maximum Threads than cun run at a time are 2. Hence 2 client will wait in FCFS. After the completion of first 2 clients remanining 2 will be allocated with threads.

In RR mode for both TCP and UDP



```
Size: 32768
Recieved 83 Chunks till now.
Size: 32768
Recieved 84 Chunks till now.
Size: 32768
Recieved 85 Chunks till now.
Size: 32768
Recieved 86 Chunks till now.
Size: 32768
Recieved 87 Chunks till now.
Size: 32768
Recieved 88 Chunks till now.
Size: 32768
Recieved 89 Chunks till now.
Size: 32768
Recieved 90 Chunks till now.
Size: 32768
Recieved 91 Chunks till now.
```

```
Size: 32768
Recieved 77 Chunks till now.
Size: 32768
Recieved 78 Chunks till now.
Size: 32768
Recieved 79 Chunks till now.
Size: 32768
Recieved 80 Chunks till now.
Size: 32768
Recieved 81 Chunks till now.
Size: 32768
Recieved 82 Chunks till now.
Size: 32768
Recieved 83 Chunks till now.
Size: 32768
Recieved 84 Chunks till now.
Size: 32768
Recieved 85 Chunks till now.
```

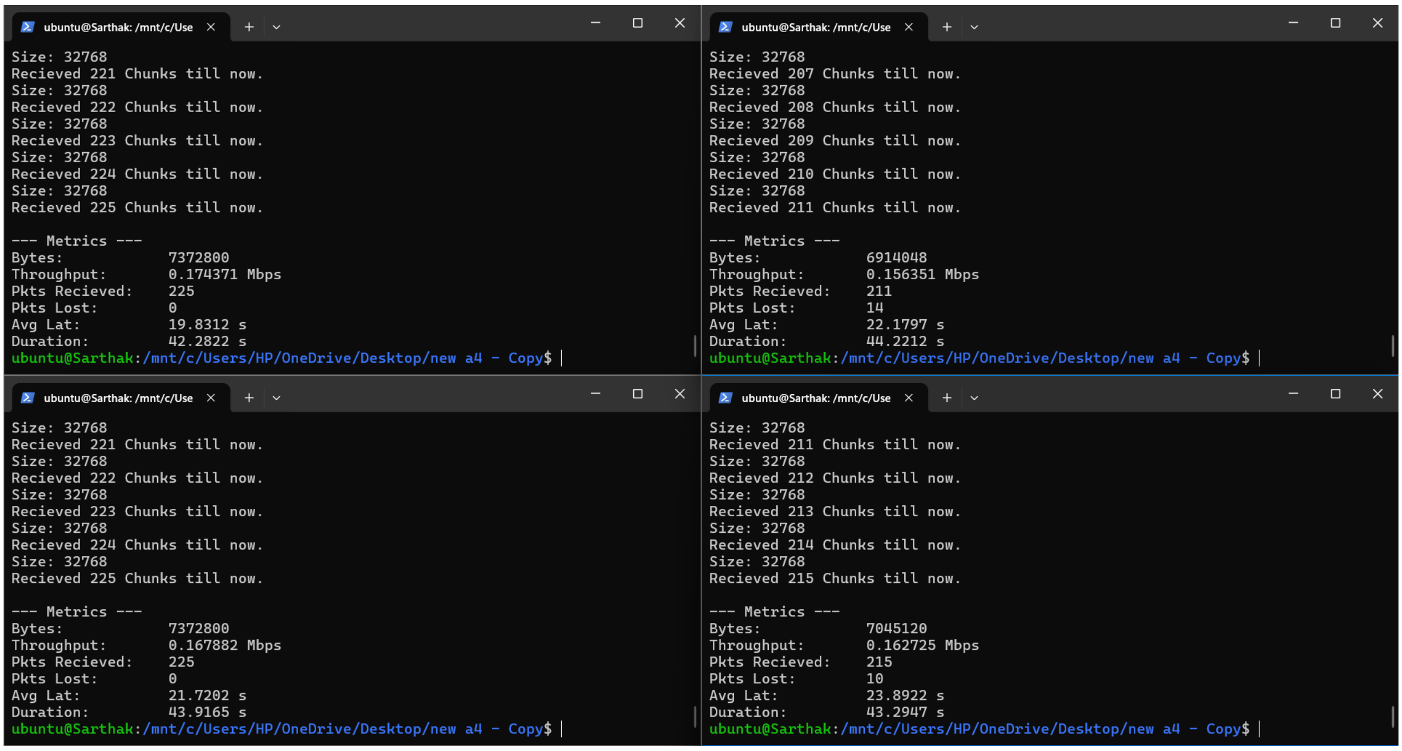
```
Size: 32768
Recieved 68 Chunks till now.
Size: 32768
Recieved 69 Chunks till now.
Size: 32768
Recieved 70 Chunks till now.
Size: 32768
Recieved 71 Chunks till now.
Size: 32768
Recieved 72 Chunks till now.
Size: 32768
Recieved 73 Chunks till now.
Size: 32768
Recieved 74 Chunks till now.
Size: 32768
Recieved 75 Chunks till now.
Size: 32768
Recieved 76 Chunks till now.
```

```
Size: 32768
Recieved 48 Chunks till now.
Size: 32768
Recieved 49 Chunks till now.
Size: 32768
Recieved 50 Chunks till now.
Size: 32768
Recieved 51 Chunks till now.
Size: 32768
Recieved 52 Chunks till now.
Size: 32768
Recieved 53 Chunks till now.
Size: 32768
Recieved 54 Chunks till now.
Size: 32768
Recieved 55 Chunks till now.
Size: 32768
Recieved 56 Chunks till now.
```

