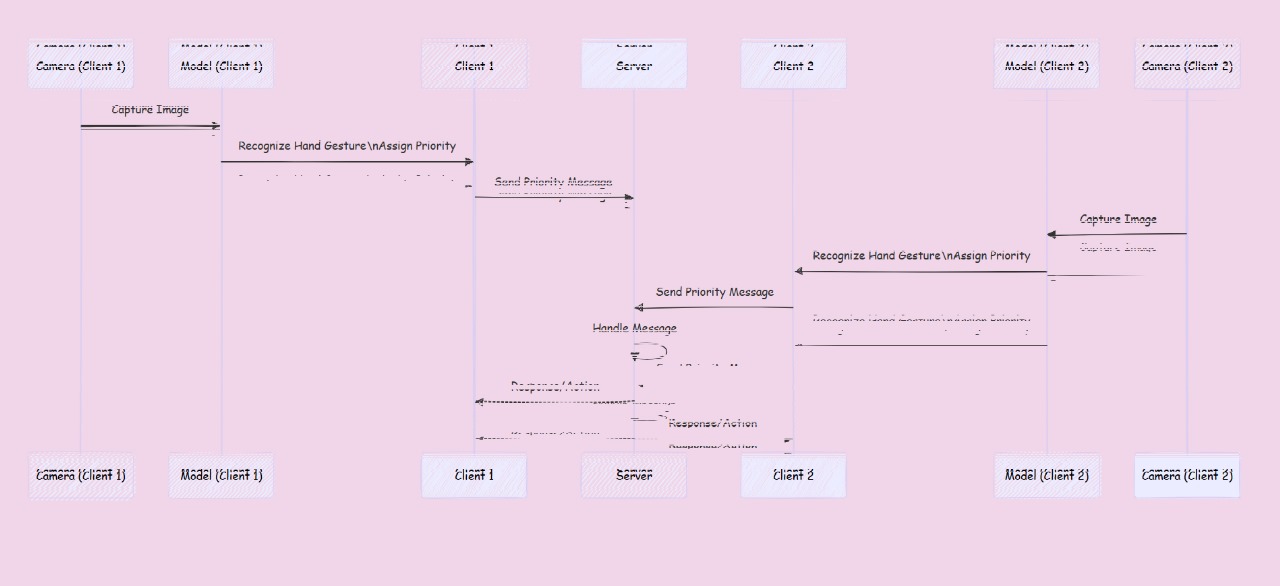
**Project Report: Client-Server Priority Based Intelligent Hand Gesture Analysis System**

## Project Members

1. Hafsa Javed, 22pwcse2171
2. Fiza Nihar, 22pwcse2100
3. Syeda Afia Shah,

**Introduction**

This project involves developing a client-server system where images are sent to an AI model for analysis. The AI model is trained to recognize two specific gestures: one indicating domestic violence and the other indicating correction. Based on the analysis, the AI model assigns a priority to each image, and clients with higher priority are served first by the server. This report details the design, implementation, and components of the system.



**Objectives**

1. Implement a server that can handle multiple clients simultaneously.
2. Use priority-based scheduling to manage client requests.
3. Integrate an AI model to analyze images and assign priorities.
4. Ensure the system is robust and efficient in handling real-time data.

**Components**

1. **Server Code**: Handles client connections, manages priorities, and schedules clients.
2. **Client Code**: Sends images and priorities to the server.
3. **ML Model**: (To be integrated) Analyzes images and assigns priorities based on recognized gestures.