In RR all will works. On the 2 threads after the fixed time quantum threads will pre-empt. And give chance to other thread.

<u>Results:</u>

In RR Mode



```
Size: 32768
Recieved 221 Chunks till now.
Size: 32768
Recieved 222 Chunks till now.
Size: 32768
Recieved 223 Chunks till now.
Size: 32768
Recieved 224 Chunks till now.
Size: 32768
Recieved 225 Chunks till now.

--- Metrics ---
Bytes:            7372800
Throughput:       0.174371 Mbps
Pkts Recieved:    225
Pkts Lost:        0
Avg Lat:          19.8312 s
Duration:         42.2822 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

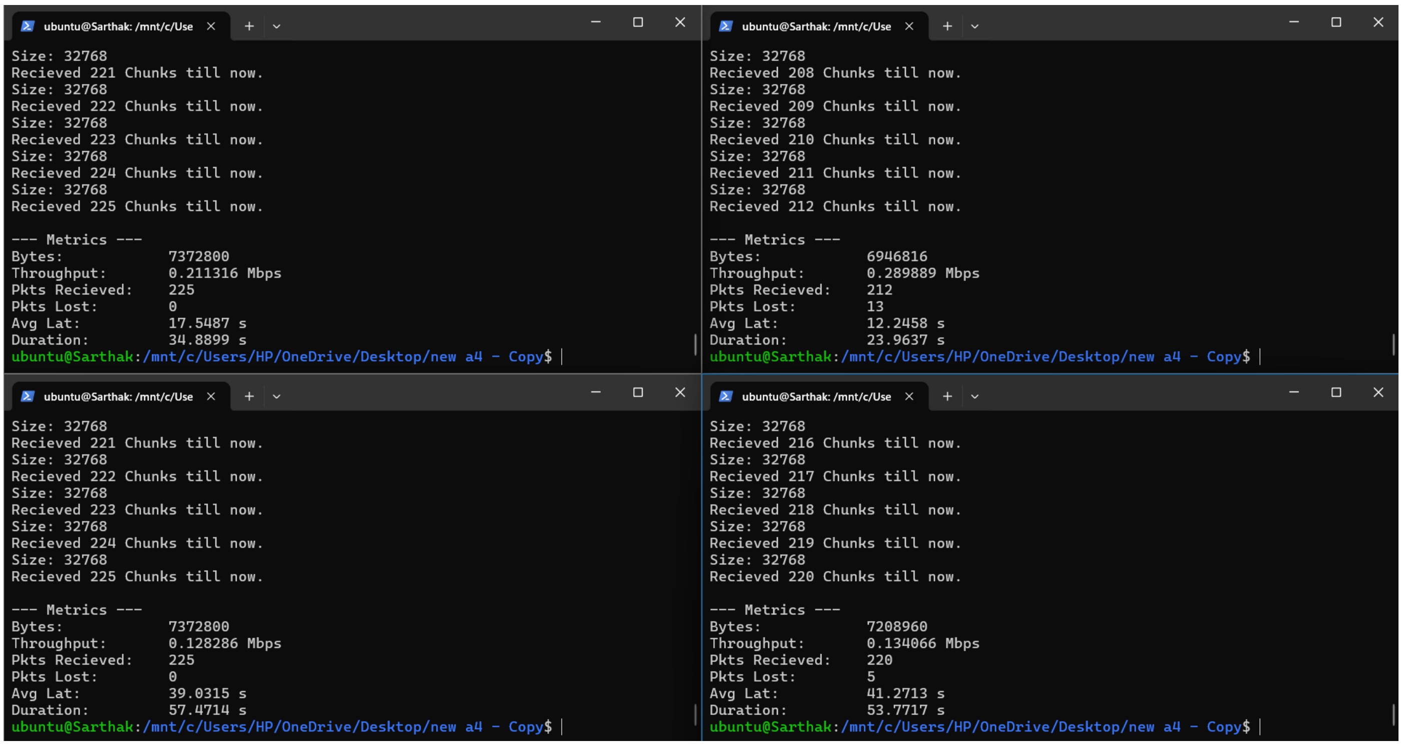
```
Size: 32768
Recieved 207 Chunks till now.
Size: 32768
Recieved 208 Chunks till now.
Size: 32768
Recieved 209 Chunks till now.
Size: 32768
Recieved 210 Chunks till now.
Size: 32768
Recieved 211 Chunks till now.