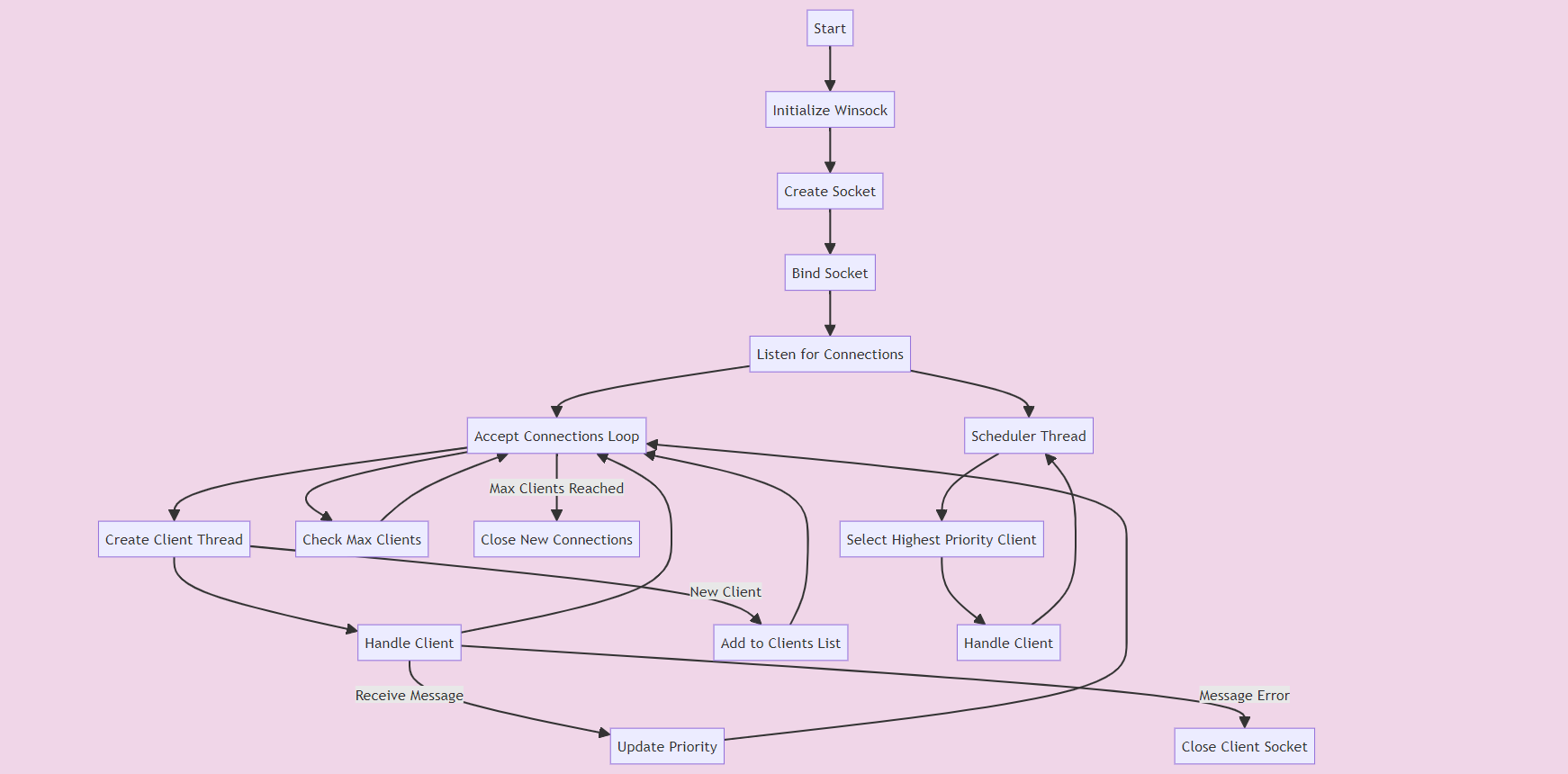
**List of System Calls Used**

**Server Code System Calls:**

1. **WSAStartup**: Initializes the Winsock library.
2. **socket**: Creates a socket for network communication.
3. **bind**: Binds the socket to an IP address and port number.
4. **listen**: Listens for incoming connections on the socket.
5. **accept**: Accepts an incoming connection from a client.
6. **recv**: Receives data from a connected socket.
7. **closesocket**: Closes an existing socket.
8. **WSACleanup**: Cleans up the Winsock library.

**Client Code System Calls:**

1. **WSAStartup**: Initializes the Winsock library.
2. **socket**: Creates a socket for network communication.
3. **gethostbyname**: Resolves a hostname to an IP address.
4. **connect**: Establishes a connection to a server.
5. **send**: Sends data to a connected socket.
6. **closesocket**: Closes an existing socket.
7. **WSACleanup**: Cleans up the Winsock library.

**Server Code Explanation**

**Initialization and Setup**

1. **Winsock Initialization**:
   * The code starts by initializing the Winsock library using WSAStartup. This step is crucial for setting up network communication in a Windows environment.

c

Copy code

printf("Initializing Winsock...\n");

if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {

error("Failed to initialize Winsock");

}

1. **Socket Creation**:
   * A socket is created using socket(AF\_INET, SOCK\_STREAM, 0). This socket will be used for listening to incoming client connections.

c

Copy code

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == INVALID\_SOCKET) {

error("Error opening socket");

}

1. **Binding**:
   * The socket is bound to a specific port, provided as a command-line argument. The server can then listen for incoming connections on this port.

c

Copy code

memset((char \*)&server\_addr, 0, sizeof(server\_addr));

portno = atoi(argv[1]);

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(portno);

if (bind(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) == SOCKET\_ERROR) {

error("Error on binding");

}

1. **Listening**:
   * The server starts listening for incoming connections using the listen(sockfd, 5) call. This means the server can queue up to 5 connection requests before it starts rejecting them.

c

Copy code

listen(sockfd, 5);

clilen = sizeof(client\_addr);

**Client Handling**

1. **Accepting Connections**:
   * The server enters a loop where it continuously accepts new client connections using accept. Each accepted connection results in a new socket (newsockfd).

c

Copy code

while (client\_count < MAX\_CLIENTS) {

newsockfd = accept(sockfd, (struct sockaddr \*)&client\_addr, &clilen);

if (newsockfd == INVALID\_SOCKET) {

error("Error on accept");

}

clients[client\_count].sockfd = newsockfd;

clients[client\_count].priority = -1;

pthread\_create(&threads[client\_count], NULL, handle\_client, (void \*)&clients[client\_count]);

client\_count++;

}

1. **Client Information Storage**:
   * Each new client connection is stored in the clients array, which holds the socket and priority information for each client.

c

Copy code

client\_info clients[MAX\_CLIENTS];

int client\_count = 0;

pthread\_mutex\_t lock;

1. **Client Handling Thread**:
   * For each new client, a separate thread is created to handle communication with the client. The handle\_client function is executed in these threads.

c

Copy code

pthread\_t threads[MAX\_CLIENTS];

pthread\_create(&threads[client\_count], NULL, handle\_client, (void \*)&clients[client\_count]);

1. **Receiving Data**:
   * Within handle\_client, the server receives data from the client. This data is expected to be the client's priority, which is updated in the clients array.

c

Copy code

void \*handle\_client(void \*arg) {

int n;

char buffer[255];

client\_info \*client = (client\_info \*)arg;

while (1) {

memset(buffer, 0, 255);

n = recv(client->sockfd, buffer, 255, 0);

if (n == SOCKET\_ERROR) {

error("Error reading from socket");

}

if (n == 0) {

break;

}

int priority = atoi(buffer);

if (priority < 0 || priority > 3) {

printf("Invalid priority received\n");

continue;

}

pthread\_mutex\_lock(&lock);

client->priority = priority;

pthread\_mutex\_unlock(&lock);

printf("Received from client (priority %d): %s\n", client->priority, buffer);

}

closesocket(client->sockfd);

return NULL;

}

1. **Priority Management**:
   * The priority of each client is managed using a mutex (pthread\_mutex\_lock and pthread\_mutex\_unlock) to ensure thread-safe operations when updating the priority.

c

Copy code

pthread\_mutex\_lock(&lock);

client->priority = priority;

pthread\_mutex\_unlock(&lock);

**Scheduling**

1. **Scheduler Thread**:
   * A separate scheduler thread is created to manage client scheduling. This thread runs the schedule\_clients function.

c

Copy code

pthread\_t scheduler\_thread;

pthread\_create(&scheduler\_thread, NULL, (void \*(\*)(void \*))schedule\_clients, NULL);

1. **Client Selection**:
   * The scheduler identifies the client with the highest priority. If multiple clients have the same priority, it uses a round-robin mechanism to decide which client to handle next.

c

Copy code

void schedule\_clients() {

int round\_robin\_index = 0;

while (1) {

pthread\_mutex\_lock(&lock);

int highest\_priority = -1;

int highest\_priority\_index = -1;

int same\_priority\_count = 0;

for (int i = 0; i < client\_count; i++) {

if (clients[i].priority > highest\_priority) {

highest\_priority = clients[i].priority;

highest\_priority\_index = i;

same\_priority\_count = 1;

} else if (clients[i].priority == highest\_priority) {

same\_priority\_count++;