--- Metrics ---
Bytes:            6914048
Throughput:       0.156351 Mbps
Pkts Recieved:    211
Pkts Lost:        14
Avg Lat:          22.1797 s
Duration:         44.2212 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

```
Size: 32768
Recieved 221 Chunks till now.
Size: 32768
Recieved 222 Chunks till now.
Size: 32768
Recieved 223 Chunks till now.
Size: 32768
Recieved 224 Chunks till now.
Size: 32768
Recieved 225 Chunks till now.

--- Metrics ---
Bytes:            7372800
Throughput:       0.167882 Mbps
Pkts Recieved:    225
Pkts Lost:        0
Avg Lat:          21.7202 s
Duration:         43.9165 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

```
Size: 32768
Recieved 211 Chunks till now.
Size: 32768
Recieved 212 Chunks till now.
Size: 32768
Recieved 213 Chunks till now.
Size: 32768
Recieved 214 Chunks till now.
Size: 32768
Recieved 215 Chunks till now.

--- Metrics ---
Bytes:            7045120
Throughput:       0.162725 Mbps
Pkts Recieved:    215
Pkts Lost:        10
Avg Lat:          23.8922 s
Duration:         43.2947 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

In RR, Throughput for all the client will be more that FCFS because. All the threads are getting chance. By taking one-by-one resources.

In FCFS



```
Size: 32768
Recieved 221 Chunks till now.
Size: 32768
Recieved 222 Chunks till now.
Size: 32768
Recieved 223 Chunks till now.
Size: 32768
Recieved 224 Chunks till now.
Size: 32768
Recieved 225 Chunks till now.

--- Metrics ---
Bytes:             7372800
Throughput:        0.211316 Mbps
Pkts Recieved:     225
Pkts Lost:         0
Avg Lat:           17.5487 s
Duration:          34.8899 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

```
Size: 32768
Recieved 208 Chunks till now.
Size: 32768
Recieved 209 Chunks till now.
Size: 32768
Recieved 210 Chunks till now.
Size: 32768
Recieved 211 Chunks till now.
Size: 32768
Recieved 212 Chunks till now.

--- Metrics ---
Bytes:             6946816
Throughput:        0.289889 Mbps
Pkts Recieved:     212
Pkts Lost:         13
Avg Lat:           12.2458 s
Duration:          23.9637 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

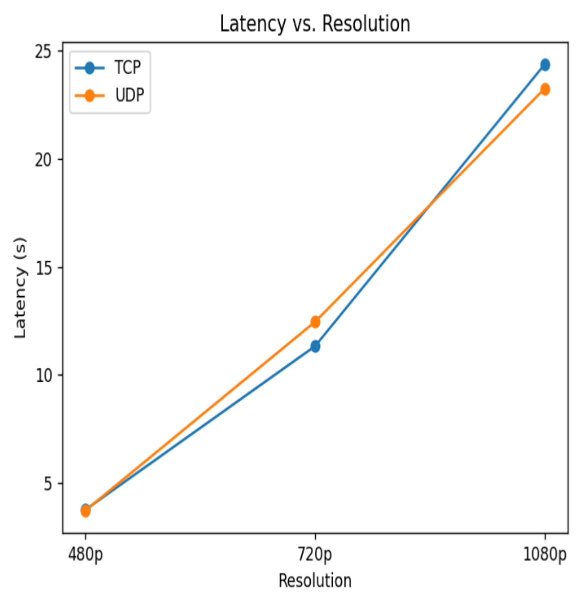
```
Size: 32768
Recieved 221 Chunks till now.
Size: 32768
Recieved 222 Chunks till now.
Size: 32768
Recieved 223 Chunks till now.
Size: 32768
Recieved 224 Chunks till now.
Size: 32768
Recieved 225 Chunks till now.

--- Metrics ---
Bytes:             7372800
Throughput:        0.128286 Mbps
Pkts Recieved:     225
Pkts Lost:         0
Avg Lat:           39.0315 s
Duration:          57.4714 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

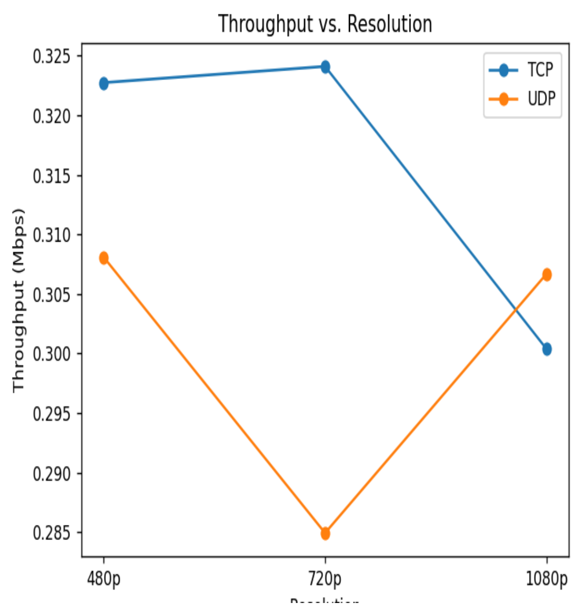
```
Size: 32768
Recieved 216 Chunks till now.
Size: 32768
Recieved 217 Chunks till now.
Size: 32768
Recieved 218 Chunks till now.
Size: 32768
Recieved 219 Chunks till now.
Size: 32768
Recieved 220 Chunks till now.