}

}

if (highest\_priority != -1) {

if (same\_priority\_count > 1) {

int start\_index = round\_robin\_index % same\_priority\_count;

int handled\_client\_index = -1;

for (int i = 0; i < client\_count; i++) {

if (clients[i].priority == highest\_priority) {

if (start\_index == 0) {

handled\_client\_index = i;

break;

}

start\_index--;

}

}

printf("Handling priority %d client\n", clients[handled\_client\_index].priority);

clients[handled\_client\_index].priority = -1; // Reset priority

round\_robin\_index++;

} else {

printf("Handling priority %d client\n", clients[highest\_priority\_index].priority);

clients[highest\_priority\_index].priority = -1; // Reset priority

}

}

pthread\_mutex\_unlock(&lock);

Sleep(5000); // Sleep to prevent busy waiting

}

}

1. **Priority Reset**:
   * Once a client is selected and handled, its priority is reset to -1, indicating that it needs to send a new priority to be considered again.

c

Copy code

clients[handled\_client\_index].priority = -1; // Reset priority

1. **Sleep to Prevent Busy Waiting**:
   * The scheduler thread includes a sleep interval (Sleep(5000)) to prevent busy waiting and reduce CPU usage.

c

Copy code

Sleep(5000); // Sleep to prevent busy waiting

**Code Structure and Flow**

* **Main Function**:
  + The main function sets up the server, including initializing Winsock, creating and binding the socket, and starting the listening process. It also starts the scheduler thread and enters a loop to accept and handle client connections.

c

Copy code

int main(int argc, char \*argv[]) {

if (argc < 2) {

fprintf(stderr, "Usage: %s <port>\n", argv[0]);

exit(1);

}

WSADATA wsa;

SOCKET sockfd, newsockfd;

struct sockaddr\_in server\_addr, client\_addr;

int portno;

int clilen;

pthread\_t threads[MAX\_CLIENTS];

printf("Initializing Winsock...\n");

if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0) {

error("Failed to initialize Winsock");

}

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == INVALID\_SOCKET) {

error("Error opening socket");

}

memset((char \*)&server\_addr, 0, sizeof(server\_addr));

portno = atoi(argv[1]);

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(portno);

if (bind(sockfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) == SOCKET\_ERROR) {

error("Error on binding");

}

listen(sockfd, 5);

clilen = sizeof(client\_addr);

pthread\_mutex\_init(&lock, NULL);

// Start the scheduler thread

pthread\_t scheduler\_thread;

pthread\_create(&scheduler\_thread, NULL, (void \*(\*)(void \*))schedule\_clients, NULL);

while (client\_count < MAX\_CLIENTS) {

newsockfd = accept(sockfd, (struct sockaddr \*)&client\_addr, &clilen);

if (newsockfd == INVALID\_SOCKET) {

error("Error on accept");

}

clients[client\_count].sockfd = newsockfd;

clients[client\_count].priority = -1;

pthread\_create(&threads[client\_count], NULL, handle\_client, (void \*)&clients[client\_count]);

client\_count++;

}

for (int i = 0; i < client\_count; i++) {

pthread\_join(threads[i], NULL);

}

closesocket(sockfd);

pthread\_mutex\_destroy(&lock);

WSACleanup();

return 0;

}

* **handle\_client Function**:
  + This function runs in a separate thread for each client. It continuously receives data from the client, updates the client's priority, and prints received messages.

c

Copy code

void \*handle\_client(void \*arg) {

int n;

char buffer[255];

client\_info \*client = (client\_info \*)arg;

while (1) {

memset(buffer, 0, 255);

n = recv(client->sockfd, buffer, 255, 0);

if (n == SOCKET\_ERROR) {

error("Error reading from socket");

}

if (n == 0) {

break;

}

int priority = atoi(buffer);

if (priority < 0 || priority > 3) {

printf("Invalid priority received\n");

continue;

}

pthread\_mutex\_lock(&lock);

client->priority = priority;

pthread\_mutex\_unlock(&lock);

printf("Received from client (priority %d): %s\n", client->priority, buffer);

}

closesocket(client->sockfd);

return NULL;

}

* **schedule\_clients Function**:
  + This function runs in the scheduler thread. It selects and handles clients based on their priorities using a combination of priority and round-robin scheduling.

c

Copy code

void schedule\_clients() {

int round\_robin\_index = 0;

while (1) {

pthread\_mutex\_lock(&lock);

int highest\_priority = -1;

int highest\_priority\_index = -1;

int same\_priority\_count = 0;

for (int i = 0; i < client\_count; i++) {

if (clients[i].priority > highest\_priority) {

highest\_priority = clients[i].priority;

highest\_priority\_index = i;

same\_priority\_count = 1;

} else if (clients[i].priority == highest\_priority) {

same\_priority\_count++;

}

}

if (highest\_priority != -1) {

if (same\_priority\_count > 1) {

int start\_index = round\_robin\_index % same\_priority\_count;

int handled\_client\_index = -1;

for (int i = 0; i < client\_count; i++) {

if (clients[i].priority == highest\_priority) {

if (start\_index == 0) {

handled\_client\_index = i;

break;

}

start\_index--;

}

}

printf("Handling priority %d client\n", clients[handled\_client\_index].priority);

clients[handled\_client\_index].priority = -1; // Reset priority

round\_robin\_index++;

} else {

printf("Handling priority %d client\n", clients[highest\_priority\_index].priority);

clients[highest\_priority\_index].priority = -1; // Reset priority

}

}

pthread\_mutex\_unlock(&lock);