--- Metrics ---
Bytes:             7208960
Throughput:        0.134066 Mbps
Pkts Recieved:     220
Pkts Lost:         5
Avg Lat:           41.2713 s
Duration:          53.7717 s
ubuntu@Sarthak:/mnt/c/Users/HP/OneDrive/Desktop/new a4 - Copy$
```

In FCFS, throughput will be high for the 2 client which are invoked later. But it will high for the one which are invoked first.

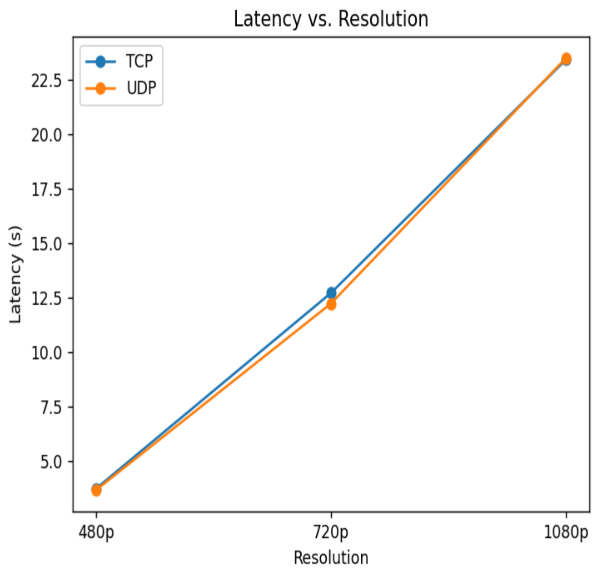
Resulting Graphs:

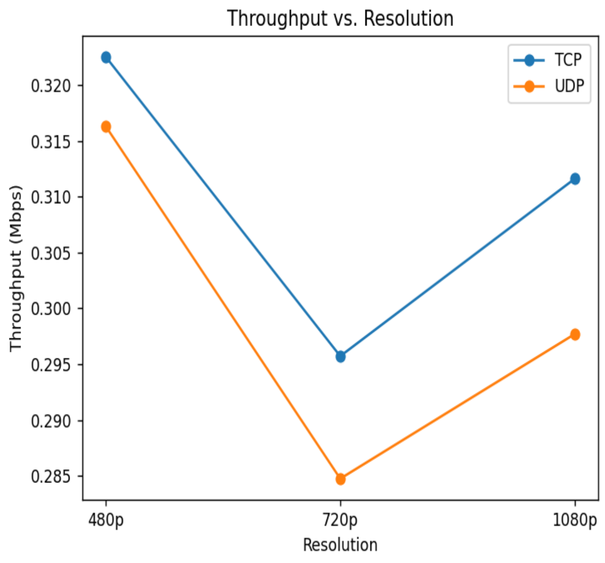
In FCFS





In RR


Latency vs. Resolution
TCP
UDP
Latency (s)
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
480p
720p
1080p
Resolution


Throughput vs. Resolution
TCP
UDP
Throughput (Mbps)
0.320
0.315
0.310
0.305
0.300
0.295
0.290
0.285
480p
720p
1080p
Resolution