Sleep(5000); // Sleep to prevent busy waiting

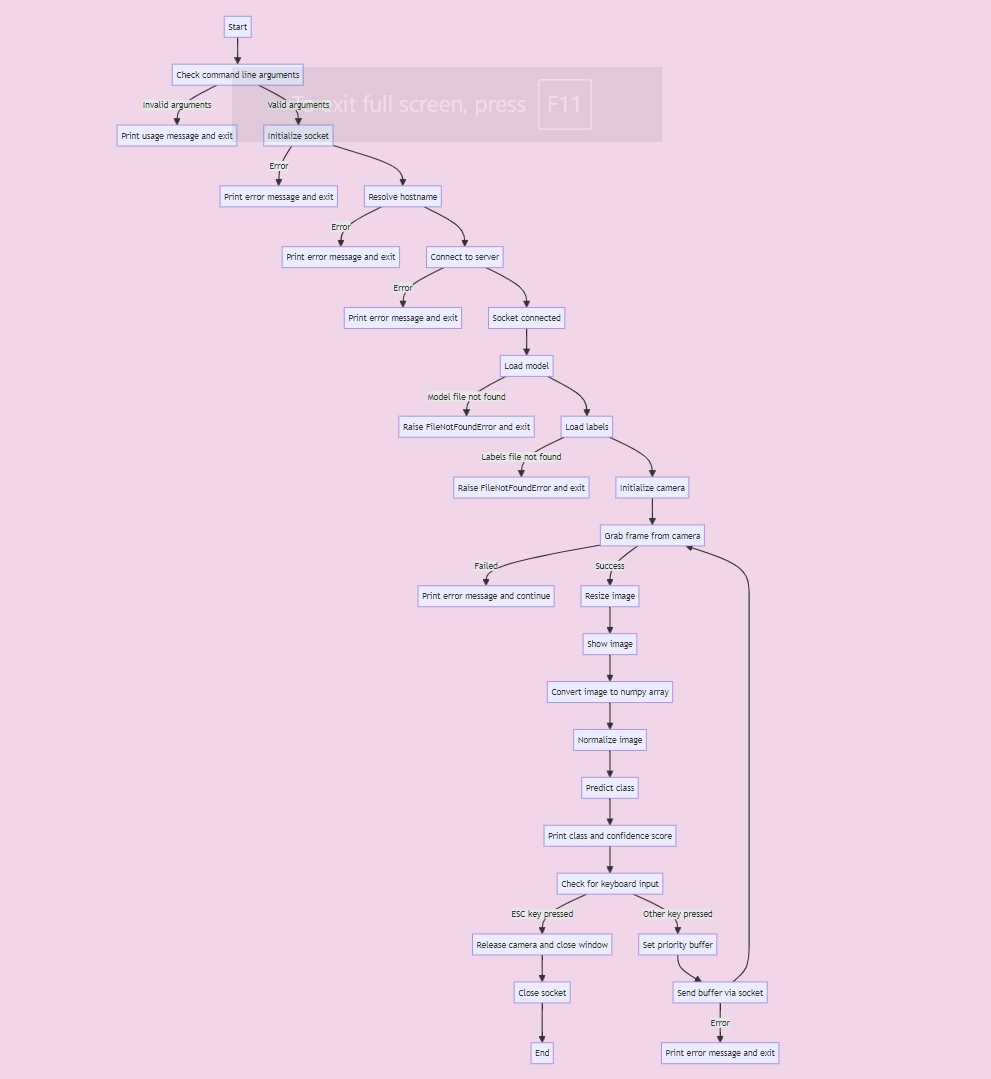
}

}

Overall, the code implements a multi-threaded server that can handle multiple clients, prioritizes clients based on received priority values, and uses round-robin scheduling for clients with the same priority. The structure ensures efficient handling and scheduling of clients while maintaining thread safety through the use of mutexes.

# Client side

## Code Explanation



### Client Application Detailed Description

### System Calls Used

* socket.socket(): Creates a new socket.
* socket.gethostbyname(): Resolves the hostname to an IP address.
* socket.connect(): Connects to the server using the resolved IP and port number.
* cv2.VideoCapture(): Captures video from the webcam.
* cv2.resize(): Resizes the captured image to the required dimensions.
* cv2.imshow(): Displays the image in a window.
* np.asarray(): Converts the image to a NumPy array.
* np.argmax(): Finds the index of the highest prediction score.
* cv2.waitKey(): Waits for a key event.
* sock.sendall(): Sends data to the server.
* time.sleep(): Introduces a delay between sending data.
* cv2.destroyAllWindows(): Closes all OpenCV windows.

#### Initialization and Setup

The client application starts by importing necessary libraries and defining utility functions for error handling and socket management.

python

Copy code

import tensorflow as tf

import keras

from keras.models import load\_model

import cv2 # Ensure you have opencv-python installed

import numpy as np

import os

import socket

import sys

import time

def error(msg):

print(f"Error: {msg}")

sys.exit(1)

#### Socket Setup

A function socket\_func is defined to handle socket initialization, resolving the server address, and establishing a connection.

python

Copy code

def socket\_func() -> socket:

if len(sys.argv) < 3:

print(f"Usage: {sys.argv[0]} <hostname> <port>")

sys.exit(1)

hostname = sys.argv[1]

portno = int(sys.argv[2])

try:

sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

except socket.error as err:

error(f"Error opening socket: {err}")

try:

server\_ip = socket.gethostbyname(hostname)

except socket.error as err:

error(f"Error, no such host: {err}")

server\_addr = (server\_ip, portno)

try:

sock.connect(server\_addr)

except socket.error as err:

error(f"Error connecting: {err}")

return sock

#### Model Loading and Image Processing

The model\_func function handles loading the pre-trained AI model, capturing images from the webcam, processing the images, and sending the classification results to the server.

python

Copy code

def model\_func(sock: socket):

np.set\_printoptions(suppress=True)

model\_path = "./model.savedmodel/"

if not os.path.exists(model\_path):

raise FileNotFoundError(f"Model file '{model\_path}' not found. Please ensure the file is in the correct location.")

model = tf.saved\_model.load("model.savedModel")

labels\_path = "labels.txt"

if not os.path.exists(labels\_path):

raise FileNotFoundError(f"Labels file '{labels\_path}' not found. Please ensure the file is in the correct location.")

class\_names = open(labels\_path, "r").readlines()

camera = cv2.VideoCapture(0)

while True:

ret, image = camera.read()

if not ret:

print("Failed to grab frame")

continue

image\_resized = cv2.resize(image, (224, 224), interpolation=cv2.INTER\_AREA)

cv2.imshow("Webcam Image", image\_resized)

image\_array = np.asarray(image\_resized, dtype=np.float32).reshape(1, 224, 224, 3)

image\_normalized = (image\_array / 127.5) - 1

prediction = model(image\_normalized)

index = np.argmax(prediction)

class\_name = class\_names[index][2]

confidence\_score = prediction[0][index]

print(f"Class: {class\_name.strip()} Confidence Score: {confidence\_score:.2%}")

keyboard\_input = cv2.waitKey(1)

if keyboard\_input == 27:

break

buffer = class\_name.strip()

if buffer not in ["1", "2"]:

buffer = "0"

try:

sock.sendall(buffer.encode())

time.sleep(0.25)

except socket.error as err:

error(f"Error writing to socket: {err}")

camera.release()

cv2.destroyAllWindows()

sock.close()

# Output

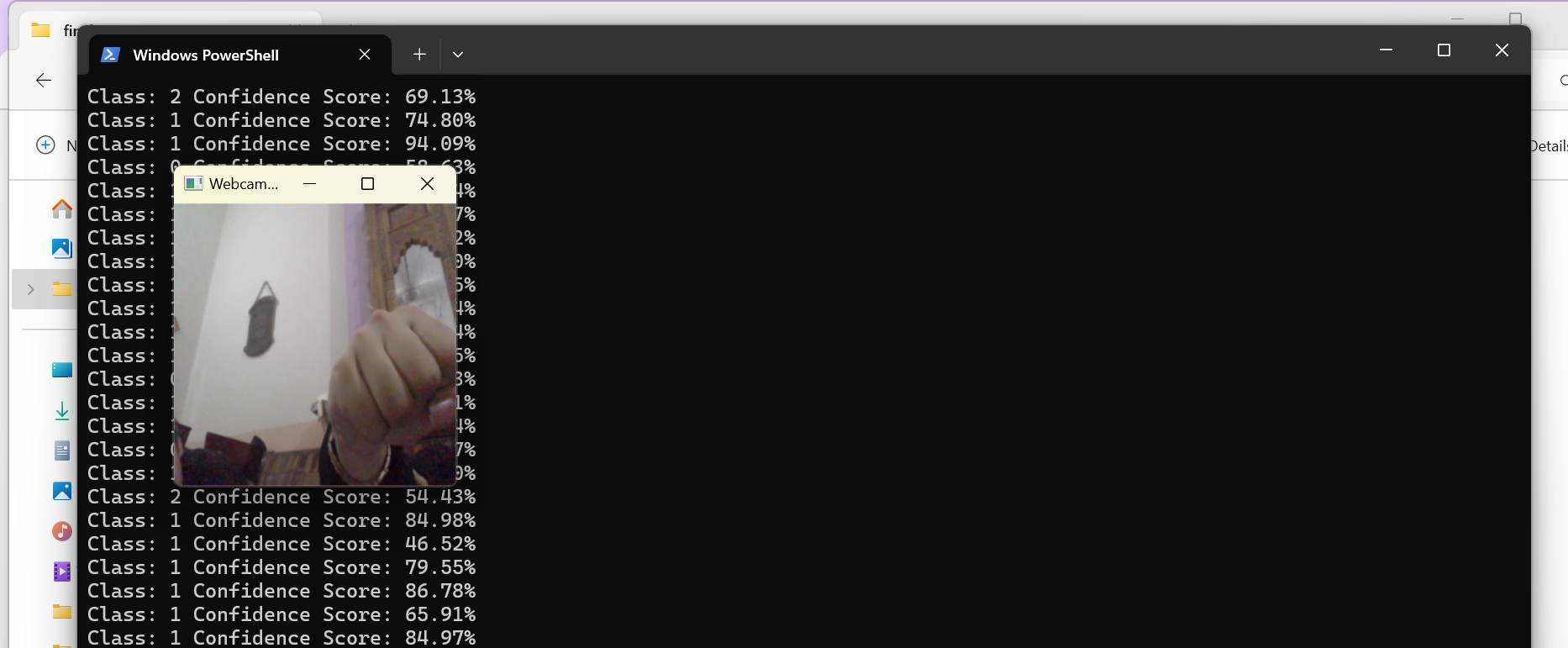


Figure 1- Client 1

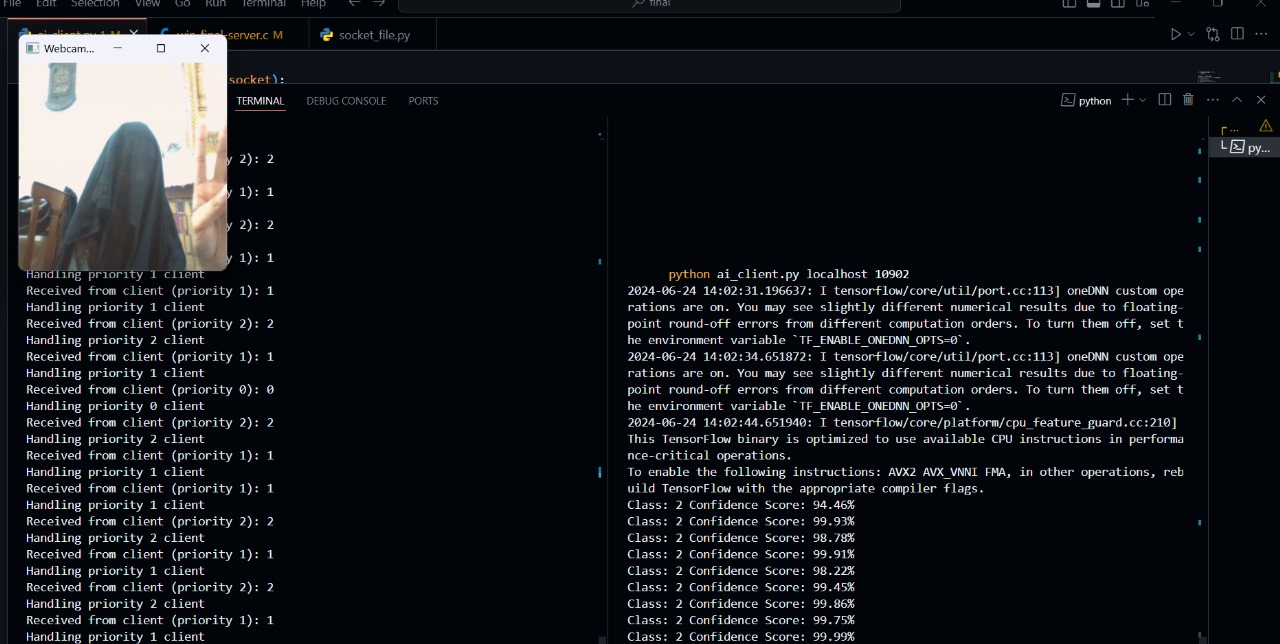


Figure 2 - Client 2 and Server

## **Future Work and Improvements**

### 1. Enhanced Gesture Recognition

* **Expand Gesture Library**: Train the AI model to recognize more gestures.
* **Real-Time Updates**: Implement real-time training for new gestures.

### 2. Improved Model Accuracy and Performance

* **Advanced Preprocessing**: Use data augmentation techniques.
* **Model Optimization**: Apply pruning, quantization, or knowledge distillation.
* **Ensemble Models**: Use multiple models to improve accuracy.

### 3. Integration with Other Safety Systems

* **Cross-Platform Integration**: Develop APIs for integration with other systems.
* **IoT Device Integration**: Extend to smartwatches and home assistants.

### 4. Enhanced Client-Server Communication

* **Secure Communication**: Implement SSL/TLS encryption.
* **Load Balancing**: Distribute workload across multiple servers.
* **Scalability**: Optimize server to handle more connections.

### 5. User Interface and User Experience

* **Graphical User Interface (GUI)**: Develop a user-friendly GUI.
* **Mobile Application**: Create mobile apps for Android and iOS.
* **Accessibility Features**: Add voice commands and text-to-speech.

### 6. Advanced Scheduling Algorithms

* **Dynamic Scheduling**: Implement priority inheritance or EDF algorithms.
* **Machine Learning-Based Scheduling**: Use ML for better client prioritization.

### 7. Comprehensive Logging and Monitoring

* **Detailed Logging**: Enhance logs for better troubleshooting.
* **Real-Time Monitoring**: Implement dashboards for system performance.
* **Anomaly Detection**: Use ML to detect security threats.

### 8. Extensive Testing and Validation

* **Unit and Integration Testing**: Develop comprehensive tests.
* **User Acceptance Testing (UAT)**: Conduct extensive UAT.
* **CI/CD Pipelines**: Automate testing, building, and deployment.

### 9. Documentation and Training

* **Comprehensive Documentation**: Create detailed system documentation.
* **Training Programs**: Develop materials for users and administrators.
* **Community Support**: Establish forums and support systems.

### 10. Ethical and Legal Considerations

* **Privacy Policies**: Ensure compliance with data protection regulations.
* **Ethical AI Use**: Avoid biases and ensure fairness.
* **Legal Compliance**: Regularly update to meet legal standards.